Modern C++ Programming

1. Introduction

Federico Busato

2024-03-29

Table of Contents

"When recruiting research assistants, I look at grades as the last indicator. I find that ***imagination, ambition, initiative, curiosity, drive****, are far better predictors of someone who will do useful work with me. Of course, these characteristics are themselves correlated with high grades, but there is something to be said about a student who decides that a given course is a waste of time and that he works on a side project instead.*

Breakthroughs don't happen in regular scheduled classes, they happen in side projects. We want people who complete the work they were assigned, but ***we also need people who can reflect critically on what is genuinely important****"*

Daniel Lemire*, Prof. at the University of Quebec*

Academic excellence is not a strong predictor of career excellence

"Across industries, research shows that the correlation between grades and job performance is modest in the first year after college and trivial within a handful of years...
Academic grades rarely assess qualities like creativity, leadership and teamwork skills, or social, emotional and political intelligence. Yes, straight-A students master cramming information and regurgitating it on exams. But ***career success is rarely about finding the right solution to a problem — it's more about finding the right problem to solve****..."*

"Getting straight A's requires conformity. ***Having an influential career demands originality.***

This might explain why Steve Jobs finished high school with a 2.65 G.P.A., J.K. Rowling graduated from the University of Exeter with roughly a C average, and the Rev. Dr. Martin Luther King Jr. got only one A in his four years at Morehouse

If your goal is to graduate without a blemish on your transcript, you end up taking easier classes and staying within your comfort zone. If you're willing to tolerate the occasional B... ***You gain experience coping with failures and setbacks, which builds resilience****"*

"Straight-A students also miss out socially. More time studying in the library means less time to start lifelong friendships, join new clubs or volunteer...Looking back, I don't wish my grades had been higher. If I could do it over again, I'd study less"

***Adam Grant**, the New York Times*

www.nytimes.com/2018/12/08/opinion/college-gpa-career-success.html

"Got a 2.4 GPA my first semester in college. Thought maybe I wasn't cut out for engineering. Today I've landing two spacecraft on Mars, and designing one for the moon.

STEM is hard for everyone. Grades ultimately aren't what matters. ***Curiosity and persistence matter****"*

Ben Cichy*, Chief Software Engineer,*
NASA Mars Science Laboratory

twitter.com/bencichy/status/1197752802929364992?s=20

"And programming computers was so fascinating. You create your own little universe, and then it does what you tell it to do"

***Vint Cerf**, TCP/IP co-inventor and Turing Award*

"Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program"

***Linus Torvalds**, principal developer of the Linux kernel*

"You might not think that programmers are artists, but programming is an extremely creative profession. It's logic-based creativity"

***John Romero**, co-founder of id Software*

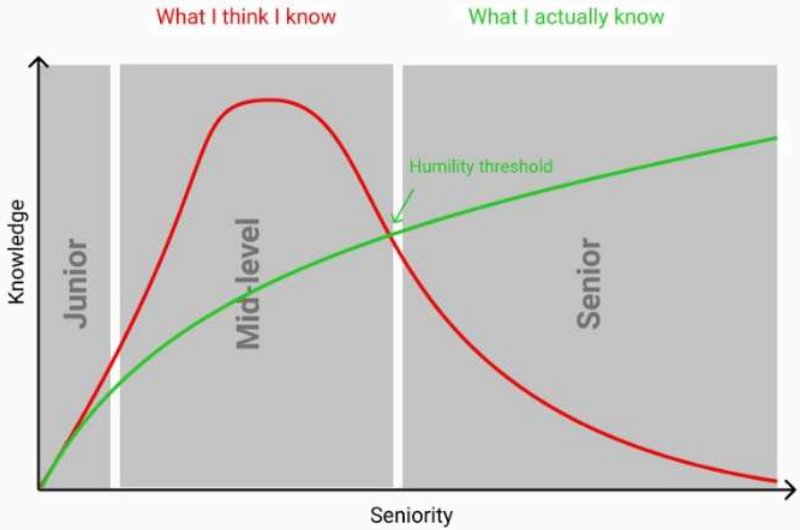
Creativity *Programming is extremely creative. The ability to perceive the problem in a novel way, provide new and original solutions. Creativity allows recognizing and generating alternatives*

Form of Art *Art is the expression of human creative skills. Every programmer has his own style. Codes and algorithms show elegance and beauty in the same way as painting or music*

Learn *Programming gives the opportunity to learn new things every day, improve own skills and knowledge*

Challenge *Programming is a challenge. A challenge against yourself, the problem, and the environment*

Knowledge-Experience Relation



Learning

"In software development, learning is not a big part of the job. ***It is the job****."*

Woody Zuill

A Little History of C/C++ Programming Language

The Assembly Programming Language

A long time ago, in a galaxy far, far away....there was **Assembly**

- Extremely simple instructions
- Requires lots of code to do simple tasks
- Can express anything your computer can do
- Hard to read, write
- ...redundant, boring programming, bugs proliferation

```
main:
.Lfunc_begin0:
    push rbp
.Lcfi0:
.Lcfi1:
    mov rbp, rsp
.Lcfi2:
    sub rsp, 16
    movabs rdi, .L.str
.Ltmp0:
    mov al, 0
    call printf
    xor ecx, ecx
    mov dword ptr [rbp - 4], eax
    mov eax, ecx
    add rsp, 16
    pop rbp
    ret
.Ltmp1:
.Lfunc_end0:
.L.str:
.asciz "Hello World\n"
```

In the 1969 **Dennis M. Ritchie** and **Ken Thompson** (AT&T, Bell Labs) worked on developing an operating system for a large computer that could be used by a thousand users. The new operating system was called **UNIX**

The whole system was still written in assembly code. Besides assembler and Fortran, UNIX also had an interpreter for the **programming language B**. A high-level language like B made it possible to write many pages of code task in just a few lines of code. In this way the code could be produced much faster than in assembly

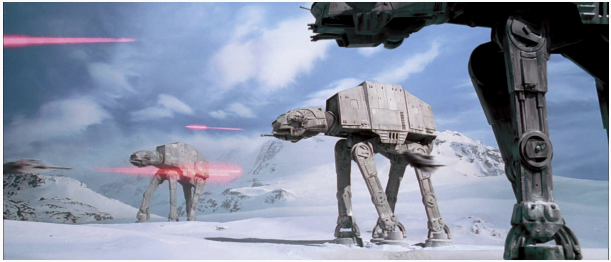
A drawback of the B language was that it did not know data-types (everything was expressed in machine words). Another functionality that the B language did not provide was the use of "structures". The lack of these things formed the reason for Dennis M. Ritchie to develop the **programming language C**. In 1988 they delivered the final standard definition ANSI C

Dennis M. Ritchie and Ken Thompson

```c
#include "stdio.h"

int main() {
    printf("Hello World\n");
}
```

Areas of Application:

- UNIX operating system
- Computer games
- Due to their power and ease of use, C were used in the programming of the special effects for Star Wars



Star Wars - The Empire Strikes Back

The **C++ programming language** (originally named "C with Classes") was devised by **Bjarne Stroustrup** also an employee from Bell Labs (AT&T). Stroustrup started working on C with Classes in 1979. (The ++ is C language operator)

The first commercial release of the C++ language was in October 1985

Domain-specific abstraction

Fortran

Cobol

General-purpose abstraction

Simula

Java

C++ → C++11 → C++14

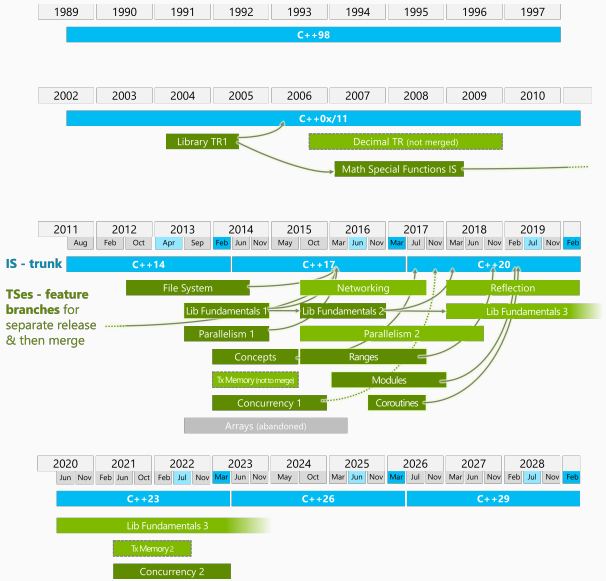
C#

Direct mapping to hardware

Assembler → BCPL → C

The roots of C++

"*The Evolution of C++Past, Present, and Future*", B. Stroustrup, CppCon16

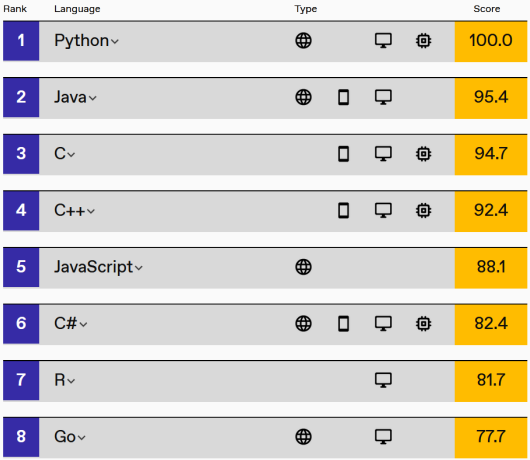

1989 1990 1991 1992 1993 1994 1995 1996 1997
C++98
2002 2003 2004 2005 2006 2007 2008 2009 2010
C++0x/11
Library TR1
Decimal TR (not merged)
Math Special Functions IS
2011 2012 2013 2014 2015 2016 2017 2018 2019
Aug Feb Oct Apr Sep Feb Jun Nov May Oct Mar Jun Nov Mar Jul Nov Mar Jun Nov Feb Jul Nov Feb
IS - trunk
C++14
C++17
C++20
TSes - feature branches for separate release & then merge
File System
Networking
Reflection
Lib Fundamentals 1
Lib Fundamentals 2
Lib Fundamentals 3
Parallelism 1
Parallelism 2
Concepts
Ranges
Tx Memory (not to merge)
Modules
Concurrency 1
Coroutines
Arrays (abandoned)
2020 2021 2022 2023 2024 2025 2026 2027 2028
Jun Nov Feb Jun Oct Feb Jul Nov Mar Jun Nov May Oct Mar Jun Nov Mar Jul Nov Mar Jun Nov Feb Jul Nov Feb
C++23
C++26
C++29
Lib Fundamentals 3
Tx Memory 2
Concurrency 2

"If you're teaching today what you were teaching five years ago, either the field is dead or you are"

Noam Chomsky

Areas of Application and Popularity

Most Popular Programming Languages (IEEE Spectrum - 2022)



Rank	Language	Type	Score
1	Python	web, desktop, server	100.0
2	Java	web, mobile, desktop	95.4
3	C	mobile, desktop, server	94.7
4	C++	mobile, desktop, server	92.4
5	JavaScript	web	88.1
6	C#	web, mobile, desktop, server	82.4
7	R	desktop	81.7
8	Go	web, desktop	77.7

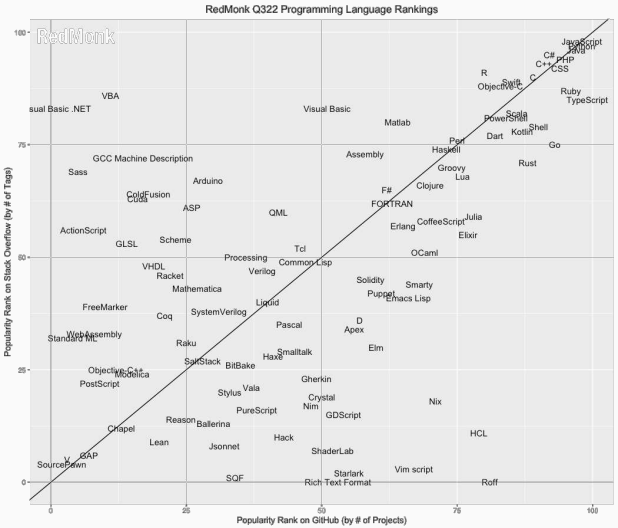
Interactive: The Top Programming Languages 2022

Most Popular Programming Languages (TIOBE - December. 2022)

Programming Language		Ratings	Change
	Python	16.66%	+3.76%
	**C**	**16.56%**	**+4.77%**
	C++	11.94%	+4.21%
	Java	11.82%	+1.70%
	C#	4.92%	-1.48%
	Visual Basic	3.94%	-1.46%
	JavaScript	3.19%	+0.90%

www.tiobe.com/tiobe-index/

Most Popular Programming Languages (Redmonk - June, 2022)



There may be more than 200 billion lines of C/C++ code globally

- **Performance is the defining aspect of C++**. No other programming language provides the performance-critical facilities of C++
- **Provide the programmer control over every aspect of performance**
- **Leave no room for a lower level language**

Total number of lines of all code in use?

- ***Ubiquity.*** C++ can run from a low-power embedded device to large-scale supercomputers
- ***Multi-Paradigm.*** Allow writing efficient code without losing high-level abstraction
- ***Allow writing low-level code.*** Drivers, kernels, assembly (asm), etc.
- ***Ecosystem.*** Many support tools such as debuggers, memory checkers, coverage, static analysis, profiling, etc.
- ***Maturity.*** C++ has a 40 years history. Many software problems have been already addressed and developing practices have been investigated

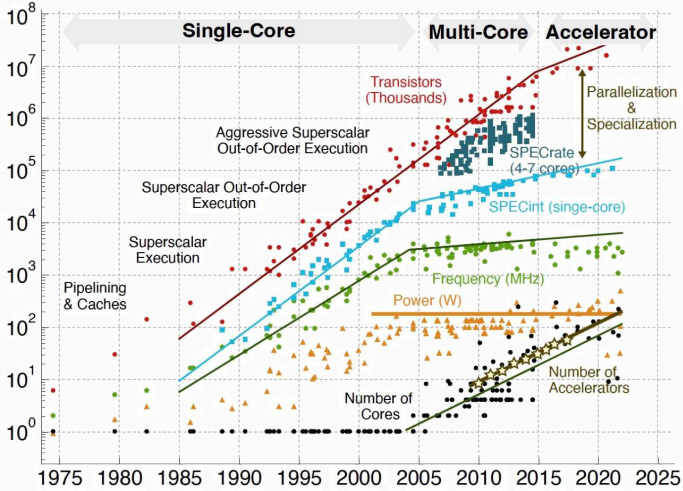
- **Operating systems:** Windows, Android, OS X, Linux
- **Compilers:** LLVM, Swift compiler
- **Artificial Intelligence:** TensorFlow, Caffe, Microsoft Cognitive Toolkit
- **Image Editing:** Adobe Premier, Photoshop, Illustrator
- **Web browser:** Firefox, Chrome, etc. + WebAssembly
- **High-Performance Computing**: drug developing and testing, large scale climate models, physic simulations
- **Embedded systems:** IoT, network devices (e.g. GSM), automotive
- Google and Microsoft use C++ for web indexing

- **Scientific Computing:** CERN/NASA*, SETI@home, Folding@home
- **Database:** MySQL, ScyllaDB
- **Video Games:** Unreal Engine, Unity
- **Entertainment:** Movie rendering (see `Interstellar black hole rendering`), virtual reality
- **Finance:** electronic trading systems (Goldman, JPMorgan, Deutsche Bank)**

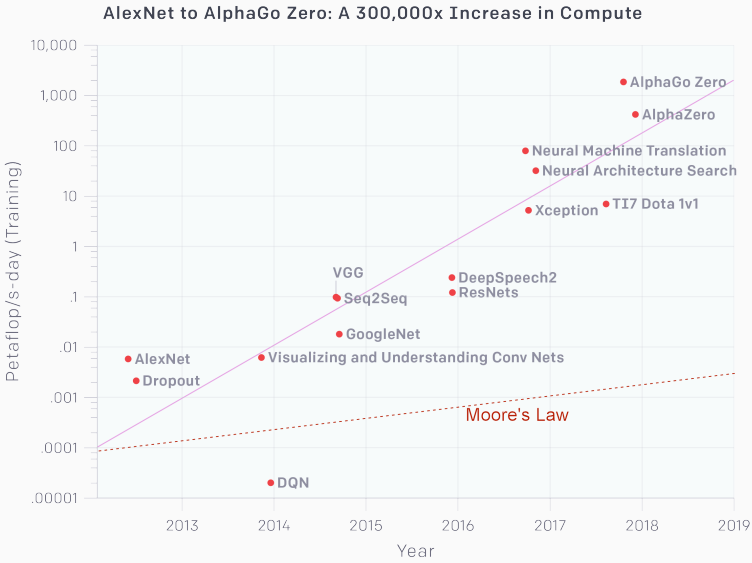
... and many more

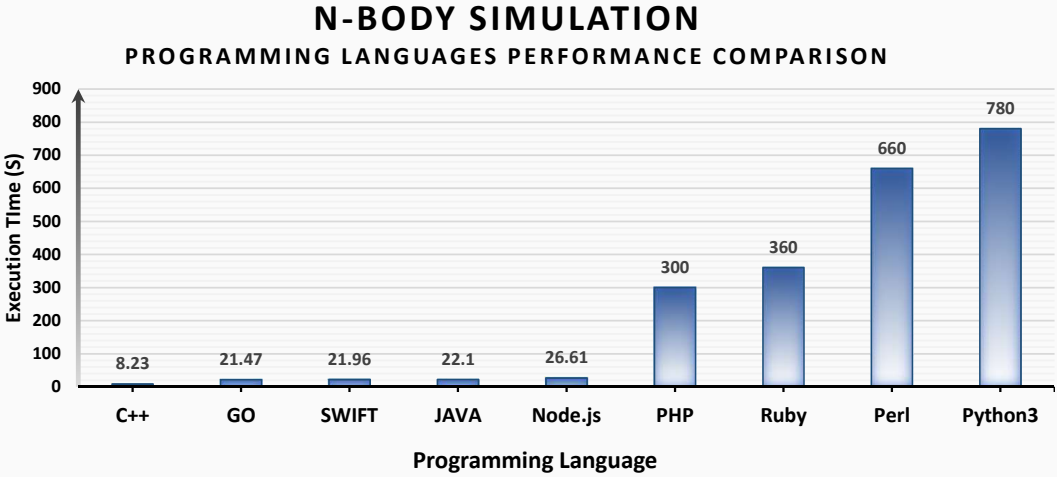
* The flight code of the NASA Mars drone for the **Perseverance** Mission, as well as the **Webb telescope** software, are mostly written in C++ `github.com/nasa/fprime`, `James Webb Space Telescope's Full Deployment`

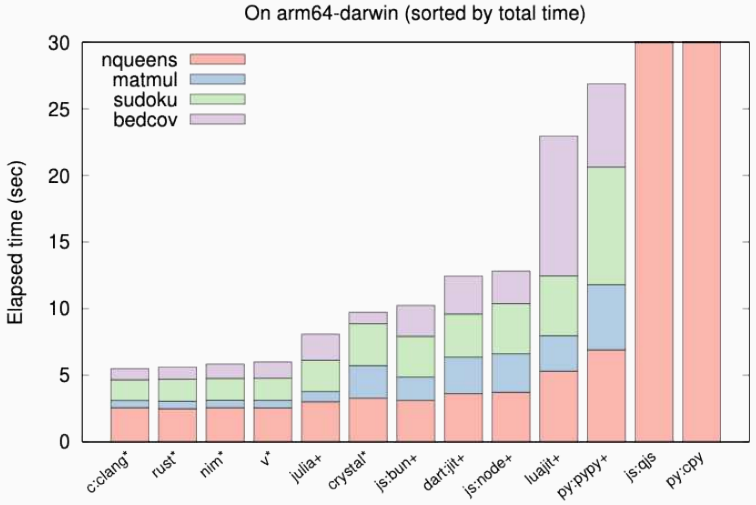
The End of Historical Performance Scaling

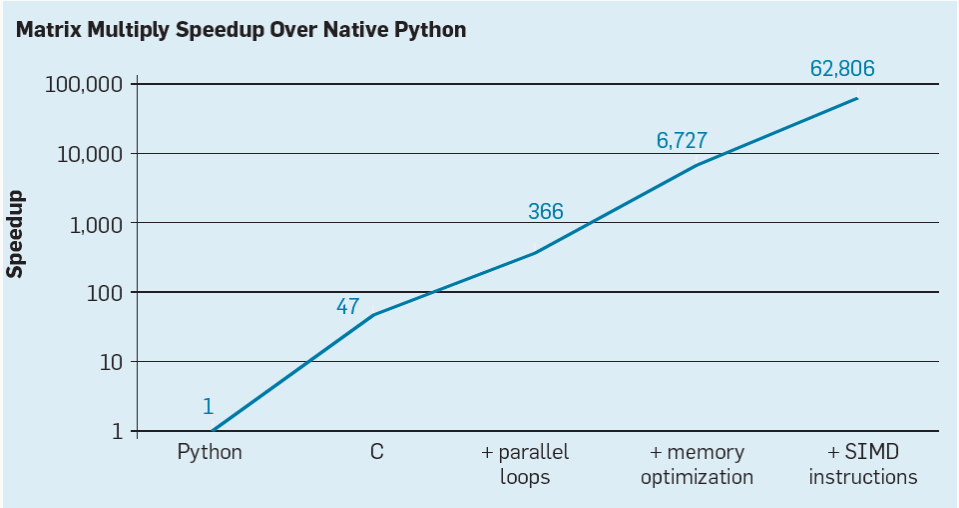


An Important Example... (AI Evolution)




N-BODY SIMULATION
PROGRAMMING LANGUAGES PERFORMANCE COMPARISON
Execution TIme (S)
900
800
700
600
500
400
300
200
100
0
8.23
21.47
21.96
22.1
26.61
300
360
660
780
C++
GO
SWIFT
JAVA
Node.js
PHP
Ruby
Perl
Python3
Programming Language


On arm64-darwin (sorted by total time)
Elapsed time (sec)
0
5
10
15
20
25
30
nqueens
matmul
sudoku
bedcov
c:clang*
rust*
nim*
v*
julia+
crystal*
js:bun+
dart:jit+
js:node+
luajit+
py:pypy+
js:qjs
py:cpy



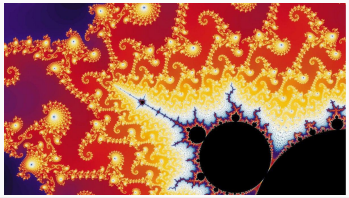
"*A New Golden Age for Computer Architecture*", J. L. Heneessey, D. A. Patterson, 2019

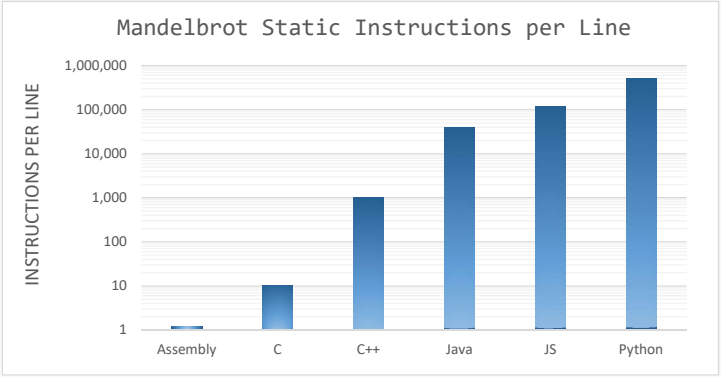
Hello World

Language	Execution Time
C (on my machine)	0.7 ms
C	2 ms
Go	4 ms
Crystal	8 ms
Shell	10 ms
Python	78 ms
Node	110 ms
Ruby	150 ms
jRuby	1.4 s

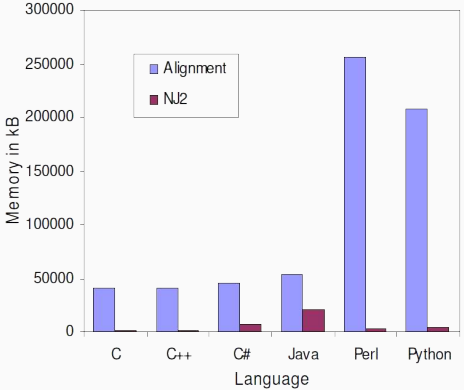
Time to "hello world" on my machine

Performance/Expressiveness Trade-off





Memory Usage



Memory usage comparison of the
Neighbor-Joining and global alignment programs

A comparison of common programming languages used in bioinformatics (BMC Informatic)

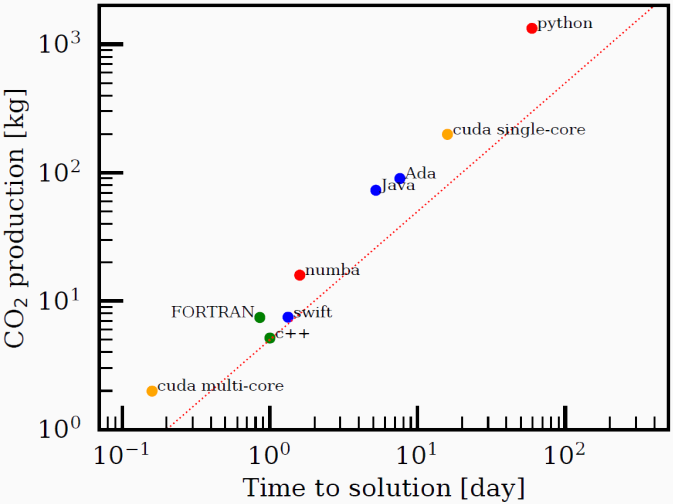
Energy Efficiency

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

Energy Efficiency across Programming Languages

CO^2 Production



C++ Philosophy

*Do not sacrifice **performance** except as a last resort*

Zero Overhead Principle (zero-cost abstraction)

> *"it basically says if you have an abstraction it should not cost anything compared to write the equivalent code at lower level"*
>
> *"so I have say a matrix multiply it should be written in a such a way that you could not drop to the C level of abstraction and use arrays and pointers and such and run faster"*
>
> **Bjarne Stroustrup**

*Enforce **safety at compile time** whenever possible*

Statically Typed Language

> *"The C++ compiler provides type safety and catches many bugs at compile time instead of run time (a critical consideration for many commercial applications.)"*
>
> `www.python.org/doc/FAQ.html`

- The *type annotation* makes the code more readable
- Promote compiler optimizations and runtime efficiency
- Allow users to define their own type system

- **Programming model**: *compartmentalization*, only add features if they solve an actual problem, and allow *full control*
- **Predictable runtime** (under constraints): no garbage collector, no dynamic type system $\rightarrow$ *real-time systems*
- **Low resources**: low memory and energy consumption $\rightarrow$ *restricted hardware platforms*
- **Well suited for static analysis** $\rightarrow$ *safety critical software*
- **Portability** $\rightarrow$ Modern C++ standards are highly portable

Who is C++ for?

"C++ is for people who want to use hardware very well and manage the complexity of doing that through abstraction"

Bjarne Stroustrup

"a language like C++ is not for everybody. It is generated via sharp and effective tool for professional basically and definitely for people who aim at some kind of precision"

Bjarne Stroustrup

Suggested Introduction Video

C++ Weaknesses

... and why teaching C++ as first programming language is a bad idea?

C++ is the hardest language from students to master

- *More languages in one*
  - Standard C/C++ programming
  - Preprocessor
  - Object-Oriented features
  - Templates and Meta-Programming
- *Huge set of features*
- *Worry about memory management*
- *Low-level implementation details*: pointer arithmetic, structure, padding, undefined behavior, etc.
- *Frustrating*: compiler/runtime errors (e.g. seg. fault)

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows your whole leg off"

***Bjarne Stroustrup**, Creator of the C++ language*

"The problem with using C++...is that there's already a strong tendency in the language to require you to know everything before you can do anything"

***Larry Wall**, Creator of the Perl language*

"Despite having 20 years of experience with C++, when I compile a non-trivial chunk of code for the first time without any error or warning, I am suspicious. It is not, usually, a good sign"

***Daniel Lemire**, Prof. at the University of Quebec*

Backward-compatibility

"**Dangerous defaults and constructs**, often originating from C, cannot be removed or altered"

"Despite the hard work of the committee, **newer features sometimes have flaws that only became obvious after extensive user experience**, which cannot then be fixed"

"C++ practice has put an **ever-increasing cognitive burden** on the developer for what I feel has been very little gain in productivity or expressiveness and at a huge cost to code clarity"

C++ critics and replacements:

- `Epochs: a backward-compatible language evolution mechanism`
- `Goals and priorities for C++`
- `Carbon Language`
- `Circle C++ Compiler`
- `Cppfront: Can C++ be 10x simpler & safer ... ?`

C++ Alternatives: Rust

Rust (1.0, 2015) has been Stack Overflow's most loved language for eight years in a row. Rust focuses on performance and zero-abstraction overhead as C++. It is designed to prevent many vulnerabilities that affect C++, especially memory bugs, enforcing constraints at compile type. In addition, it promotes cross-platform compatibility

> *"First-time contributors to Rust projects are about 70 times less likely to introduce vulnerabilities than first-time contributors to C++ projects"*
>
> *Tracey et al.* [1]

[1] Grading on a Curve: How Rust can Facilitate New Contributors while Decreasing Vulnerabilities

- CISA, NSA: The Case for Memory Safe Roadmap
- Octoverse: The Fastest Growing Languages
- Secure by Design: Google's Perspective on Memory Safety

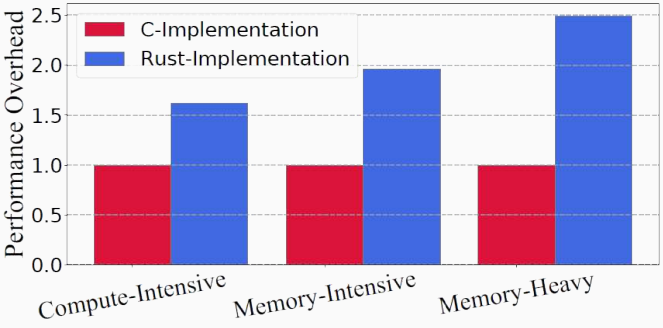
C++ Alternatives: Zig

Zig (2016) is a minimal open-source programming language that can be intended as replacement of C. Zig supports compile time generics, reflection and evaluation, cross-compiling, and manual memory management. It is made to be fully interoperable with C and also includes a C/C++ compiler.

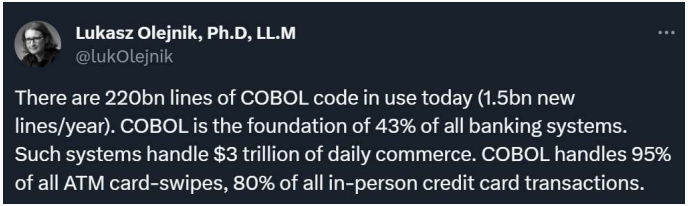
- **No perfect language.** There are always newer *'shining'* languages
- **Alignment.** Force all developers to switch to the new language
- **Interoperability.** Hundreds of billion lines of existing code. Must interoperate with C and C++ code imposing serious design constraints
- **Ecosystem.** Lack of tools and libraries developed in the last four decades
- **Time and Cost.** Converting a codebase of 10 million lines: 500 developers, 5 years, $1,400,000,000[1]

[1] Bjarne Stroustrup: Delivering Safe C++

- **Performance overhead of safe programming languages**



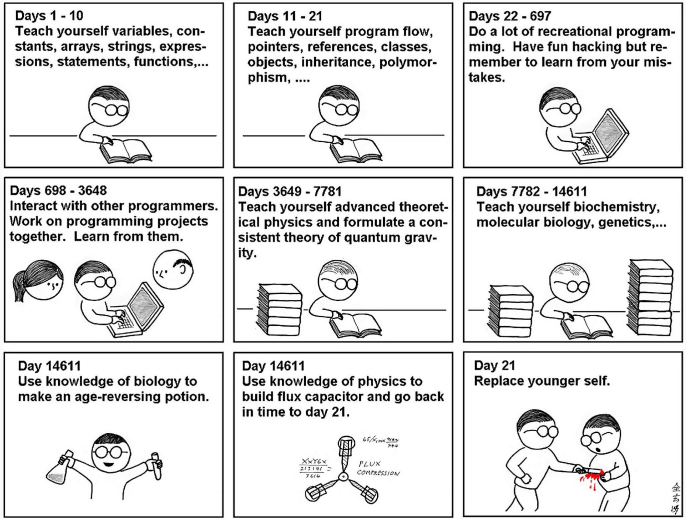
- Towards Understanding the Runtime Performance of Rust
- How much does Rust's bounds checking actually cost?
- How to avoid bounds checks in Rust (without unsafe!)
- Is coding in Rust as bad as in C++?


Lukasz Olejnik, Ph.D, LL.M
@lukOlejnik
There are 220bn lines of COBOL code in use today (1.5bn new lines/year). COBOL is the foundation of 43% of all banking systems. Such systems handle $3 trillion of daily commerce. COBOL handles 95% of all ATM card-swipes, 80% of all in-person credit card transactions.

Every second spent trying to understand the language is one not spent understanding the problem

The Course

The Course


Days 1 - 10
Teach yourself variables, constants, arrays, strings, expressions, statements, functions,...
Days 11 - 21
Teach yourself program flow, pointers, references, classes, objects, inheritance, polymorphism, ...
Days 22 - 697
Do a lot of recreational programming. Have fun hacking but remember to learn from your mistakes.
Days 698 - 3648
Interact with other programmers. Work on programming projects together. Learn from them.
Days 3649 - 7781
Teach yourself advanced theoretical physics and formulate a consistent theory of quantum gravity.
Days 7782 - 14611
Teach yourself biochemistry, molecular biology, genetics,...
Day 14611
Use knowledge of biology to make an age-reversing potion.
Day 14611
Use knowledge of physics to build flux capacitor and go back in time to day 21.
FLUX CONNECTION
Day 21
Replace younger self.
As far as I know, this is the easiest way to
"Teach Yourself C++ in 21 Days".

Don't forget: The right name of the course should be *"Introduction to Modern C++ Programming"*

For many topics in the course, there are more than one book devoted to present the concepts in detail

The Course

The primary goal of the course is to drive who has previous experience with C/C++ and object-oriented programming to a proficiency level of (C++) programming

- *Proficiency*: know what you are doing and the related implications
- Understand what problems/issues address a given language feature
- Learn engineering practices (e.g. code conventions, tools) and hardware/software techniques (e.g. semantic, optimizations) that are not strictly related to C++

What the course **is not**:

- A theoretical course on programming
- A high-level concept description

What the course **is**:

- A practical course, prefer examples to long descriptions
- A "quite" advanced C++ programming language course

The Course

Organization:

- 22 lectures
- ∼1,500 slides
- `C++03 / C++11 / C++14 / C++17 / C++20 / (C++23) / (C++26)`

Roadmap:

- Review C concepts in C++ (built-in types, memory management, preprocessing, etc.)
- Introduce object-oriented and template concepts
- Present how to organize the code and the main conventions
- C++ tool goals and usage (debugger, static analysis, etc.)

Who I Am

***Federico Busato*, Ph.D.**

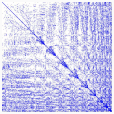
`federico-busato.github.io`

- Senior Software Engineer at Nvidia,
  CUDA Mathematical Libraries
- Lead engineer of the Sparse Linear Algebra group
- `Research/Work interests:`
  - Linear Algebra
  - Graph Algorithms
  - Parallel/High-Performance Computing
  - Code Optimization

Follow @fedebusato

NOT a C++ expert/"guru", still learning

A Little Bit about My Work

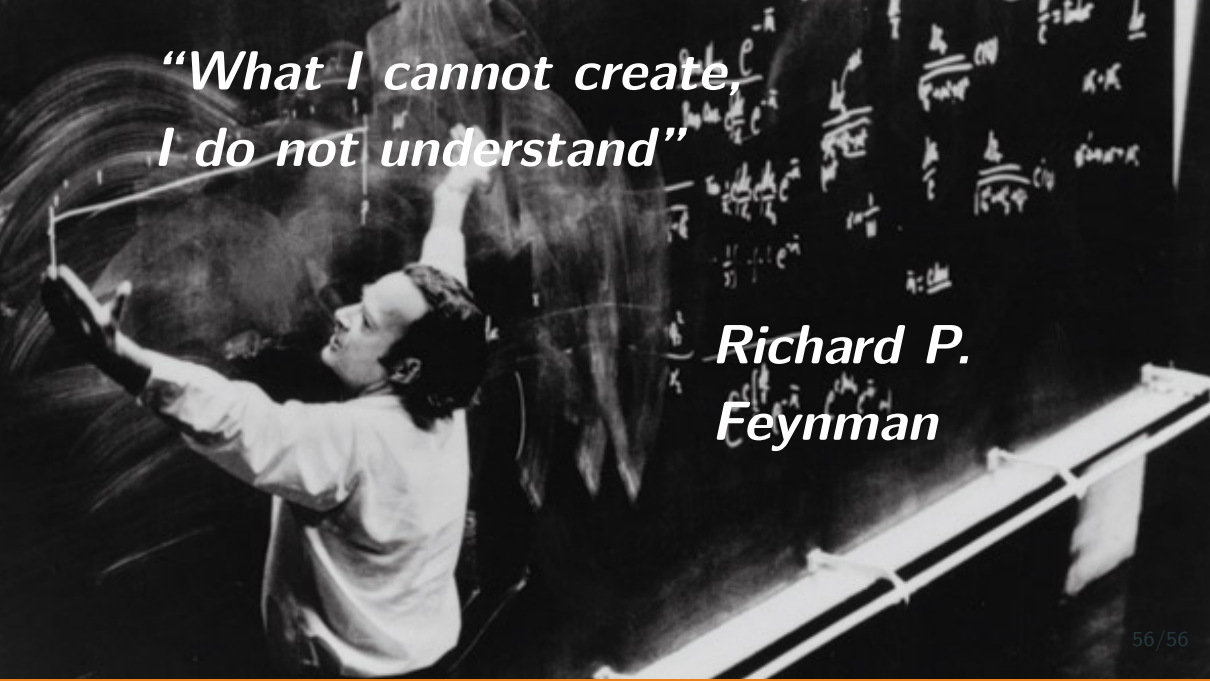
Our projects: 

`cuSPARSE` GPU-accelerated sparse linear algebra library (matrix-matrix multiplication, triangular solver, etc.), part of the `CUDA Toolkit` (8M downloads every year)

`cuSPARSELt` Specialized library for sparse matrix-matrix multiplication that exploits the most advanced GPU features such as Sparse Tensor Cores

`NVPL Sparse` CPU-accelerated (ARM) sparse linear algebra library

`Top500 HPCG` NVIDIA Supercomputing benchmark that performs a fixed number of multigrid preconditioned (using a symmetric Gauss-Seidel smoother) conjugate gradient (PCG) iterations


“What I cannot create,
I do not understand”
Richard P.
Feynman

“The only way to learn a new programming language is by writing programs in it”

Dennis Ritchie

Creator of the C programming language

Modern C++ Programming

1a. Preparation

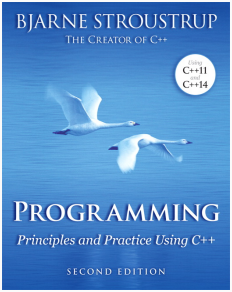
Federico Busato

2024-03-29

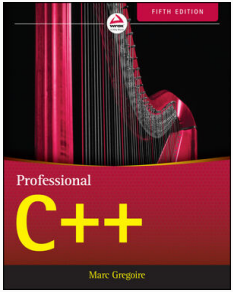
Table of Contents

Books and References

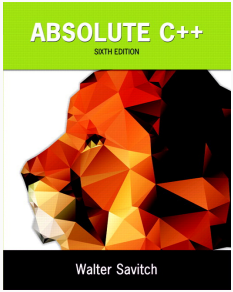
Suggested Books



Programming and Principles using C++ (2nd)
B. Stroustrup, 2014

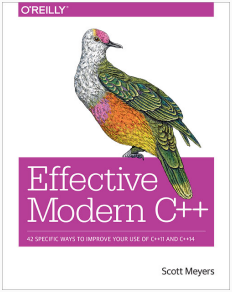


Professional C++ (5th)
S. J. Kleper, N. A. Solter, 2021

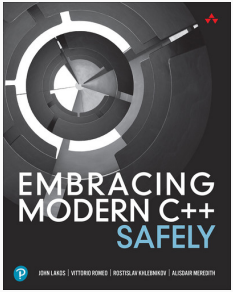


Absolute C++ (6th)
W. Savitch, 2015

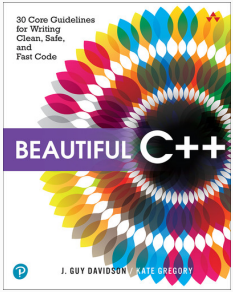
More Advanced Books



Effective Modern C++
S. Meyer, 2014



Embracing Modern C++ Safely
J. Lakos, V. Romeo, R. Khlebnikov, A. Meredith, 2021



Beautiful C++: 30 Core Guidelines for Writing Clean, Safe, and Fast Code
J. G. Davidson, K. Gregory, 2021

(Un)official C++ reference:*

- `en.cppreference.com`
- `C++ Standard Draft`

Tutorials:

- `www.learncpp.com`
- `www.tutorialspoint.com/cplusplus`
- `en.wikibooks.org/wiki/C++`
- `yet another insignificant...programming notes`

Other resources:

- `stackoverflow.com/questions/tagged/c++`

* The full C++ standard draft can be found at `eel.is/c++draft/full` (32 MB!)

News:

- isocpp.org (Standard C++ Foundation)
- cpp.libhunt.com/newsletter/archive
- www.meetingcpp.com/blog/blogroll/

Main conferences:

- www.meetingcpp.com (slides)
- cppcon.org (slides)
- isocpp.com conference list

Coding exercises and other resources:

- www.hackerrank.com/domains/cpp
- leetcode.com/problemset/algorithms
- open.kattis.com
- cpppatterns.com

Slide Legend

★ **Advanced Concepts**. *In general, they are not fundamental*. They can be related to very specific aspects of the language or provide a deeper exploration of C++ features.
A beginner reader should skip these sections/slides

⇝ **See next**. C++ concepts are closely linked, and it is almost impossible to find a way to explain them without referring to future topics. These slides should be revisited after reading the suggested topic

🏠 **Homework**. The slide contains questions/exercises for the reader

```
this is a code section
```

This is a language `keyword/token` and not a program symbol (variable, functions, etc.). Future references to the token could use a standard code section for better readability

What Editor/ IDE/Compiler Should I Use?

What Compiler Should I Use?

Most popular compilers:

- Microsoft Visual Code (**MSVC**) is the compiler offered by Microsoft
- The GNU Compiler Collection (**GCC**) contains the most popular C++ Linux compiler
- **Clang** is a C++ compiler based on LLVM Infrastructure available for Linux/Windows/Apple (default) platforms

Suggested compiler on Linux for beginner: **Clang**

- Comparable performance with GCC/MSVC and low memory usage
- Expressive diagnostics (examples and propose corrections)
- Strict C++ compliance. GCC/MSVC compatibility (inverse direction is not ensured)
- Includes very useful tools: memory sanitizer, static code analyzer, automatic formatting, linter, etc.

Install the Compiler on Linux

Install the last gcc/g++ (v11) (v12 on Ubuntu 22.04)

```
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
$ sudo apt update
$ sudo apt install gcc-12 g++-12
$ gcc-12 --version
```

Install the last clang/clang++ (v17)

```
$ bash -c "$(wget -O - https://apt.llvm.org/llvm.sh)"
$ wget https://apt.llvm.org/llvm.sh
$ chmod +x llvm.sh
$ sudo ./llvm.sh 17
$ clang++ --version
```

Install the Compiler on Windows

Microsoft Visual Studio

- Direct Installer: `Visual Studio Community 2022`

Clang on Windows

Two ways:

- Windows Subsystem for Linux (WSL)
  - `Run` → `optionalfeatures`
  - Select `Windows Subsystem for Linux`, `Hyper-V`, `Virtual Machine Platform`
  - `Run` → `ms-windows-store:` → Search and install `Ubuntu 22.04 LTS`
- Clang + MSVC Build Tools
  - Download Build Tools per Visual Studio
  - Install `Desktop development with C++`

Popular C++ IDE (Integrated Development Environment):

- **Microsoft Visual Studio** (MSVC) (link). Most popular IDE for Windows
- **Clion** (link). (free for student). Powerful IDE with a lot of options
- **QT-Creator** (link). Fast (written in C++), simple
- **XCode**. Default on Mac OS
- **Cevelop** (Eclipse) (link)

Standalone GUI-based coding editors:

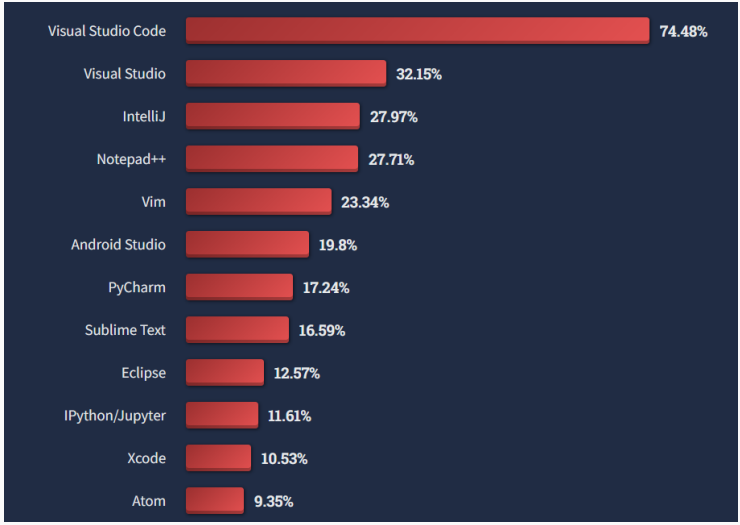
- **Microsoft Visual Studio Code** (VSCode) (link)
- **Sublime** (link)
- **Lapce** (link)
- **Zed** (link)

Standalone text-based coding editors (powerful, but needs expertise):

- **Vim**
- **Emacs**
- **NeoVim** (link)
- **Helix** (link)

Not suggested: `Notepad, Gedit, and other similar editors` (lack of support for programming)

What Editor/IDE/Compiler Should I Use? 3/3



StackOverflow Developer Survey 2022

How to compile?

How to Compile?

Compile C++11, C++14, C++17, C++20, C++23, C++26 programs:

```
g++  -std=c++11 <program.cpp> -o program
g++  -std=c++14 <program.cpp> -o program
g++  -std=c++17 <program.cpp> -o program
g++  -std=c++20 <program.cpp> -o program
g++  -std=c++23 <program.cpp> -o program
g++  -std=c++26 <program.cpp> -o program
```

Any C++ standard is backward compatible*

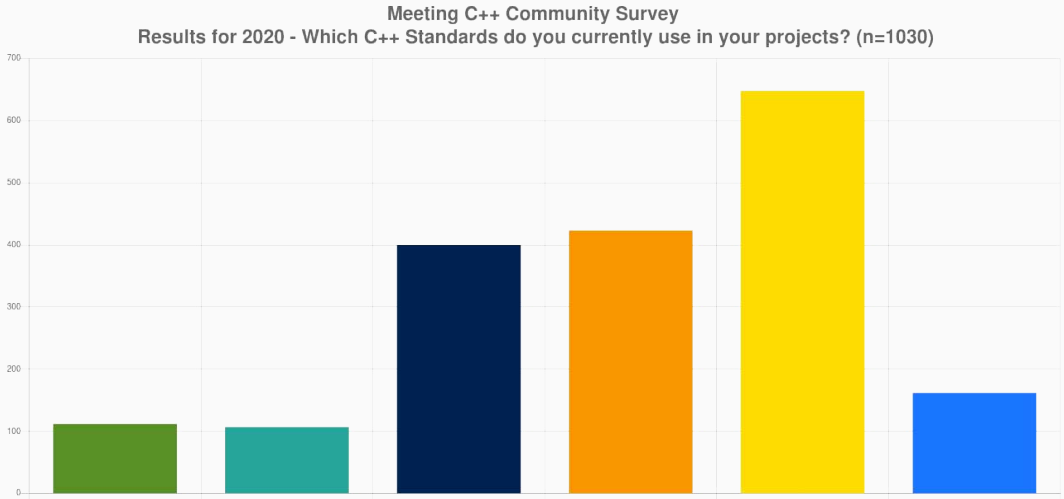
C++ is also backward compatible with C in most case, except if it contains C++ keywords (`new`, `template`, `class`, `typename`, `etc.`)
We can potentially compile a pure C program in C++26

*except for very minor deprecated features

Compiler	C++11		C++14		C++17		C++20	
	Core	Library	Core	Library	Core	Library	Core	Library
g++	4.8.1	5.1	5.1	5.1	7.1	9.0	11+	11+
clang++	3.3	3.3	3.4	3.5	5.0	11.0	16+	16+
MSVC	19.0	19.0	19.10	19.0	19.15	19.15	19.29+	19.29

en.cppreference.com/w/cpp/compiler_support

C++ Standard


Meeting C++ Community Survey
Results for 2020 - Which C++ Standards do you currently use in your projects? (n=1030)
700
600
500
400
300
200
100
0
c++98
c++03
c++11
c++14
c++17
c++20

Hello World

C code with `printf`:

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
}
```

`printf`
prints on standard output

C++ code with `streams`:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World!\n";
}
```

`cout`
represents the standard output stream

The previous example can be written with the global `std` namespace:

```cpp
#include <iostream>

using namespace std;
int main() {
    cout << "Hello World!\n";
}
```

Note: For sake of space and for improving the readability, we intentionally omit the `std` namespace in most slides

`std::cout` is an example of *output* stream. Data is redirected to a destination, in this case the destination is the standard output

C:

```c
#include <stdio.h>
int main() {
    int    a   = 4;
    double b   = 3.0;
    char   c[] = "hello";
    printf("%d %f %s\n", a, b, c);
}
```

C++:

```cpp
#include <iostream>
int main() {
    int    a   = 4;
    double b   = 3.0;
    char   c[] = "hello";
    std::cout << a << " " << b << " " << c << "\n";
}
```

- **Type-safe**: The type of object provided to the I/O stream is known statically by the compiler. In contrast, `printf` uses `%` fields to figure out the types dynamically
- **Less error prone**: With I/O Stream, there are no redundant `%` tokens that have to be consistent with the actual objects passed to I/O stream. Removing redundancy removes a class of errors
- **Extensible**: The C++ I/O Stream mechanism allows new user-defined types to be passed to I/O stream without breaking existing code
- **Comparable performance**: If used correctly may be faster than C I/O (`printf` , `scanf` , etc.) .

- Forget the number of parameters:

```
printf("long phrase %d long phrase %d", 3);
```

- Use the wrong format:

```
int a = 3;
...many lines of code...
printf(" %f", a);
```

- The `%c` conversion specifier does not automatically skip any leading white space:

```
scanf("%d", &var1);
scanf(" %c", &var2);
```

std::print

C++23 introduces an improved version of `printf` function `std::print` based on *formatter strings* that provides all benefits of C++ stream and is less verbose

```cpp
#include <print>

int main() {
    std::print("Hello World! {}, {}, {}\n", 3, 4ll, "aa");
    // print "Hello World! 3 4 aa"
}
```

This will be the default way to print when the C++23 standard is widely adopted

Modern C++ Programming

2. Basic Concepts I

Type System, Fundamental Types, and Operators

Federico Busato

2024-03-29

Table of Contents

Table of Contents

The C++ Type System

The C++ Type System

C++ is a **strongly typed** and **statically typed** language

Every entity has a type and that type never changes

Every variable, function, or expression has a **type** in order to be compiled. Users can introduce new types with `class` or `struct`

The **type** specifies:

- The *amount of memory* allocated for the variable (or expression result)
- The *kinds of values* that may be stored and how the compiler interprets the bit patterns in those values
- The *operations* that are permitted for those entities and provides semantics

Type Categories

C++ organizes the language types in two main categories:

- **Fundamental types**: Types provided by the language itself
  - Arithmetic types: integer and floating point
  - `void`
  - `nullptr` C++11
- **Compound types**: Composition or references to other types
  - Pointers
  - References
  - Enumerators
  - Arrays
  - `struct`, `class`, `union`
  - Functions

C++ types can be also classified based on their properties:

- **Objects**:
  - *size*: `sizeof` is defined
  - *alignment requirement*: `alignof` is defined
  - *storage duration*: describe when an object is allocated and deallocated
  - *lifetime*, bounded by storage duration or temporary
  - *value*, potentially indeterminate
  - optionally, a *name*.

  Types: Arithmetic, Pointers and `nullptr`, Enumerators, Arrays, `struct`, `class`, `union`

- **Scalar**:
  - *Hold a single value* and is not composed of other objects
  - *Trivially Copyable*: can be copied bit for bit
  - *Standard Layout*: compatible with C functions and structs
  - *Implicit Lifetime*: no user-provided constructor or destructor

  Types: Arithmetic, Pointers and `nullptr`, Enumerators

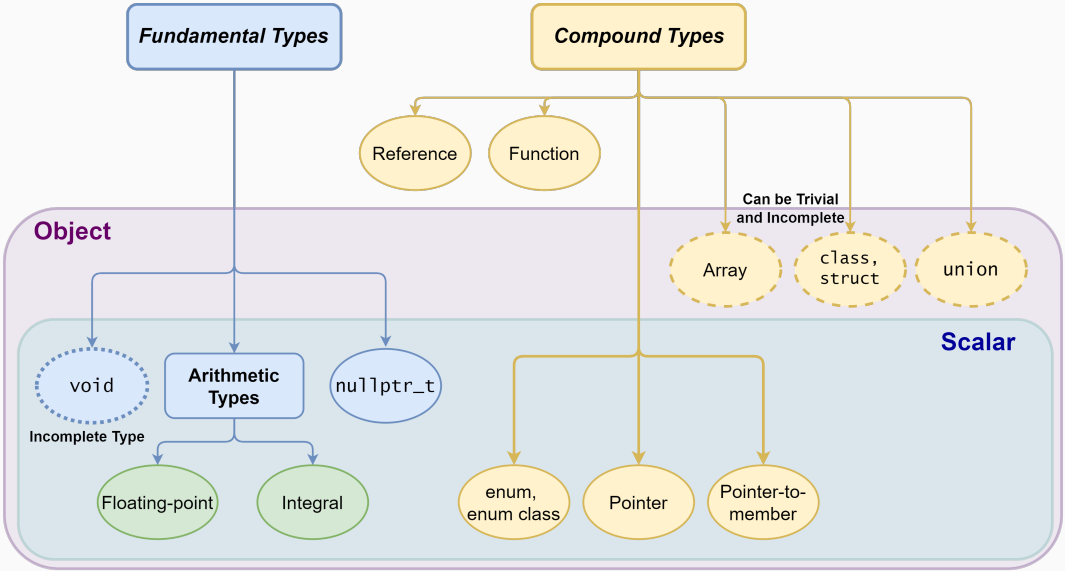
- **Trivial types**: Trivial default/copy constructor, copy assignment operator, and destructor → *Trivially Copyable*

  Types: Scalar, trivial class types, arrays of such types

- **Incomplete types**: A type that has been declared but not yet defined

  Types: `void`, incompletely-defined object types, e.g. `struct A;`, array of elements of incomplete type

C++ Types Summary



Fundamental Types Overview

Arithmetic Types - Integral

Native Type	Bytes	Range	Fixed width types <cstdint>
`bool`	1	true, false	
`char` †	1	implementation defined	
`signed char`	1	-128 to 127	int8_t
`unsigned char`	1	0 to 255	uint8_t
`short`	2	-2^{15} to $2^{15}-1$	int16_t
`unsigned short`	2	0 to $2^{16}-1$	uint16_t
`int`	4	-2^{31} to $2^{31}-1$	int32_t
`unsigned int`	4	0 to $2^{32}-1$	uint32_t
`long int`	4/8		int32_t/int64_t
`long unsigned int`	4/8*		uint32_t/uint64_t
`long long int`	8	-2^{63} to $2^{63}-1$	int64_t
`long long unsigned int`	8	0 to $2^{64}-1$	uint64_t

* 4 bytes on Windows64 systems, † `signed/unsigned`, two-complement from C++11

Arithmetic Types - Floating-Point

Native Type	IEEE	Bytes	Range	Fixed width types C++23 `<stdfloat>`
(bfloat16)	N	2	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$	std::bfloat16_t
(float16)	Y	2	0.00006 to 65,536	std::float16_t
`float`	Y	4	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$	std::float32_t
`double`	Y	8	$\pm 2.23 \times 10^{-308}$ to $\pm 1.8 \times 10^{+308}$	std::float64_t

Arithmetic Types - Short Name

Signed Type	short name
`signed char`	/
`signed short int`	`short`
`signed int`	`int`
`signed long int`	`long`
`signed long long int`	`long long`

Unsigned Type	short name
`unsigned char`	/
`unsigned short int`	`unsigned short`
`unsigned int`	`unsigned`
`unsigned long int`	`unsigned long`
`unsigned long long int`	`unsigned long long`

Arithmetic Types - Suffix (Literals)

Type	SUFFIX	Example	Notes
`int`	/	2	
`unsigned int`	u, U	3u	
`long int`	l, L	8L	
`long unsigned`	ul, UL	2ul	
`long long int`	ll, LL	4ll	
`long long unsigned int`	ull, ULL	7ULL	
`float`	f, F	3.0f	only decimal numbers
`double`		3.0	only decimal numbers

C++23 Type	SUFFIX	Example	Notes
`std::bfloat16_t`	bf16, BF16	3.0bf16	only decimal numbers
`std::float16_t`	f16, F16	3.0f16	only decimal numbers
`std::float32_t`	f32, F32	3.0f32	only decimal numbers
`std::float64_t`	f64, F64	3.0f64	only decimal numbers
`std::float128_t`	f128, F128	3.0f128	only decimal numbers

Arithmetic Types - Prefix (Literals)

Representation	PREFIX	Example
`Binary` C++14	`0b`	`0b010101`
`Octal`	`0`	`0307`
`Hexadecimal`	`0x` or `0X`	`0xFFA010`

C++14 also allows *digit separators* for improving the readability `1'000'000`

Other Arithmetic Types

- C++ also provides `long double` (no IEEE-754) of size 8/12/16 bytes depending on the implementation

- Reduced precision floating-point supports before C++23:
  - Some compilers provide support for *half* (16-bit floating-point) (GCC for ARM: `__fp16`, LLVM compiler: `half`)
  - Some modern CPUs and GPUs provide *half* instructions
  - Software support: OpenGL, Photoshop, Lightroom, `half.sourceforge.net`

- C++ does not provide **128-bit integers** even if some architectures support it. `clang` and `gcc` allow 128-bit integers as compiler extension (`__int128`)

`void` Type

`void` is an incomplete type (not defined) without a value

- `void` indicates also a function with no return type or no parameters
  e.g. `void f()` , `f(void)`

- In C `sizeof(void) == 1` (GCC), while in C++ `sizeof(void)` does not compile!!

```c
int main() {
//  sizeof(void); // compile error
}
```

`nullptr` Keyword

C++11 introduces the keyword `nullptr` to represent a `null` pointer (`0x0`) and replacing the `NULL` macro

`nullptr` is an object of type `nullptr_t` → safer

```
int* p1 = NULL;      // ok, equal to int* p1 = 0l
int* p2 = nullptr;   // ok, nullptr is convertible to a pointer

int   n1 = NULL;     // ok, we are assigning 0 to n1
//int n2 = nullptr; // compile error nullptr is not convertible to an integer

//int* p2 = true ? 0 : nullptr; // compile error incompatible types
```

Conversion Rules

Conversion Rules

Implicit type conversion rules, applied in order, before any operation:

$\otimes$: any operation (*, +, /, -, %, etc.)

(A) Floating point promotion

`floating_type` $\otimes$ `integer_type` $\rightarrow$ `floating_type`

(B) Implicit integer promotion

`small_integral_type` := any signed/unsigned integral type smaller than `int`

`small_integral_type` $\otimes$ `small_integral_type` $\rightarrow$ `int`

(C) Size promotion

`small_type` $\otimes$ `large_type` $\rightarrow$ `large_type`

(D) Sign promotion

`signed_type` $\otimes$ `unsigned_type` $\rightarrow$ `unsigned_type`

Examples and Common Errors

```
float    f = 1.0f;
unsigned u = 2;
int      i = 3;
short    s = 4;
uint8_t  c = 5; // unsigned char

f * u; // float × unsigned → float: 2.0f
s * c; // short × unsigned char → int: 20
u * i; // unsigned × int → unsigned: 6u
+c;    // unsigned char → int: 5
```

Integers are not floating points!

```
int   b = 7;
float a = b / 2;   // a = 3 not 3.5!!
int   c = b / 2.0; // again c = 3 not 3.5!!
```

Implicit Promotion

Integral data types smaller than 32-bit are *implicitly* promoted to `int`, independently if they are *signed* or *unsigned*

- Unary `+, -, ~` and Binary `+, -, &, etc.` promotion:

```
char a = 48;      // '0'
cout << a;        // print '0'
cout << +a;       // print '48'
cout << (a + 0); // print '48'

uint8_t a1 = 255;
uint8_t b1 = 255;
cout << (a1 + b1);  // print '510' (no overflow)
```

`auto` Keyword

C++11 The auto keyword specifies that the type of the variable will be automatically deduced by the compiler (from its initializer)

```
auto a = 1 + 2;   // 1 is int, 2 is int, 1 + 2 is int!
//    -> 'a' is "int"
auto b = 1 + 2.0; // 1 is int, 2.0 is double. 1 + 2.0 is double
//    -> 'b' is "double"
```

`auto` can be very useful for maintainability and for hiding complex type definitions

```
for (auto i = k; i < size; i++)
    ...
```

On the other hand, it may make the code less readable if excessively used because of type hiding

Example: `auto x = 0;` in general makes no sense (`x` is `int`)

In C++11/C++14, `auto` (as well as `decltype`) can be used to define function output types

```
auto g(int x) -> int { return x * 2; } // C++11
// "-> int" is the deduction type
// a better way to express it is:

auto g2(int x) -> decltype(x * 2) { return x * 2; }  // C++11

auto h(int x) { return x * 2; }         // C++14

//-------------------------------------------------------------

int x = g(3); // C++11
```

In C++20, `auto` can be also used to define function input

```
void f(auto x) {}
// equivalent to templates but less expensive at compile-time

//-------------------------------------------------------------

f(3);    // 'x' is int
f(3.0);  // 'x' is double
```

C++ Operators

Operators Overview

Precedence	Operator	Description	Associativity
1	`a++ a--`	Suffix/postfix increment and decrement	Left-to-right
2	`+a -a ++a --a` `! not ~`	Plus/minus, Prefix increment/decrement, Logical/Bitwise Not	Right-to-left
3	`a*b a/b a%b`	Multiplication, division, and remainder	Left-to-right
4	`a+b a-b`	Addition and subtraction	Left-to-right
5	$\ll$ $\gg$	Bitwise left shift and right shift	Left-to-right
6	`< <= > >=`	Relational operators	Left-to-right
7	`== !=`	Equality operators	Left-to-right
8	`&`	Bitwise AND	Left-to-right
9	`^`	Bitwise XOR	Left-to-right
10	`|`	Bitwise OR	Left-to-right
11	`&& and`	Logical AND	Left-to-right
12	`|| or`	Logical OR	Left-to-right
13	`+= -= *= /= %=` `<<= >>= &= ^= |=`	Compound	Right-to-left

- **Unary** operators have higher precedence than **binary operators**
- **Standard math operators** (`+`, `*`, `etc.`) have higher precedence than **comparison**, **bitwise**, and **logic** operators
- **Bitwise** and **logic** operators have higher precedence than **comparison** operators
- **Bitwise** operators have higher precedence than **logic** operators
- **Compound assignment** operators `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `!=`, `&=`, `>>=`, `<<=` have lower priority
- The **comma** operator has the lowest precedence (see next slides)

en.cppreference.com/w/cpp/language/operator_precedence

Examples:

```
a + b * 4;              // a + (b * 4)

a * b / c % d;          // ((a * b) / c) % d

a + b < 3 >> 4;         // (a + b) < (3 >> 4)

a && b && c || d;       // (a && b && c) || d

a and b and c or d;     // (a && b && c) || d

a | b & c || e && d;    // ((a | (b & c)) || (e && d)
```

Important: sometimes parenthesis can make an expression verbose... but they can help!

Prefix/Postfix Increment Semantic

Prefix Increment/Decrement `++i`, `--i`

(1) Update the value
(2) Return the new (updated) value

Postfix Increment/Decrement `i++`, `i--`

(1) Save the old value (temporary)
(2) Update the value
(3) Return the old (original) value

Prefix/Postfix increment/decrement semantic applies not only to built-in types but also to objects

Operation Ordering Undefined Behavior ★

Expressions with undefined (implementation-defined) behavior:

```
int i = 0;
i = ++i + 2;       // until C++11: undefined behavior
                   // since C++11:  i = 3
i = 0;
i = i++ + 2;       // until C++17: undefined behavior
                   // since C++17: i = 3

f(i = 2, i = 1);   // until C++17: undefined behavior
                   // since C++17:  i = 2
i = 0;
a[i] = ++i;        // until C++17: undefined behavior
                   // since C++17: a[1] = 1

f(++i, ++i);       // undefined behavior
i = ++i + i++;     // undefined behavior
```

Assignment, Compound, and Comma Operators

Assignment and **compound assignment** operators have *right-to-left associativity* and their expressions return the assigned value

```
int y = 2;
int x = y = 3; // y=3, then x=3
               // the same of x = (y = 3)
if (x = 4)     // assign x=4 and evaluate to true
```

The **comma operator**★ has *left-to-right associativity*. It evaluates the left expression, discards its result, and returns the right expression

```
int a = 5, b = 7;
int x = (3, 4); // discards 3, then x=4
int y = 0;
int z;
z = y, x;       // z=y (0), then returns x (4)
```

Spaceship Operator <=> ★

C++20 provides the **three-way comparison operator** `<=>`, also called *spaceship operator*, which allows comparing two objects similarly of `strcmp`. The operator returns an object that can be directly compared with a positive, 0, or negative integer value

```
(3 <=> 5)     == 0; // false
('a' <=> 'a') == 0; // true

(3 <=> 5)     < 0;  // true
(7 <=> 5)     < 0;  // false
```

The semantic of the *spaceship operator* can be extended to any object (see next lectures) and can greatly simplify the comparison operators overloading

Safe Comparison Operators ★

C++20 introduces a set of functions `<utility>` to safely compare integers of different types (`signed`, `unsigned`)

```
bool cmp_equal(T1 a, T2 b)
bool cmp_not_equal(T1 a, T2 b)
bool cmp_less(T1 a, T2 b)
bool cmp_greater(T1 a, T2 b)
bool cmp_less_equal(T1 a, T2 b)
bool cmp_greater_equal(T1 a, T2 b)
```

example:

```
#include <utility>
unsigned a  = 4;
int      b  = -3;
bool     v1 = (a > b);           // false!!!, see next slides
bool     v2 = std::cmp_greater(a, b); // true
```

Modern C++ Programming

3. Basic Concepts II

Integral and Floating-point Types

Federico Busato

2024-03-29

Table of Contents

2 Floating-point Types and Arithmetic

Table of Contents

Integral Data Types

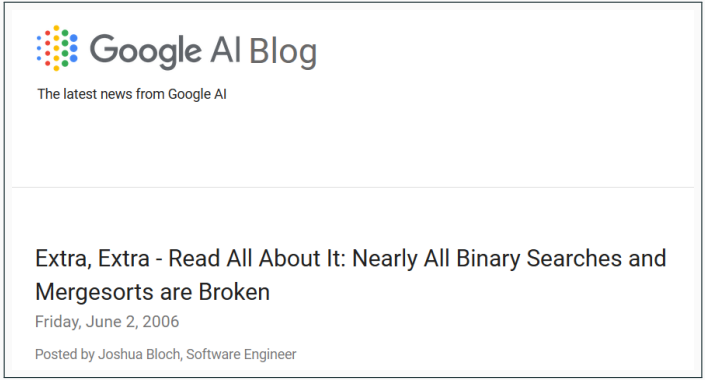
A Firmware Bug

> *"Certain SSDs have a firmware bug causing them to irrecoverably fail after exactly 32,768 hours of operation. SSDs that were put into service at the same time will fail simultaneously, so RAID won't help"*

`HPE SAS Solid State Drives - Critical Firmware Upgrade`

Via `twitter.com/martinkl/status/1202235877520482306?s=19`

Overflow Implementations



Note: Computing the average in the right way is not trivial, see `On finding the average of two unsigned integers without overflow`

related operations: ceiling division, rounding division

`ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html`

Potentially Catastrophic Failure

$$51\ \textit{days} = 51 \cdot 24 \cdot 60 \cdot 60 \cdot 1000 = 4\,406\,400\,000\ \textit{ms}$$

Boeing 787s must be turned off and on every 51 days to prevent ‘misleading data’ being shown to pilots

C++ Data Model

Model/Bits	OS	`short`	`int`	`long`	`long long`	`pointer`
ILP32	Windows/Unix 32-b	16	32	32	64	32
LLP64	Windows 64-bit	16	32	32	64	64
LP64	Linux 64-bit	16	32	64	64	64

`char` is always 1 byte

LP32 Windows 16-bit APIs (no more used)

`int*_t <cstdint>`

C++ provides fixed width integer types.
They have the same size on any architecture:

int8_t, uint8_t
int16_t, uint16_t
int32_t, uint32_t
int64_t, uint64_t

Good practice: Prefer fixed-width integers instead of native types. `int` and `unsigned` can be directly used as they are widely accepted by C++ data models

`int*_t` types are not "real" types, they are merely *typedefs* to appropriate fundamental types

C++ standard does not ensure a one-to-one mapping:

- There are **five** distinct *fundamental types* (`char` , `short` , `int` , `long` , `long long`)
- There are **four** `int*_t` *overloads* (`int8_t` , `int16_t` , `int32_t` , and `int64_t`)

ithare.com/c-on-using-int_t-as-overload-and-template-parameters

Warning: I/O Stream interprets `uint8_t` and `int8_t` as `char` and not as integer values

```
int8_t var;
cin >> var;  // read '2'
cout << var; // print '2'
int a = var * 2;
cout << a;   // print '100' !!
```

`size_t` and `ptrdiff_t`

`size_t` `ptrdiff_t` `<cstddef>`

`size_t` and `ptrdiff_t` are *aliases* data types capable of storing the biggest representable value on the current architecture

- `size_t` is an unsigned integer type (of at least 16-bit)
- `ptrdiff_t` is the signed version of `size_t` commonly used for computing pointer differences
- `size_t` is the return type of `sizeof()` and commonly used to represent size measures
- `size_t` / `ptrdiff_t` are 4 bytes on 32-bit architectures, and 8 bytes on 64-bit architectures
- C++23 adds `uz` / `UZ` literals for `size_t`, and `z` / `Z` for `ptrdiff_t`

Signed/Unsigned Integer Characteristics

`Signed` and `Unsigned` integers use the same hardware for their operations, but they have very different semantic

Basic concepts:

Overflow The result of an arithmetic operation exceeds the word length, namely the positive/negative the largest values

Wraparound The result of an arithmetic operation is reduced modulo 2^N where N is the number of bits of the word

Signed Integer

- Represent positive, negative, and zero values ($\mathbb{Z}$)

✅ Represent the human intuition of numbers

⚠ More negative values ($2^{31} - 1$) than positive ($2^{31} - 2$)

Even multiply, division, and modulo by -1 can fail

⚠ *Overflow/underflow semantic* $\rightarrow$ undefined behavior

Possible behavior: overflow: $(2^{31} - 1) + 1 \rightarrow \mathit{min}$

underflow: $-2^{31} - 1 \rightarrow \mathit{max}$

⚠ Bit-wise operations are implementation-defined

e.g. signed shift $\rightarrow$ undefined behavior

- *Properties*: commutative, reflexive, not associative (overflow/underflow)

Unsigned Integer

- Represent only *non-negative* values ($\mathbb{N}$)
- Discontinuity in 0, $2^{32} - 1$

- ☑ Wraparound semantic $\rightarrow$ well-defined (modulo 2^{32})
- ☑ Bit-wise operations are well-defined

- *Properties*: commutative, reflexive, associative

Google Style Guide

> Because of historical accident, the C++ standard also uses unsigned integers to represent the size of containers - many members of the standards body believe this to be a mistake, but it is effectively impossible to fix at this point

Solution: use `int64_t`

max value: $2^{63} - 1 =$ 9,223,372,036,854,775,807 or
9 quintillion (9 billion of billion),
about 292 years in nanoseconds,
9 million terabytes

When use `signed` integer?

- if it can be mixed with negative values, e.g. subtracting byte sizes
- prefer expressing non-negative values with signed integer and assertions
- optimization purposes, e.g. exploit undefined behavior for overflow or in loops

When use `unsigned` integer?

- if the quantity can never be mixed with negative values (?)
- bitmask values
- optimization purposes, e.g. division, modulo
- safety-critical system, signed integer overflow could be "non-deterministic"

`Subscripts and sizes should be signed`, *Bjarne Stroustrup*
`Don't add to the signed/unsigned mess`, *Bjarne Stroustrup*
`Integer Type Selection in C++: in Safe, Secure and Correct Code`, *Robert C. Seacord*

Arithmetic Type Limits

Query properties of arithmetic types in C++11:

```
#include <limits>

std::numeric_limits<int>::max();       // 2^31 − 1
std::numeric_limits<uint16_t>::max(); // 65,535

std::numeric_limits<int>::min();       // −2^31
std::numeric_limits<unsigned>::min(); // 0
```

* this syntax will be explained in the next lectures

Promotion and Truncation

Promotion to a larger type keeps the sign

```
int16_t x = -1;
int     y = x; // sign extend
cout << y;     // print -1
```

Truncation to a smaller type is implemented as a modulo operation with respect to the number of bits of the smaller type

```
int     x = 65537; // 2^16 + 1
int16_t y = x;     // x % 2^16
cout << y;         // print 1

int     z = 32769; // 2^15 + 1 (does not fit in a int16_t)
int16_t w = z;     // (int16_t) (x % 2^16 = 32769)
cout << w;         // print -32767
```

```
unsigned a = 10;  // array is small
int      b = -1;
array[10ull + a * b] = 0; // ?
```

☠ Segmentation fault!

```
int f(int a, unsigned b, int* array) {  // array is small
    if (a > b)
        return array[a - b]; // ?
    return 0;
}
```

☠ Segmentation fault for `a < 0`!

```
// v.size() return unsigned
for (size_t i = 0; i < v.size() - 1; i++)
    array[i] = 3; // ?
```

☠ Segmentation fault for `v.size() == 0`!

Easy case:

```
unsigned x = 32;      // x can be also a pointer
x         += 2u - 4; // 2u - 4 = 2 + (2^32 - 4)
                     //        = 2^32 - 2
                     // (32 + (2^32 - 2)) % 2^32
cout << x;           // print 30 (as expected)
```

What about the following code?

```
uint64_t x = 32;      // x can be also a pointer
x         += 2u - 4;
cout << x;
```

A real-world case:

```
// allocate a zeroed rtx vector  of N elements
//
// sizeof(struct rtvec_def) == 16
// sizeof(rtunion) == 8
rtvec rtvec_alloca(int n) {
    rtvec rt;
    int i;
    rt = (rtvec)obstack_alloc(
        rtl_obstack,
        sizeof(struct rtvec_def) + ((n - 1) * sizeof(rtunion)));
//  ...
    return rt;
}
```

The C++ standard does not prescribe any specific behavior (undefined behavior) for several integer/unsigned arithmetic operations

- *Signed integer overflow/underflow*

```
int x = std::numeric_limits<int>::max() + 20;
```

- *More negative values than positive*

```
int x = std::numeric_limits<int>::max() * -1; //  (2^31 -1) * -1
cout << x;                                    //  -2^31 +1 ok

int y = std::numeric_limits<int>::min() * -1; // -2^31 * -1
cout << y; // hard to see in complex examples // 2^31 overflow!!
```

- *Initialize* an integer with a value larger than its range is undefined behavior

```
int z = 3000000000; // undefined behavior!!
```

- *Bitwise operations* on signed integer types is undefined behavior

```
int y = 1 << 12;     // undefined behavior!!
```

- *Shift* larger than #bits of the data type is undefined behavior even for **unsigned**

```
unsigned y = 1u << 32u; // undefined behavior!!
```

- *Undefined behavior in implicit conversion*

```
uint16_t a = 65535; // 0xFFFF
uint16_t b = 65535; // 0xFFFF                    expected: 4'294'836'225
cout << (a * b);    // print '-131071' undefined behavior!! (int overflow)
```

```cpp
#include <climits>
#include <cstdio>

void f(int* ptr, int pos) {
    pos++;
    if (pos < 0)    // <-- the compiler could assume that signed overflow never
        return;     //     happen and "simplify" the condition to check
    ptr[pos] = 0;
}

int main() {                  // the code compiled with optimizations, e.g. -O3
    int* tmp = new int[10];   // leads to segmentation faults with clang, while
    f(tmp, INT_MAX);          // it terminates correctly with gcc
    printf("%d\n", tmp[0]);
}
```

s/open.c of the Linux kernel

```
int do_fallocate(..., loff_t offset, loff_t len) {
    inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes) || (offset + len < 0))
        return -EFBIG;      // the compiler is able to infer that both 'offset' and
    ...                     // 'len' are non-negative and can eliminate this check,
}                           // without verify integer overflow
```

src/backend/utils/adt/int8.c of PostgreSQL

```
if (arg2 == 0) {
    ereport(ERROR, (errcode(ERRCODE_DIVISION_BY_ZERO), // the compiler is not aware
            errmsg("division by zero")));              // that this function
}                                                      // doesn't return
/* No overflow is possible */
PG_RETURN_INT32((int32) arg1 / arg2);  // the compiler assumes that the divisor is
                                       // non-zero and can move this statement on
                                       // the top  (always executed)
```

Even worse example:

```cpp
#include <iostream>

int main() {
    for (int i = 0; i < 4; ++i)
        std::cout << i * 1000000000 << std::endl;
}
// with optimizations, it is an infinite loop
// --> 1000000000 * i > INT_MAX
// undefined behavior!!

// the compiler translates the multiplication constant into an addition
```


Why does this loop produce undefined behavior?

Is the following loop safe?

```
void f(int size) {
    for (int i = 1; i < size; i += 2)
        ...
}
```

- What happens if `size` is equal to `INT_MAX`?
- How to make the previous loop safe?
- `i >= 0 && i < size` is not the solution because of *undefined behavior* of signed overflow
- Can we generalize the solution when the increment is `i += step`?

Overflow / Underflow

Detecting wraparound for unsigned integral types is **not trivial**

```
// some examples
bool is_add_overflow(unsigned a, unsigned b) {
    return (a + b) < a || (a + b) < b;
}

bool is_mul_overflow(unsigned a, unsigned b) {
    unsigned x = a * b;
    return a != 0 && (x / a) != b;
}
```

Detecting overflow/underflow for signed integral types is even harder and must be checked before performing the operation

Floating-point Types and Arithmetic

IEEE Floating-Point Standard

IEEE754 is the technical standard for floating-point arithmetic

The standard defines the binary format, operations behavior, rounding rules, exception handling, etc.

First Release : 1985
Second Release : 2008. Add 16-bit, 128-bit, 256-bit floating-point types
Third Release : 2019. Specify `min/max` behavior

see `The IEEE Standard 754: One for the History Books`

IEEE754 technical document:

`754-2019 - IEEE Standard for Floating-Point Arithmetic`

In general, **C/C++ adopts IEEE754 floating-point standard**:

`en.cppreference.com/w/cpp/types/numeric_limits/is_iec559`

32/64-bit Floating-Point

- **IEEE754 Single-precision** (32-bit) `float`

Sign	**Exponent** (or base)	**Mantissa** (or significant)
1-bit	8-bit	23-bit

- **IEEE754 Double-precision** (64-bit) `double`

Sign	**Exponent** (or base)	**Mantissa** (or significant)
1-bit	11-bit	52-bit

128/256-bit Floating-Point

- **IEEE754 Quad-Precision** (128-bit) `std::float128` C++23

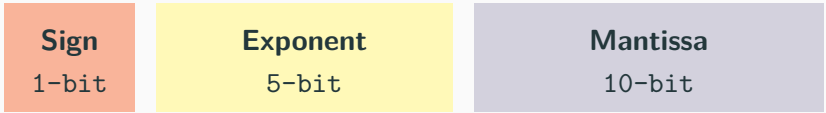
Sign	**Exponent** (or base)	**Mantissa** (or significant)
1-bit	15-bit	112-bit

- **IEEE754 Octuple-Precision** (256-bit) (not standardized in C++)

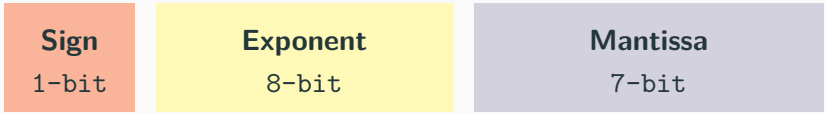
Sign	**Exponent** (or base)	**Mantissa** (or significant)
1-bit	19-bit	236-bit

16-bit Floating-Point

- **IEEE754 16-bit Floating-point** (`std::binary16`) C++23 → GPU, Arm7

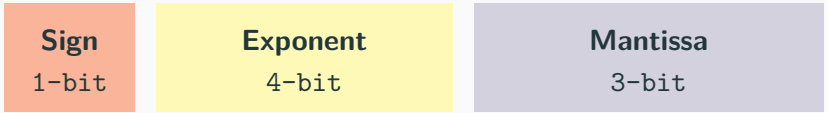


- **Google 16-bit Floating-point** (`std::bfloat16`) C++23 → TPU, GPU, Arm8

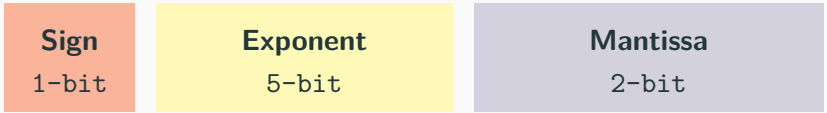


8-bit Floating-Point (Non-Standardized in C++/IEEE)

- `E4M3`



- `E5M2`



- `Floating Point Formats for Machine Learning`, *IEEE draft*
- `FP8 Formats for Deep Learning`, *Intel, Nvidia, Arm*

- **TensorFloat-32 (TF32)** Specialized floating-point format for deep learning applications

- **Posit** (John Gustafson, 2017), also called *unum III* (*universal number*), represents floating-point values with *variable-width* of exponent and mantissa.
  It is implemented in experimental platforms

- NVIDIA Hopper Architecture In-Depth
- Beating Floating Point at its Own Game: Posit Arithmetic
- Posits, a New Kind of Number, Improves the Math of AI
- Comparing posit and IEEE-754 hardware cost

- **Microscaling Formats (MX)** Specification for low-precision floating-point formats defined by AMD, Arm, Intel, Meta, Microsoft, NVIDIA, and Qualcomm. It includes FP8, FP6, FP4, (MX)INT8

- **Fixed-point** representation has a fixed number of digits after the radix point (decimal point). The gaps between adjacent numbers are always equal. The range of their values is significantly limited compared to floating-point numbers.
  It is widely used on embedded systems

- OCP Microscaling Formats (MX) Specification

Floating-point number:

- *Radix* (or base): β
- *Precision* (or digits): p
- *Exponent* (magnitude): e
- *Mantissa*: M

$$n = \underbrace{M}_{p} \times \beta^e \quad \rightarrow \quad \text{IEEE754: } 1.M \times 2^e$$

```
float  f1 = 1.3f;   // 1.3
float  f2 = 1.1e2f; // 1.1 · 10^2
float  f3 = 3.7E4f; // 3.7 · 10^4
float  f4 = .3f;    // 0.3
double d1 = 1.3;    // without "f"
double d2 = 5E3;    // 5 · 10^3
```

Exponent Bias

In IEEE754 floating point numbers, the exponent value is offset from the actual value by the **exponent bias**

- The exponent is stored as an unsigned value suitable for comparison
- Floating point values are lexicographic ordered
- For a single-precision number, the exponent is stored in the range $[1, 254]$ (0 and 255 have special meanings), and is biased by subtracting 127 to get an exponent value in the range $[-126, +127]$

0	10000111	11000000000000000000000
+	$2^{(135-127)} = 2^8$	$\frac{1}{2^1} + \frac{1}{2^2} = 0.5 + 0.25 = 0.75 \overset{normal}{\rightarrow} 1.75$

$$+1.75 * 2^8 = 448.0$$

Normal number

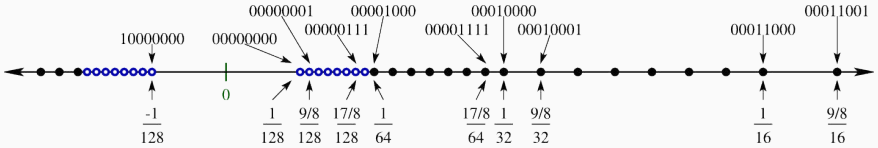
A **normal** number is a floating point value that can be represented with *at least one bit set in the exponent* or the mantissa has all 0s

Denormal number

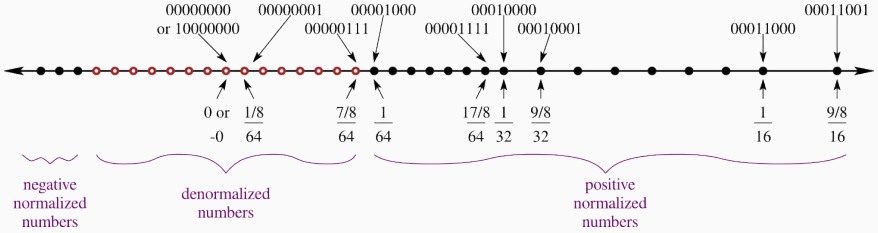
Denormal (or subnormal) numbers fill the underflow gap around zero in floating-point arithmetic. Any non-zero number with magnitude smaller than the smallest normal number is denormal

A **denormal** number is a floating point value that can be represented with *all 0s in the exponent*, but the mantissa is non-zero

Why denormal numbers make sense: (↓ normal numbers)



The problem: distance values from zero (↓ denormal numbers)



Infinity

In the IEEE754 standard, `inf` (infinity value) is a numeric data type value that exceeds the maximum (or minimum) representable value

Operations generating `inf`:

- $\pm\infty \cdot \pm\infty$
- $\pm\infty \cdot \pm$`finite_value`
- `finite_value` *op* `finite_value` $>$ `max_value`
- *finite value* $/ \pm 0$

There is a single representation for `+inf` and `-inf`

Comparison: `(inf == finite_value)` $\rightarrow$ `false`
`(`$\pm$`inf == `$\pm$`inf)` $\rightarrow$ `true`

```
cout << 5.0 / 0.0;      // print "inf"
cout << -5.0 / 0.0;     // print "-inf"

auto inf = std::numeric_limits<float>::infinity;
cout << (-0.0 == 0.0);                        // true, 0 == 0
cout << ((5.0f / inf) == ((-5.0f / inf));  // true, 0 == 0
cout << (10e40f) == (10e40f + 9999999.0f); // true, inf == inf
cout << (10e40)  == (10e40f + 9999999.0f); // false, 10e40 != inf
```

NaN

In the IEEE754 standard, `NaN` (not a number) is a numeric data type value representing an undefined or non-representable value

Floating-point operations generating `NaN`:

- Operations with a `NaN` as at least one operand
- $\pm\infty \cdot \mp\infty$, $0 \cdot \infty$
- $0/0, \infty/\infty$
- $\sqrt{x}$, $\log(x)$ for $x < 0$
- $\sin^{-1}(x), \cos^{-1}(x)$ for $x < -1$ *or* $x > 1$

Comparison: `(NaN == x)` $\rightarrow$ `false`, for every `x`

`(NaN == NaN)` $\rightarrow$ `false`

There are many representations for `NaN` (e.g. $2^{24} - 2$ for `float`)

The specific (bitwise) `NaN` value returned by an operation is implementation/compiler specific

```
cout << 0 / 0;          // undefined behavior
cout << 0.0 / 0.0;      // print "nan" or "-nan"
```

quiet_NaN

Machine Epsilon

Machine epsilon

Machine epsilon ε (or *machine accuracy*) is defined to be the smallest number that can be added to 1.0 to give a number other than one

IEEE 754 Single precision : $\varepsilon = 2^{-23} \approx 1.19209 * 10^{-7}$

IEEE 754 Double precision : $\varepsilon = 2^{-52} \approx 2.22045 * 10^{-16}$

Units at the Last Place (ULP)

ULP

Units at the Last Place is the gap between consecutive floating-point numbers

$$ULP(p, e) = \beta^{e-(p-1)} \rightarrow 2^{e-(p-1)}$$

Example:

$\beta = 10,\ p = 3$

$\pi = 3.1415926... \rightarrow x = 3.14 \times 10^0$

$ULP(3, 0) = 10^{-2} = 0.01$

Relation with ε:

- $\varepsilon = ULP(p, 0)$
- $ULP_x = \varepsilon * \beta^{e(x)}$

Floating-Point Representation of a Real Number

The machine floating-point representation **fl(**x**)** of a *real number* x is expressed as $fl(x) = x(1+\delta)$, where δ is a small constant

The approximation of a *real number* x has the following properties:

Absolute Error: $|fl(x) - x| \leq \frac{1}{2} \cdot ULP_x$

Relative Error: $\left|\frac{fl(x) - x}{x}\right| \leq \frac{1}{2} \cdot \boldsymbol{\varepsilon}$

- NaN (mantissa $\neq 0$)

*	11111111	**********

- $\pm$ infinity

*	11111111	00000000000000000000000

- Lowest/Largest ($\pm 3.40282 * 10^{+38}$)

*	11111110	11111111111111111111111

- Minimum (normal) ($\pm 1.17549 * 10^{-38}$)

*	00000001	00000000000000000000000

- Denormal number ($< 2^{-126}$)(minimum: $1.4 * 10^{-45}$)

*	00000000	**********

- ± 0

*	00000000	00000000000000000000000

	E4M3	E5M2	half
Exponent	4 [0*-14] (no inf)	5-bit [0*-30]	
Bias	7	15	
Mantissa	4-bit	2-bit	10-bit
Largest ($\pm$)	$1.75 * 2^8$ 448	$1.75 * 2^{15}$ 57,344	2^{16} 65,536
Smallest ($\pm$)	2^{-6} 0.015625	2^{-14} 0.00006	
Smallest Denormal	2^{-9} 0.001953125	2^{-16} $1.5258 * 10^{-5}$	2^{-24} $6.0 \cdot 10^{-8}$
Epsilon	2^{-4} 0.0625	2^{-2} 0.25	2^{-10} 0.00098

	`bfloat16`	`float`	`double`
Exponent	8-bit [0*-254]		11-bit [0*-2046]
Bias	127		1023
Mantissa	7-bit	23-bit	52-bit
Largest ($\pm$)	2^{128} $3.4 \cdot 10^{38}$		2^{1024} $1.8 \cdot 10^{308}$
Smallest ($\pm$)	2^{-126} $1.2 \cdot 10^{-38}$		2^{-1022} $2.2 \cdot 10^{-308}$
Smallest Denormal	/	2^{-149} $1.4 \cdot 10^{-45}$	2^{-1074} $4.9 \cdot 10^{-324}$
Epsilon	2^{-7} 0.0078	2^{-23} $1.2 \cdot 10^{-7}$	2^{-52} $2.2 \cdot 10^{-16}$

Floating-point - Limits

```cpp
#include <limits>
// T: float or double

std::numeric_limits<T>::max();        // largest value

std::numeric_limits<T>::lowest();     // lowest value (C++11)

std::numeric_limits<T>::min();        // smallest value

std::numeric_limits<T>::denorm_min() // smallest (denormal) value

std::numeric_limits<T>::epsilon();    // epsilon value

std::numeric_limits<T>::infinity()    // infinity

std::numeric_limits<T>::quiet_NaN()  // NaN
```

Floating-point - Useful Functions

```
#include <cmath> // C++11

bool std::isnan(T value)      // check if value is NaN
bool std::isinf(T value)      // check if value is ±infinity
bool std::isfinite(T value)   // check if value is not NaN
                              // and not ±infinity

bool std::isnormal(T value); // check if value is Normal

T    std::ldexp(T x, p)       // exponent shift x * 2^p
int  std::ilogb(T value)      // extracts the exponent of value
```

Floating-point operations are written

- $\oplus$ addition
- $\ominus$ subtraction
- $\otimes$ multiplication
- $\oslash$ division

$\odot \in \{\oplus, \ominus, \otimes, \oslash\}$

$op \in \{+, -, *, /\}$ denotes exact precision operations

(P1) In general, $a \; op \; b \neq a \odot b$

(P2) **Not Reflexive** $a \neq a$

- *Reflexive* without `NaN`

(P3) **Not Commutative** $a \odot b \neq b \odot a$

- *Commutative* without `NaN` (`NaN` $\neq$ `NaN`)

(P4) In general, **Not Associative** $(a \odot b) \odot c \neq a \odot (b \odot c)$

- even excluding `NaN` and `inf` in intermediate computations

(P5) In general, **Not Distributive** $(a \oplus b) \otimes c \neq (a \otimes c) \oplus (b \otimes c)$

- even excluding `NaN` and `inf` in intermediate computations

(P6) **Identity on operations is not ensured**

- $(a \ominus b) \oplus b \neq a$
- $(a \oslash b) \otimes b \neq a$

(P7) **Overflow/Underflow** Floating-point has *"saturation"* values `inf`, `-inf`

- as opposite to integer arithmetic with wrap-around behavior

Special Values Behavior

Zero behavior

- $a \oslash 0 =$ `inf`, $a \in \{finite - 0\}$ [IEEE-764], undefined behavior in C++
- $0 \oslash 0$, `inf` $\oslash 0 =$ `NaN` [IEEE-764], undefined behavior in C++
- $0 \otimes$ `inf = NaN`
- `+0 = -0` but they have a different binary representation

`Inf` behavior

- `inf` $\odot a =$ `inf`, $a \in \{finite - 0\}$
- `inf` $\oplus\otimes$ `inf = inf`
- `inf` $\ominus\oslash$ `inf = NaN`
- $\pm$ `inf` $\odot \mp$ `inf = NaN`
- $\pm$ `inf` $= \pm$ `inf`

`NaN` behavior

- `NaN` $\odot a =$ `NaN`
- `NaN` $\neq a$

Floating-Point Undefined Behavior

- **Division by zero**
  e.g., $10^8/0.0$
- **Conversion to a narrower floating-point type**:
  e.g., `0.1 double → float`
- **Conversion from floating-point to integer**:
  e.g., 10^8 `float → int`
- **Operations on signaling NaNs**: Arithmetic operations that cause an "invalid operation" exception to be signaled
  e.g., `inf - inf`
- **Incorrectly assuming IEEE-754 compliance for all platforms**:
  e.g., Some embedded Linux distribution on ARM

C++11 allows determining if a floating-point exceptional condition has occurred by using floating-point exception facilities provided in `<cfenv>`

```
#include <cfenv>
// MACRO
FE_DIVBYZERO  // division by zero
FE_INEXACT    // rounding error
FE_INVALID    // invalid operation, i.e. NaN
FE_OVERFLOW   // overflow (reach saturation value +inf)
FE_UNDERFLOW  // underflow  (reach saturation value -inf)
FE_ALL_EXCEPT // all exceptions

// functions
std::feclearexcept(FE_ALL_EXCEPT); // clear exception status
std::fetestexcept(<macro>);        // returns a value != 0 if an
                                   // exception has been detected
```

```cpp
#include <cfenv>    // floating point exceptions
#include <iostream>
#pragma STDC FENV_ACCESS ON // tell the compiler to manipulate the floating-point
                            // environment (not supported by all compilers)
                            // gcc: yes, clang: no
int main() {
    std::feclearexcept(FE_ALL_EXCEPT); // clear
    auto x = 1.0 / 0.0;                // all compilers
    std::cout << (bool) std::fetestexcept(FE_DIVBYZERO); // print true

    std::feclearexcept(FE_ALL_EXCEPT); // clear
    auto x2 = 0.0 / 0.0;               // all compilers
    std::cout << (bool)  std::fetestexcept(FE_INVALID);  // print true

    std::feclearexcept(FE_ALL_EXCEPT); // clear
    auto x4 = 1e38f * 10;              // gcc: ok
    std::cout << std::fetestexcept(FE_OVERFLOW);         // print true
}
```


see What is the difference between quiet NaN and signaling NaN?

Floating-point Issues

Ariene 5: data conversion from 64-bit floating point value to 16-bit signed integer $\rightarrow$ *$137 million*

Patriot Missile: small chopping error at each operation, 100 hours activity $\rightarrow$ *28 deaths*

Integer type is more accurate than floating type for large numbers

```
cout << 16777217;          // print 16777217
cout << (int) 16777217.0f; // print 16777216!!
cout << (int) 16777217.0;  // print 16777217, double ok
```

`float` numbers are different from `double` numbers

```
cout << (1.1 != 1.1f); // print true !!!
```

The floating point precision is finite!

```
cout << setprecision(20);
cout << 3.33333333f; // print 3.333333254!!
cout << 3.33333333;  // print 3.333333333
cout << (0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1); // print 0.59999999999999998
```

Floating point arithmetic is not associative

```
cout << 0.1 + (0.2 + 0.3) == (0.1 + 0.2) + 0.3; // print false
```

IEEE754 Floating-point computation guarantees to produce **deterministic** output, namely the exact bitwise value for each run, if and only if the **order of the operations is always the same**

→ *same result on any machine and for all runs*

"Using a double-precision floating-point value, we can represent easily the number of atoms in the universe.

If your software ever produces a number so large that it will not fit in a double-precision floating-point value, chances are good that you have a bug"

***Daniel Lemire**, Prof. at the University of Quebec*

" NASA uses just 15 digits of π to calculate interplanetary travel. With 40 digits, you could calculate the circumference of a circle the size of the visible universe with an accuracy that would fall by less than the diameter of a single hydrogen atom"

***Latest in space**, Twitter*

Floating-point Algorithms

- `addition` **algorithm** (simplified):

(1) Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent

(2) Add the mantissa

(3) Normalize the sum if needed (shift right/left the exponent by 1)

- `multiplication` **algorithm** (simplified):

(1) Multiplication of mantissas. The number of bits of the result is twice the size of the operands ($46 + 2$ bits, with $+2$ for implicit normalization)

(2) Normalize the product if needed (shift right/left the exponent by 1)

(3) Addition of the exponents

- `fused multiply-add` (`fma`):
  - Recent architectures (also GPUs) provide `fma` to compute addition and multiplication in a single instruction (performed by the compiler in most cases)
  - The rounding error of $fma(x, y, z)$ is less than $(x \otimes y) \oplus z$

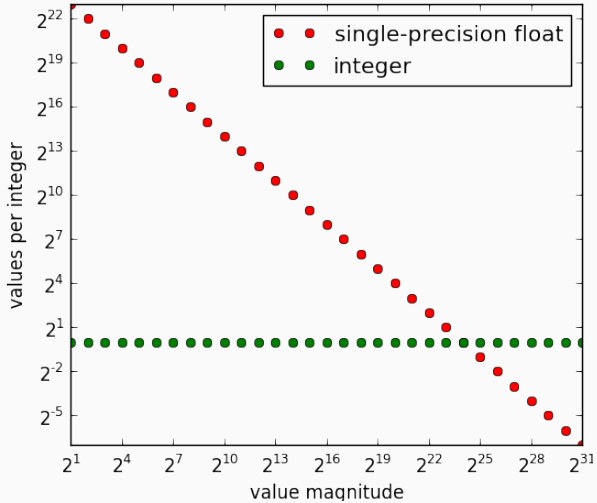
Catastrophic Cancellation

Catastrophic cancellation (or *loss of significance*) refers to loss of relevant information in a floating-point computation that cannot be revered

Two cases:

(C1) $\mathbf{a} \pm \mathbf{b}$, where $\mathbf{a} \gg \mathbf{b}$ or $\mathbf{b} \gg \mathbf{a}$. The value (or part of the value) of the smaller number is lost

(C2) $\mathbf{a} - \mathbf{b}$, where $\mathbf{a}, \mathbf{b}$ are approximation of exact values and $\mathbf{a} \approx \mathbf{b}$, namely a loss of precision in both $\mathbf{a}$ and $\mathbf{b}$. $\mathbf{a} - \mathbf{b}$ cancels most of the relevant part of the result because $\mathbf{a} \approx \mathbf{b}$. It implies a *small absolute error* but a *large relative error*



Intersection $= 16,777,216 = 2^{24}$

How many iterations performs the following code?

```
while (x > 0)
    x = x - y;
```

How many iterations?

```
float:  x = 10,000,000   y = 1     -> 10,000,000
float:  x = 30,000,000   y = 1     -> does not terminate
float:  x =    200,000   y = 0.001 -> does not terminate
bfloat: x =        256   y = 1     -> does not terminate !!
```

Floating-point increment

```
float x = 0.0f;
for (int i = 0; i < 20000000; i++)
x += 1.0f;
```

What is the value of `x` at the end of the loop?

Ceiling division $\left\lceil \frac{a}{b} \right\rceil$

```
//        std::ceil((float) 101 / 2.0f) -> 50.5f -> 51
float x = std::ceil((float) 20000001 / 2.0f);
```

What is the value of `x`?

Let's solve a quadratic equation:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$x^2 + 5000x + 0.25$

```
(-5000 + std::sqrt(5000.0f * 5000.0f - 4.0f * 1.0f * 0.25f)) / 2 // x2
(-5000 + std::sqrt(25000000.0f - 1.0f)) / 2 // catastrophic cancellation (C1)
(-5000 + std::sqrt(25000000.0f)) / 2
(-5000 + 5000) / 2 = 0                      // catastrophic cancellation (C2)
// correct result: 0.00005!!
```

relative error: $\frac{|0 - 0.00005|}{0.00005} = 100\%$

The problem

```
cout << (0.11f + 0.11f < 0.22f); // print true!!
cout << (0.1f + 0.1f > 0.2f);    // print true!!
```

Do not use absolute error margins!!

```
bool areFloatNearlyEqual(float a, float b) {
    if (std::abs(a - b) < epsilon); // epsilon is fixed by the user
        return true;
    return false;
}
```

Problems:

- Fixed `epsilon` "looks small" but it could be too large when the numbers being compared are very small
- If the compared numbers are very large, the `epsilon` could end up being smaller than the smallest rounding error, so that the comparison always returns `false`

Solution: Use relative error $\frac{|a-b|}{b} < \varepsilon$

```
bool areFloatNearlyEqual(float a, float b) {
    if (std::abs(a - b) / b < epsilon); // epsilon is fixed
        return true;
    return false;
}
```

Problems:

- `a=0, b=0` The division is evaluated as `0.0/0.0` and the whole `if` statement is `(nan < espilon)` which always returns `false`
- `b=0` The division is evaluated as `abs(a)/0.0` and the whole `if` statement is `(+inf < espilon)` which always returns `false`
- **`a and b very small.`** The result should be `true` but the division by `b` may produces wrong results
- **`It is not commutative.`** We always divide by `b`

Possible solution: $\frac{|a-b|}{\max(|a|,|b|)} < \varepsilon$

```
bool areFloatNearlyEqual(float a, float b) {
    constexpr float normal_min     = std::numeric_limits<float>::min();
    constexpr float relative_error = <user_defined>

    if (!std::isfinite(a) || !isfinite(b)) // a = ±∞, NaN or b = ±∞, NaN
        return false;
    float diff  = std::abs(a - b);
    // if "a" and "b" are near to zero, the relative error is less effective
    if (diff <= normal_min) // or also: user_epsilon *  normal_min
        return true;

    float abs_a = std::abs(a);
    float abs_b = std::abs(b);
    return (diff / std::max(abs_a, abs_b)) <= relative_error;
}
```

Minimize Error Propagation - Summary

- Prefer **multiplication/division** rather than addition/subtraction
- Try to reorganize the computation to **keep near** numbers with the same scale (e.g. sorting numbers)
- Consider **putting a zero** very small number (under a threshold). Common application: iterative algorithms
- Scale by a **power of two** is safe
- **Switch to log scale**. Multiplication becomes Add, and Division becomes Subtraction
- Use a **compensation algorithm** like Kahan summation, Dekker's FastTwoSum, Rump's AccSum

References

Suggest readings:

- What Every Computer Scientist Should Know About Floating-Point Arithmetic
- Do Developers Understand IEEE Floating Point?
- Yet another floating point tutorial
- Unavoidable Errors in Computing

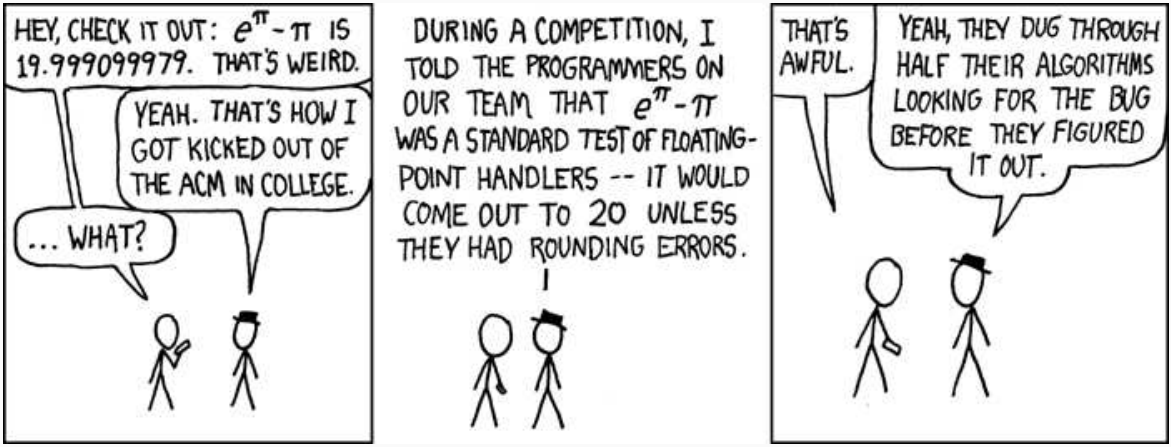
Floating-point Comparison readings:

- The Floating-Point Guide - Comparison
- Comparing Floating Point Numbers, 2012 Edition
- Some comments on approximately equal FP comparisons
- Comparing Floating-Point Numbers Is Tricky

Floating point tools:

- IEEE754 visualization/converter
- Find and fix floating-point problems

On Floating-Point



Modern C++ Programming

4. Basic Concepts III

Entities and Control Flow

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Entities

Entities

A C++ program is set of language-specific *keywords* (`for, if, new, true`, etc.), *identifiers* (symbols for variables, functions, structures, namespaces, etc.), *expressions* defined as sequence of operators, and *literals* (constant value tokens)

C++ Entity

An **entity** is a value, object, reference, function, enumerator, type, class member, or template

Identifiers and *user-defined operators* are the names used to refer to *entities*

Entities also captures the result(s) of an *expression*

Preprocessor macros are not C++ entities

Declaration and Definition

Declaration/Definition

Declaration/Prototype

A **declaration** (or *prototype*) introduces an *entity* with an *identifier* describing its type and properties

A *declaration* is what the compiler and the linker needs to accept references (usage) to that identifier

Entities can be declared multiple times. All declarations are the same

Definition/Implementation

An entity **definition** is the implementation of a declaration. It defines the properties and the behavior of the entity

For each entity, only a single *definition* is allowed

Declaration/Definition Function Example

```
void f(int a, char* b);  // function declaration

void f(int a, char*) {   // function definition
    ...                  // "b" can be omitted if not used
}

void f(int a, char* b);  // function declaration
                         // multiple declarations is valid

f(3, "abc");             // usage
```

```
void g();  // function declaration

g();       // linking error "g" is not defined
```

Declaration/Definition `struct` Example

A declaration without a concrete implementation is an incomplete type (as `void`)

```
struct A;   // declaration 1
struct A;   // declaration 2 (ok)

struct B {  // declaration and definition
    int b;
//  A   x;  // compile error incomplete type
    A*  y;  // ok, pointer to incomplete type
};

struct A {  // definition
    char c;
}
```

Enumerators

Enumerator - enum

Enumerator

An **enumerator** enum is a data type that groups a set of named integral constants

```
enum color_t { BLACK, BLUE, GREEN };

color_t color = BLUE;
cout << (color == BLACK); // print false
```

The problem:

```
enum color_t { BLACK, BLUE, GREEN };
enum fruit_t { APPLE, CHERRY };

color_t color = BLACK;    // int: 0
fruit_t fruit = APPLE;    // int: 0
bool    b     = (color == fruit); // print 'true'!!
// and, most importantly, does the match between a color and
// a fruit make any sense?
```

Strongly Typed Enumerator - `enum class`

`enum class` (C++11)

`enum class` (scoped enum) data type is a *type safe* enumerator that is not implicitly convertible to `int`

```
enum class Color { BLACK, BLUE, GREEN };
enum class Fruit { APPLE, CHERRY };

Color color = Color::BLUE;
Fruit fruit = Fruit::APPLE;

// bool b = (color == fruit) compile error we are trying to match colors with fruits
//                           BUT, they are different things entirely
// int a1 = Color::GREEN;                compile error
// int a2 = Color::RED + Color::GREEN; compile error
   int a3 = (int) Color::GREEN;         // ok, explicit conversion
```

`enum/enum class` Features

- `enum/enum class` can be compared

```
enum class Color   { RED, GREEN, BLUE };
cout << (Color::RED < Color::GREEN); // print true
```

- `enum/enum class` are automatically enumerated in increasing order

```
enum class Color { RED, GREEN = -1, BLUE, BLACK };
//                  (0)   (-1)        (0)    (1)
Color::RED == Color::BLUE; // true
```

- `enum/enum class` can contain alias

```
enum class Device { PC = 0, COMPUTER = 0, PRINTER };
```

- C++11 `enum/enum class` allows setting the underlying type

```
enum class Color : int8_t { RED, GREEN, BLUE };
```

`enum class` Features - C++17

- C++17 **`enum class`** supports *direct-list-initialization*

```
enum class Color { RED, GREEN, BLUE };
Color a{2}; // ok, equal to Color:BLUE
```

- C++17 **`enum/enum class`** support *attributes*

```
enum class Color { RED, GREEN, BLUE [[deprecated]] };
auto x = Color::BLUE; // compiler warning
```

`enum class` Features - C++20

- C++20 allows introducing the enumerator identifiers into the local scope to decrease the verbosity

```
enum class Color { RED, GREEN, BLUE };

switch (x) {
    using enum Color; // C++20
    case RED:
    case GREEN:
    case BLUE:
}
```

`enum/enum class` - Common Errors

- `enum/enum class` should be always initialized

```
enum class Color { RED, GREEN, BLUE };

Color my_color;  // "my_color" may be outside RED, GREEN, BLUE!!
```

- C++17 Cast from *out-of-range values* respect to the *underlying type* of `enum/enum class` leads to undefined behavior

```
enum Color : uint8_t { RED, GREEN, BLUE };

Color value = 256; // undefined behavior
```

enum/enum class and constexpr★

- C++17 `constexpr` expressions don't allow *out-of-range values* for (only) `enum` without explicit *underlying type*

```
enum       Color        { RED };
enum       Fruit : int  { APPLE };
enum class Device       { PC };

// constexpr Color  a1 = (Color)  -1; compile error
   const     Color  a2 = (Color)  -1; // ok
   constexpr Fruit  a3 = (Fruit)  -1; // ok
   constexpr Device a4 = (Device) -1; // ok
```

`struct`, Bitfield, and `union`

A `struct` (*structure*) aggregates different variables into a single unit

```
struct A {
    int   x;
    char  y;
};
```

It is possible to declare one or more variables after the definition of a `struct`

```
struct A {
    int x;
} a, b;
```

Enumerators can be declared within a `struct` without a name

```
struct A {
    enum {X, Y}
};
A::X;
```

It is possible to declare a `struct` in a local scope (with some restrictions), e.g. function scope

```
int f() {
    struct A {
        int x;
    } a;
    return a.x;
}
```

Anonymous and Unnamed `struct`★

Unnamed `struct`: a structured without a name, but with an associated type

Anonymous `struct`: a structured without a name and type

The C++ standard allows *unnamed* `struct` but, contrary to C, does not allow *anonymous* `struct` (i.e. without a name)

```
struct {
   int x;
} my_struct;          // unnamed struct, ok

struct S {
   int x;
   struct { int y; }; // anonymous struct, compiler warning with -Wpedantic
};                    // -Wpedantic: diagnose use of non-strict ISO C++ extensions
```

Bitfield

A **bitfield** is a variable of a structure with a predefined bit width. A bitfield can hold bits instead bytes

```
struct S1 {
    int b1 : 10;  // range [0, 1023]
    int b2 : 10;  // range [0, 1023]
    int b3 : 8;   // range [0, 255]
}; // sizeof(S1): 4 bytes

struct S2 {
    int b1 : 10;
    int    : 0;   // reset: force the next field
    int b2 : 10;  //        to start at bit 32
}; // sizeof(S1): 8 bytes
```

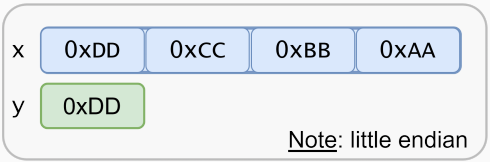
Union

A `union` is a special data type that allows to store different data types in the same memory location

- The `union` is only as big as necessary to hold its *largest* data member
- The `union` is a kind of *"overlapping"* storage

```
union A {
    int  x;
    char y;
};
```

```
A a;
a.x = 0xAABBCCDD
```



```
union A {
    int  x;
    char y;
};  // sizeof(A): 4

A a;
a.x = 1023;    // bits: 00..000001111111111
a.y = 0;       // bits: 00..000001100000000
cout << a.x;  // print 512 + 256 = 768
```

NOTE: Little-Endian encoding maps the bytes of a value in memory in the reverse order. `y` maps to the last byte of `x`

Contrary to `struct`, C++ allows *anonymous* `union` (i.e. without a name)

C++17 introduces `std::variant` to represent a *type-safe union*

[[deprecated]] Attribute ★

C++14 allows to deprecate, namely discourage, use of entities by adding the `[[deprecated]]` attribute, optionally with a message `[[deprecated("reason")]]`. It applies to:

- Functions
- Variables
- Classes and structures
- Enumerators. Single value enumerator in C++17
- Types
- Namespaces

```cpp
[[deprecated]] void f() {}

struct [[deprecated]] S1 {};

using MyInt [[deprecated]] = int;

struct S2 {
    [[deprecated]] int var = 3;
    [[deprecated]] static constexpr int var2 = 4;
};

f();       // warning
S1    s1; // warning
MyInt i;  // warning
S2{}.var; // warning
```

```cpp
enum [[deprecated]] E { EnumValue };

enum class MyEnum { A, B [[deprecated]] = 42 }; // C++17

namespace [[deprecated("please use my_ns_v2")]] my_ns {
    const int x = 5;
}

auto x = EnumValue; // warning
MyEnum::B;          // warning
my_ns::x;           // warning, "please use my_ns_v2"
```

Control Flow

`if` Statement

The `if` statement executes the first branch if the specified condition is evaluated to `true`, the second branch otherwise

- *Short-circuiting*:

```
if (<true expression> r| array[-1] == 0)
... // no error!! even though index is -1
    // left-to-right evaluation
```

- *Ternary operator*:

  `<cond> ? <expression1> : <expression2>`

  `<expression1>` and `<expression2>` must return a value of the same or convertible type

```
int value = (a == b) ? a : (b == c ? b : 3); // nested
```

`for` and `while` Loops

- `for`

```
for ([init]; [cond]; [increment]) {
    ...
}
```

To use when number of iterations is known

- `while`

```
while (cond) {
    ...
}
```

To use when number of iterations is not known

- `do while`

```
do {
...
} while (cond);
```

To use when number of iterations is not known, but there is at least one iteration

`for` Loop Features and Jump Statements

- C++ allows "in loop" definitions:

```
for (int i = 0, k = 0; i < 10; i++, k += 2)
    ...
```

- Infinite loop:

```
for (;;)   // also while(true);
   ...
```

- Jump statements (`break`, `continue`, `return`):

```
for (int i = 0; i < 10; i++) {
    if (<condition>)
        break;    // exit from the loop
    if (<condition>)
        continue; // continue with a new iteration and exec. i++
    return;       // exit from the function
}
```

C++11 introduces the **range-based `for` loop** to simplify the verbosity of traditional `for` loop constructs. They are equivalent to the `for` loop operating over a range of values, but **safer**

The range-based `for` loop avoids the user to specify start, end, and increment of the loop

```
for (int v : { 3, 2, 1 }) // INITIALIZER LIST
    cout << v << " ";     // print: 3 2 1

int values[] = { 3, 2, 1 };
for (int v : values)      // ARRAY OF VALUES
    cout << v << " ";     // print: 3 2 1

for (auto c : "abcd")     // RAW STRING
    cout << c << " ";     // print: a b c d
```

Range-based for loop can be applied in three cases:

- Fixed-size array `int array[3]`, `"abcd"`
- Branch Initializer List `{1, 2, 3}`
- Any object with `begin()` and `end()` methods

```cpp
std::vector vec{1, 2, 3, 4};

for (auto x : vec) {
   cout << x << ", ";
// print:  "1, 2, 3, 4"
```

```cpp
int matrix[2][4];
for (auto& row : matrix) {
    for (auto element : row)
        cout << "@";
    cout << "\n";
}
// print:  @@@@
//         @@@@
```

C++17 extends the concept of **range-based loop** for *structure binding*

```
struct A {
    int x;
    int y;
};

A array[] = { {1,2}, {5,6}, {7,1} };
for (auto [x1, y1] : array)
    cout << x1 << "," << y1 << "  ";  // print: 1,2  5,6  7,1
```

The `switch` statement evaluates an expression (`int` , `char` , `enum class` , `enum`) and executes the statement associated with the matching case value

```
char x = ...
switch (x) {
    case 'a': y = 1; break;
    default:  return -1;
}
return y;
```

Switch scope:

```
int x = 1;
switch (1) {
    case 0: int x;       // nearest scope
    case 1: cout << x;   // undefined!!
    case 2: { int y; }   // ok
//  case 3: cout << y;   // compile error
}
```

Fall-through:

```
MyEnum x
int y = 0;
switch (x) {
   case MyEnum::A:         // fall-through
   case MyEnum::B:         // fall-through
   case MyEnum::C: return 0;
   default: return -1;
}
```

C++17 `[[fallthrough]]` attribute

```
char x = ...
switch (x) {
    case 'a': x++;
              [[fallthrough]]; // C++17: avoid warning
    case 'b': return 0;
    default:  return -1;
}
```

Control Flow with Initializing Statement

Control flow with **initializing statement** aims at simplifying complex actions before the condition evaluation and restrict the scope of a variable which is visible only in the control flow body

C++17 introduces `if` statement with initializer

```
if (int ret = x + y; ret < 10)
    cout << ret;
```

C++17 introduces `switch` statement with initializer

```
switch (auto i = f(); x) {
   case 1: return i + x;
```

C++20 introduces `range-for loop` statement with initializer

```
for (int i = 0; auto x : {'A', 'B', 'C'})
    cout << i++ << ":" << x << " ";  // print: 0:A  1:B  2:C
```

When goto could be useful:

```
bool flag = true;
for (int i = 0; i < N && flag; i++) {
    for (int j = 0; j < M && flag; j++) {
        if (<condition>)
            flag = false;
    }
}
```

become:

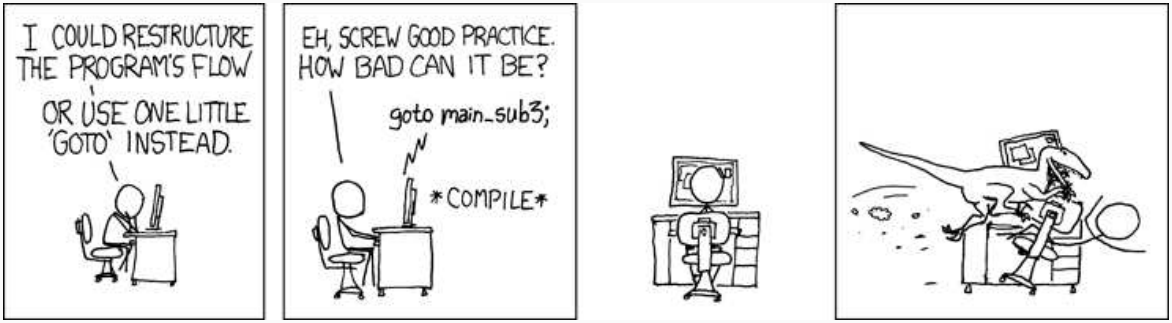
```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        if (<condition>)
            goto LABEL;
    }
}
LABEL: ;
```

Best solution:

```
bool my_function(int M, int M) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            if (<condition>)
                return false;
        }
    }
    return true;
}
```

Junior: what's wrong with goto command?

goto command:


I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE

Most compilers issue a warning when a variable is unused. There are different situations where a variable is expected to be unused

```
// EXAMPLE 1: macro dependency
int f(int value) {
    int x = value;
#if defined(ENABLE_SQUARE_PATH)
    return x * x;
#else
    return 0;
#endif
}
```

```cpp
// EXAMPLE 2: constexpr dependency (MSVC)
template<typename T>
int f(T value) {
    if constexpr (sizeof(value) >= 4)
        return 1;
    else
        return 2;
}
```

```cpp
// EXAMPLE 3: decltype dependency (MSVC)
template<typename T>
int g(T value) {
    using R = decltype(value);
    return R{};
}
```

There are different ways to solve the problem depending on the standard used

- Before C++17: `static_cast<void>(var)`
- C++17 `[[maybe_unused]]` attribute
- C++26 `auto _`

```cpp
[[maybe_unused]] int x = value;
int y  = 3;
static_cast<void>(y);
auto _ = 3;
auto _ = 4; // _ repetition is not an error

void f([[maybe_unused]] int x) {}
```

Modern C++ Programming

5. Basic Concepts IV

Memory Concepts

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Heap and Stack

Parenthesis and Brackets

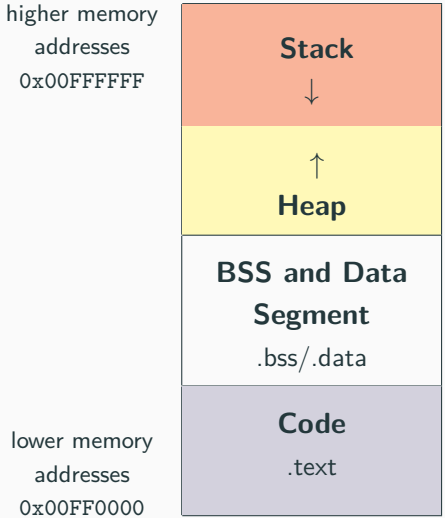
{} **braces**, informally “curly brackets”

[] **brackets**, informally “square brackets”

() **parenthesis**, informally “round brackets”

<> **angle brackets**

Process Address Space



stack memory	`int data[10]`
dynamic memory	`new int[10]` `malloc(40)`
Static/Global data	`int data[10]` (global scope)

Data and BSS Segment

```
int data[]            = {1, 2}; // DATA segment memory
int big_data[1000000] = {};     // BSS segment memory
// (zero-initialized)

int main() {
    int A[] = {1, 2, 3}; // stack memory
}
```

Data/BSS (Block Started by Symbol) segments are larger than stack memory (max $\approx$ 1GB in general) but slower

Stack and Heap Memory Overview

	Stack	Heap
Memory Organization	Contiguous (LIFO)	Contiguous within an allocation, Fragmented between allocations (relies on virtual memory)
Max size	Small (8MB on Linux, 1MB on Windows)	Whole system memory
If exceed	Program crash at function entry (hard to debug)	Exception or `nullptr`
Allocation	Compile-time	Run-time
Locality	High	Low
Thread View	Each thread has its own stack	Shared among threads

Stack Memory

A local variable is either in the stack memory or CPU registers

```
int x = 3; // not on the stack (data segment)

struct A {
    int k; // depends on where the instance of A is
};
int main() {
    int   y   = 3;          // on stack
    char  z[] = "abc";      // on stack
    A     a;                // on stack (also k)
    void* ptr = malloc(4); // variable "ptr" is on the stack
}
```

The organization of the stack memory enables much higher performance. On the other hand, this memory space is limited!!

Stack Memory Data

Types of data stored in the stack:

Local variables	Variable in a local scope
Function arguments	Data passed from caller to a function
Return addresses	Data passed from a function to a caller
Compiler temporaries	Compiler specific instructions
Interrupt contexts	

Stack Memory

Every object which resides in the stack is not valid outside his scope!!

```
int* f() {
    int array[3] = {1, 2, 3};
    return array;
}
int* ptr = f();
cout << ptr[0]; // Illegal memory access!! ☠
```

```
void g(bool x) {
    const char* str = "abc";
    if (x) {
        char xyz[] = "xyz";
        str        = xyz;
    }
    cout << str; // if "x" is true, then Illegal memory access!! ☠
}
```

Heap Memory - `new`, `delete` Keywords

`new`, `delete`

`new`/`new[]` and `delete`/`delete[]` are C++ *keywords* that perform dynamic memory allocation/deallocation, and object construction/destruction at runtime

`malloc` and `free` are C functions and they only allocate and free *memory blocks* (expressed in bytes)

`new`, `delete` Advantages

- **Language keywords**, not functions → *safer*
- **Return type**: `new` returns exact data type, while `malloc()` returns `void*`
- **Failure**: `new` throws an *exception*, while `malloc()` returns a `NULL` pointer → *it cannot be ignored*, zero-size allocations do not need special code
- **Allocation size**: The number of bytes is calculated by the compiler with the `new` keyword, while the user must take care of manually calculate the size for `malloc()`
- **Initialization**: `new` can be used to initialize besides allocate
- **Polymorphism**: objects with `virtual` functions must be allocated with `new` to initialize the virtual table pointer

Dynamic Memory Allocation

- **Allocate a single element**

```
int* value = (int*) malloc(sizeof(int)); // C
int* value = new int;                    // C++
```

- **Allocate *N* elements**

```
int* array = (int*) malloc(N * sizeof(int)); // C
int* array = new int[N];                     // C++
```

- **Allocate *N* structures**

```
MyStruct* array = (MyStruct*) malloc(N * sizeof(MyStruct)); // C
MyStruct* array = new MyStruct[N];                          // C++
```

- **Allocate and zero-initialize *N* elements**

```
int* array = (int*) calloc(N, sizeof(int)); // C
int* array = new int[N]();                  // C++
```

Dynamic Memory Deallocation

- **Deallocate a single element**

```
int* value = (int*) malloc(sizeof(int)); // C
free(value);

int* value = new int;                    // C++
delete value;
```

- **Deallocate N elements**

```
int* value = (int*) malloc(N * sizeof(int)); // C
free(value);

int* value = new int[N];                     // C++
delete[] value;
```

Allocation/Deallocation Properties

Fundamental rules:

- Each object allocated with `malloc()` must be deallocated with `free()`
- Each object allocated with `new` must be deallocated with `delete`
- Each object allocated with `new[]` must be deallocated with `delete[]`
- `malloc()`, `new`, `new[]` never produce `NULL` pointer in the *success* case, except for zero-size allocations (implementation-defined)
- `free()`, `delete`, and `delete[]` applied to `NULL` / `nullptr` pointers do not produce errors

Mixing `new`, `new[]`, `malloc` with something different from their counterparts leads to *undefined behavior*

Easy on the stack - dimensions known at compile-time:

```
int A[3][4]; // C/C++ uses row-major order: move on row elements, then columns
```

Dynamic Memory 2D allocation/deallocation - dimensions known at run-time:

```
int** A = new int*[3];        // array of pointers allocation
for (int i = 0; i < 3; i++)
    A[i] = new int[4];        // inner array allocations

for (int i = 0; i < 3; i++)
    delete[] A[i];            // inner array deallocations
delete[] A;                   // array of pointers deallocation
```

Dynamic memory 2D allocation/deallocation C++11:

```
auto A = new int[3][4];     // allocate 3 objects of type int[4]
int  n = 3;                 // dynamic value
auto B = new int[n][4];     // ok
// auto C = new int[n][n]; // compile error
delete[] A;                 // same for B, C
```

Non-Allocating Placement ★

A **non-allocating placement** `(ptr) type` allows to explicitly specify the memory location (previously allocated) of individual objects

```
// STACK MEMORY
char   buffer[8];
int*   x = new (buffer) int;
short* y = new (x + 1)  short[2];
// no need to deallocate x, y
```

```
// HEAP MEMORY
unsigned* buffer2 = new unsigned[2];
double*   z       = new (buffer2) double;
delete[] buffer2; // ok
// delete[] z;    // ok, but bad practice
```

Non-Allocating Placement and Objects ★ ⇝

Placement allocation of *non-trivial objects* requires to explicitly call the object destructor as the runtime is not able to detect when the object is out-of-scope

```
struct A {
    ~A() { cout << "destructor"; }
};

char buffer[10];
auto x = new (buffer) A();
// delete x; // runtime error 'x' is not a valid heap memory pointer
x->~A();    // print "destructor"
```

Non-Throwing Allocation ★

The `new` operator allows a non-throwing allocation by passing the `std::nothrow` object. It returns a `NULL` pointer instead of throwing `std::bad_alloc` exception if the memory allocation fails

```
int* array = new (std::nothrow) int[very_large_size];
```

note: `new` can return `NULL` pointer even if the allocated size is 0

`std::nothrow` doesn't mean that the allocated object(s) cannot throw an exception itself

```
struct A {
    A() { throw std::runtime_error{}; }
};
A* array = new (std::nothrow) A; // throw std::runtime_error
```

Memory Leak

A **memory leak** is a dynamically allocated entity in the heap memory that is no longer used by the program, but still maintained overall its execution

Problems:

- Illegal memory accesses → segmentation fault/wrong results
- Undefined values a their propagation→ segmentation fault/wrong results
- Additional memory consumption (potential segmentation fault)

```
int main() {
    int* array = new int[10];
    array      = nullptr; // memory leak!!
} // the memory can no longer be deallocated!!
```

Note: the memory leaks are especially difficult to detect in complex code and when objects are widely used

Dynamic Memory Allocation and OS

A program does not directly allocate memory itself but, it asks for a chuck of memory to the OS. The OS provides the memory at the granularity of *memory pages* (virtual memory), e.g. 4KB on Linux

Implication: out-of-bound accesses do not always lead to segmentation fault (lucky case). The worst case is an execution with undefined behavior

```
int* x         = new int;
int  num_iters = 4096 / sizeof(int); // 4 KB

for (int i = 0; i < num_iters; i++)
    x[i] = 1; // ok, no segmentation fault
```

Initialization

Variable Initialization

C++03:

```
int a1;          // default initialization (undefined value)

int a2(2);       // direct (or value) initialization
int a3(0);       // direct (or value) initialization (zero-initialization)
// int a4();     // a4 is a function

int a5 = 2;      // copy initialization
int a6 = 2u;     // copy initialization (+ implicit conversion)
int a7 = int(2); // copy initialization
int a8 = int();  // copy initialization (zero-initialization)

int a9 = {2};    // copy list initialization, brace-initialization/braced-init-list syntax
```

Uniform Initialization

C++11 **Uniform Initialization** syntax allows to initialize different entities (variables, objects, structures, etc.) in a consistent way with brace-initialization or braced-init-list syntax:

```
int b1{2};        // direct list (or value) initialization
int b2{};         // direct list (or value) initialization (default constructor/
                  //                                        zero-initialization)
int b3 = int{};   // copy initialization (default constr./zero-initialization)
int b4 = int{4}; // copy initialization

int b5 = {};      // copy list initialization (default constr./zero-initialization)
```

Brace Initialization Advantages

The **uniform initialization** can be also used to *safely* convert arithmetic types, preventing implicit *narrowing*, i.e potential value loss. The syntax is also more concise than modern casts

```
int          b4 = -1; // ok
int          b5{-1};  // ok
unsigned     b6 = -1; // ok
//unsigned b7{-1};   // compile error

float    f1{10e30};   // ok
float    f2 = 10e40;  // ok, "inf" value
//float f3{10e40};   // compile error
```

Arrays are *aggregate* types and can be initialized with brace-initialization syntax, also called braced-init-list or aggregate-initialization

One dimension:

```
int  a[3] = {1, 2, 3}; // explicit size
int  b[]  = {1, 2, 3}; // implicit size
char c[]  = "abcd";    // implicit size
int  d[3] = {1, 2};    // d[2] = 0 -> zero/default value

int  e[4] = {0};       // all values are initialized to 0
int  f[3] = {};        // all values are initialized to 0 (C++11)
int  g[3] {};          // all values are initialized to 0 (C++11)
```

Two dimensions:

```
int a[][2] = { {1,2}, {3,4}, {5,6} }; // ok
int b[][2] = { 1, 2, 3, 4 };          // ok
// the type of "a" and "b" is an array of type int[]

// int c[][]  = ...;                  // compile error
// int d[2][] = ...;                  // compile error
```

Structures are also *aggregate* types and can be initialized with brace-initialization syntax, also called braced-init-list or aggregate-initialization

```
struct S {
    unsigned x;
    unsigned y;
};
S s1;           // default initialization, x,y undefined values
S s2 = {};      // copy list initialization, x,y default constr./zero-init
S s3 = {1, 2};  // copy list initialization, x=1, y=2
S s4 = {1};     // copy list initialization, x=1, y default constr./zero-init
//S s5(3, 5);   // compiler error, constructor not found

S f() {
    S s6 = {1, 2}; // verbose
    return s6;
}
```

```
struct S {
    unsigned x;
    unsigned y;
    void*    ptr;
};

S s1{};          // direct list (or value) initialization
                 //     x,y,ptr default constr./zero-initialization

S s2{1, 2};      // direct list (or value) initialization
                 //     x=1, y=2, ptr default constr./zero-initialization

// S s3{1, -2}; // compile error, narrowing conversion

S f() { return {3, 2}; } // non-verbose
```

Non-Static Data Member Initialization (NSDMI), also called *brace or equal initialization*:

```
struct S {
    unsigned x = 3; // equal initialization
    unsigned y = 2; // equal initialization
};
struct S1 {
    unsigned x {3}; // brace initialization
};
//-------------------------------------------------------------------------------
S s1;       // call default constructor (x=3, y=2)
S s2{};     // call default constructor (x=3, y=2)
S s3{1, 4}; // set x=1, y=4

S1 s4;      // call default constructor (x=3)
S1 s5{3};   // set x=3
```

C++20 introduces the **designated initializer list**

```
struct A {
    int x, y, z;
};
A a1{1, 2, 3};                  // is the same of
A a2{.x = 1, .y = 2, .z = 3}; // designated initializer list
```

Designated initializer list can be very useful for improving code readability

```
void f1(bool a, bool b, bool c, bool d, bool e) {}
// long list of the same data type -> error-prone

struct B {
    bool a, b, c, d, e;
};                              // f2(B b)
f2({.a = true, .c = true});  // b, d, e = false
```

Structure Binding

Structure Binding declaration C++17 binds the specified names to elements of initializer:

```
struct A {
    int x = 1;
    int y = 2;
} a;

A f() { return A{4, 5}; }
// Case (1): struct
auto [x1, y1] = a;      // x1=1, y1=2
auto [x2, y2] = f();    // x2=4, y2=5
// Case (2): raw arrays
int  b[2]     = {1,2};
auto [x3, y3] = b;      // x3=1, y3=2
// Case (3): tuples
auto [x4, y4] = std::tuple<float, int>{3.0f, 2};
```

Dynamic Memory Initialization

Dynamic memory initialization applies the same rules of the object that is allocated

C++03:

```
int* a1 = new int;          // undefined
int* a2 = new int();        // zero-initialization, call "= int()"
int* a3 = new int(4);       // allocate a single value equal to 4
int* a4 = new int[4];       // allocate 4 elements with undefined values
int* a5 = new int[4]();     // allocate 4 elements zero-initialized, call "= int()"
// int* a6 = new int[4](3); // not valid
```

C++11:

```
int* b1 = new int[4]{};     // allocate 4 elements zero-initialized, call "= int{}"
int* b2 = new int[4]{1, 2}; // set first, second, zero-initialized
```

Initialization - Undefined Behavior Example ★

lib/libc/stdlib/rand.c of the FreeBSD libc

```
struct timeval tv;
unsigned long junk;                          // not initialized, undefined value

/* XXX left uninitialized on purpose */
gettimeofday(&tv, NULL);
srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
  // A compiler can assign any value not only to the variable,
  // but also to expressions derived from the variable

  // GCC assigns junk to a register. Clang further eliminates computation
  // derived from junk completely, and generates code that does not use
  // either gettimeofday or getpid
```

Pointers and References

Pointer

A **pointer** `T*` is a value referring to a location in memory

Pointer Dereferencing

Pointer **dereferencing** (`*ptr`) means obtaining the value stored in at the location referred to the pointer

Subscript Operator []

The subscript operator (`ptr[]`) allows accessing to the pointer element at a given position

The **type of a pointer** (e.g. `void*`) is an *unsigned* integer of 32-bit/64-bit depending on the underlying architecture

- It only supports the operators `+, -, ++, --`, comparisons `==, !=, <, <=, >, >=`, subscript `[]`, and dereferencing `*`
- A pointer can be *explicitly* converted to an integer type

```
void*  x;
size_t y = (size_t) x; // ok (explicit conversion)
// size_t y = x;       // compile error (implicit conversion)
```

Pointer Conversion

- Any pointer type can be implicitly converted to `void*`
- Non-`void` pointers must be explicitly converted
- `static_cast`† does not allow pointer conversion for safety reasons, except for `void*`

```
int*  ptr1 = ...;
void* ptr2 = ptr1;          // int* -> void*, implicit conversion

void* ptr3 = ...;
int*  ptr4 = (int*) ptr3;   // void* -> int, explicit conversion required
                            // static_cast allowed
int*  ptr5 = ...;
char* ptr6 = (char*) ptr5; // int* -> char*, explicit conversion required,
                            // static_cast not allowed, dangerous
```

† see next lectures for `static_cast` details

Dereferencing:

```
int* ptr1 = new int;
*ptr1     = 4;     // dereferencing (assignment)
int a     = *ptr1; // dereferencing (get value)
```

Array subscript:

```
int* ptr2 = new int[10];
ptr2[2]   = 3;
int var   = ptr2[4];
```

Common error:

```
int *ptr1, ptr2;  // one pointer and one integer!!
int *ptr1, *ptr2; // ok, two pointers
```

Subscript operator meaning:

`ptr[i]` is equal to `*(ptr + i)`

Note: subscript operator accepts also negative values

Pointer arithmetic rule:

`address(ptr + i) = address(ptr) + (sizeof(T) * i)`

where `T` is the type of elements pointed by `ptr`

```
int array[4] = {1, 2, 3, 4};
cout << array[1];      // print 2
cout << *(array + 1); // print 2
cout << array;         // print 0xFFFAFFF2
cout << array + 1;     // print 0xFFFAFFF6!!
int* ptr = array + 2;
cout << ptr[-1];       // print 2
```

`char arr[4] = "abc"`

value	address	
'a'	0x0	←arr[0]
'b'	0x1	←arr[1]
'c'	0x2	←arr[2]
'\0'	0x3	←arr[3]

`int arr[3] = {4,5,6}`

value	address	
4	0x0	←arr[0]
	0x1	
	0x2	
	0x3	
5	0x4	←arr[1]
	0x5	
	0x6	
	0x7	
6	0x8	←arr[2]
	0x9	
	0x10	
	0x11	

Pointer Arithmetic - Undefined Behavior 3/3

lib/vsprintf.c of the Linux kernel

```
int vsnprintf(char *buf, size_t size, ...) {
    char *end;
    /* Reject out-of-range values early
       Large positive sizes are used for unknown buffer sizes */
    if (WARN_ON_ONCE((int) size < 0))
        return 0;
    end = buf + size;
    /* Make sure end is always >= buf */
    if (end < buf) { ... } // Even if pointers are represented with unsigned values,
    ...                    // pointer overflow is undefined behavior.
                           // Both GCC and Clang will simplify the overflow check
                           // buf + size < buf to size < 0 by eliminating
}                          // the common term buf
```

Address-of operator &

The **address-of operator** (&) returns the address of a variable

```
int  a = 3;
int* b = &a; // address-of operator,
             // 'b' is equal to the address of 'a'
a++;
cout << *b;  // print 4;
```

To not confuse with **Reference syntax:** `T& var = ...`

Wild and Dangling Pointers

Wild pointer:

```
int main() {
    int* ptr;   // wild pointer: Where will this pointer points?
    ...         // solution: always initialize a pointer
}
```

Dangling pointer:

```
int main() {
    int* array = new int[10];
    delete[] array; // ok -> "array" now is a dangling pointer
    delete[] array; // double free or corruption!!
    // program aborted, the value of "array" is not null
}
```

note:

```
int* array = new int[10];
delete[] array;  // ok -> "array" now is a dangling pointer
array = nullptr; // no more dagling pointer
delete[] array;  // ok, no side effect
```

`void` Pointer - Generic Pointer

Instead of declaring different types of pointer variable it is possible to declare single pointer variable which can act as any pointer types

- `void*` can be compared
- Any pointer type can be implicitly converted to `void*`
- Other operations are unsafe because the compiler does not know what kind of object is really pointed to

```
cout << (sizeof(void*) == sizeof(int*));  // print true

int array[] = { 2, 3, 4 };
void* ptr   = array; // implicit conversion
cout << *array;      // print 2
// *ptr;             // compile error
// ptr + 2;          // compile error
```

Reference

A variable **reference** `T&` is an **alias**, namely another name for an already existing variable. Both variable and variable reference can be applied to refer the value of the variable

- A pointer has its own memory address and size on the stack, reference shares the **same memory address** (with the original variable)
- The compiler can internally implement references as *pointers*, but treats them in a very different way

References are safer than pointers:

- References **cannot have NULL** value. You must always be able to assume that a reference is connected to a legitimate storage
- References **cannot be changed**. Once a reference is initialized to an object, it cannot be changed to refer to another object
  (Pointers can be pointed to another object at any time)
- References must be **initialized** when they are created
  (Pointers can be initialized at any time)

Reference - Examples

Reference syntax: `T& var = ...`

```
//int& a;      // compile error no initialization
//int& b = 3; // compile error "3" is not a variable
int  c = 2;
int& d = c;   // reference. ok valid initialization
int& e = d;   // ok. the reference of a reference is a reference
++d;          // increment
++e;          // increment
cout << c;    // print 4
```

```
int  a = 3;
int* b = &a; // pointer
int* c = &a; // pointer
++b;         // change the value of the pointer 'b'
++*c;        // change the value of 'a' (a = 4)
int& d = a;  // reference
++d;         // change the value of 'a'  (a = 5)
```

Reference vs. pointer arguments:

```
void f(int* value) {} // value may be a nullptr

void g(int& value) {} // value is never a nullptr

int a = 3;
f(&a);   // ok
f(0);    // dangerous but it works!! (but not with other numbers)
//f(a);  // compile error "a" is not a pointer

g(a);    // ok
//g(3);  // compile error "3" is not a reference of something
//g(&a); // compile error "&a" is not a reference
```

References can be use to indicate fixed size arrays:

```
void f(int (&array)[3]) { // accepts only arrays of size 3
    cout << sizeof(array);
}
void g(int array[]) {
    cout << sizeof(array); // any surprise?
}

int  A[3], B[4];
int* C = A;
//-----------------------------------------------------
f(A);    // ok
// f(B); // compile error B has size 4
// f(C); // compile error C is a pointer
g(A);   // ok
g(B);   // ok
g(C);   // ok
```

Reference - Arrays★

```
int A[4];
int (&B)[4] = A;      // ok, reference to array
int C[10][3];
int (&D)[10][3] = C; // ok, reference to 2D array

auto c = new int[3][4];  // type is int (*)[4]
// read as "pointer to arrays of 4 int"
// int (&d)[3][4] = c;    // compile error
// int (*e)[3]    = c;    // compile error
int (*f)[4] = c;          // ok
```

```
int array[4];
// &array is a pointer to an array of size 4
int size1 = (&array)[1] - array;
int size2 = *(&array + 1) - array;
cout << size1; // print 4
cout << size2; // print 4
```

`struct` Member Access

- The `dot` (`.`) operator is applied to local objects and references
- The `arrow` operator (`->`) is used with a pointer to an object

```
struct A {
   int x;
};

A a;          // local object
a.x;          // dot syntax

A& ref = a;   // reference
ref.x;        // dot syntax

A* ptr = &a; // pointer
ptr->x;       // arrow syntax: same of *ptr.x
```

Constants, Literals, `const`, `constexpr`, `consteval`, `constinit`

Constants and Literals

A **constant** is an expression that can be *evaluated at compile-time*

A **literal** is a *fixed value* that can be assigned to a *constant*

formally, *"Literals are the tokens of a C++ program that represent constant values embedded in the source code"*

Literal types:

- **Concrete values** of the scalar types `bool`, `char`, `int`, `float`, `double`, e.g. `true`, `'a'`, `3`, `2.0f`
- **String literal** of type `const char[]`, e.g `"literal"`
- `nullptr`
- User-defined literals, e.g. `2s`

const Keyword

const keyword

The const keyword declares an object that **never changes** value after the initialization. A const variable must be initialized when declared

A const variable is evaluated at compile-time value if the right expression is also evaluated at compile-time

```
int size = 3;              // 'size' is dynamic
int A[size] = {1, 2, 3}; // technically possible but, variable size stack array
                           // are considered BAD programming
const int SIZE = 3;
// SIZE = 4;               // compile error, SIZE is const
int B[SIZE] = {1, 2, 3}; // ok

const int size2 = size;  // 'size2' is dynamic
```

- `int*` → `const int*`
- `const int*` ↛ `int*`

```
void read(const int* array) {} // the values of 'array' cannot be modified

void write(int* array) {}

int*       ptr       = new int;
const int* const_ptr = new int;
read(ptr);           // ok
write(ptr);          // ok
read(const_ptr);     // ok
// write(const_ptr); // compile error
```

- `int*` pointer to `int`
  - The value of the pointer can be modified
  - The elements referred by the pointer can be modified
- `const int*` pointer to `const int`. Read as `(const int)*`
  - The value of the pointer can be modified
  - The elements referred by the pointer cannot be modified
- `int *const` const pointer to `int`
  - The value of the pointer cannot be modified
  - The elements referred by the pointer can be modified
- `const int *const` const pointer to `const int`
  - The value of the pointer cannot be modified
  - The elements referred by the pointer cannot be modified

Note: `const int*` (*West notation*) is equal to `int const*` (*East notation*)
Tip: pointer types should be read from right to left

Common error: adding `const` to a pointer is not the same as adding `const` to a type alias of a pointer

```
using ptr_t       = int*;
using const_ptr_t = const int*;

void f1(const int* ptr) {   // read as '(const int)*'
//  ptr[0] = 0;             // not allowed: pointer to const objects
    ptr    = nullptr;       // allowed
}

void f2(const_ptr_t ptr) {} // same as before

void f3(const ptr_t ptr) {  // warning!! equal to 'int* const'
    ptr[0] = 0;             // allowed!!
//  ptr    = nullptr;       // not allowed: const pointer to modifiable objects
}
```

constexpr (C++11)

`constexpr` specifier declares an expression that **can** be evaluated at compile-time

- `const` guarantees the value of a variable to be fixed during the initialization
- `constexpr` implies `const`
- `constexpr` can improve performance and memory usage
- `constexpr` can potentially impact the compilation time

constexpr Variable

constexpr variables are **always** evaluated at compile-time

```
const int v1 = 3;          // compile-time evaluation
const int v2 = v1 * 2;     // compile-time evaluation

int       a  = 3;          // "a" is dynamic
const int v3 = a;          // run-time evaluation!!

constexpr int c1 = v1;     // ok
// constexpr int c2 = v3; // compile error, "v3" is dynamic
```

constexpr Function

constexpr guarantees compile-time evaluation of a function as long as **all** its arguments are evaluated at compile-time

- C++11: must contain exactly one `return` statement, and it must not contain loops or switch
- C++14: no restrictions

```
constexpr int square(int value) {
    return value * value;
}
square(4); // compile-time evaluation, '4' is a literal
int a = 4; // "a" is dynamic
square(a); // run-time evaluation
```

- cannot contain run-time features such as try-catch blocks, exceptions, and RTTI
- cannot contain `goto` and `asm` statements
- cannot contain `assert()` until C++14
- cannot be a `virtual` member function until C++20
- cannot contain `static` variables until C++23
- undefined behavior code is not allowed, e.g. `reinterpret_cast`, unsafe usage of `union`, signed integer overflow, etc.

It is always *evaluated at run-time* if:

- contain run-time functions, namely non-`constexpr` functions
- contain references to run-time global variables

`constexpr` *non-static member functions* of run-time objects cannot be used at compile-time if they contain data members or non-compile-time functions

`static constexpr` *member functions* don't present this issue because they don't depend on a specific instance

```
struct A {
    int v { 3 };
    constexpr int        f() const { return v; }
    static constexpr int g()       { return 3; }
};
A a1;
// constexpr int x = a1.f(); // compile error, f() is evaluated at run-time
constexpr    int y = a1.g(); // ok, same as 'A::g()'

constexpr    A   a2;
constexpr    int x = a2.f(); // ok
```

`consteval` Keyword

`consteval` (C++20)

`consteval`, or *immediate function*, guarantees compile-time evaluation.

A run-time value always produces a compile error

```
consteval int square(int value) {
    return value * value;
}

square(4);    // compile-time evaluation

int v = 4;    // "v" is dynamic
// square(v); // compile error
```

`constinit` Keyword

`constinit` (C++20)

`constinit` guarantees compile-time initialization of a variable. A run-time initialization value always produces a compile error

- The value of a variable can change during the execution
- `const constinit` does not imply `constexpr`, while the opposite is true

```
constexpr int square(int value) {
    return value * value;
}
constinit int v1 = square(4);     // compile-time evaluation
v1 = 3;                           // ok, v1 can change

int a = 4;                        // "v" is dynamic
// constinit int v2 = square(a); // compile error
```

if constexpr

`if constexpr` C++17 allows to *conditionally* compile code based on a *compile-time* predicate

The `if constexpr` statement forces the compiler to evaluate the branch at compile-time (similarly to the `#if` preprocessor)

```
auto f() {
    if constexpr (sizeof(void*) == 8)
        return "hello";               // const char*
    else
        return 3;                     // int, never compiled
}
```

Note: Ternary (conditional) operator does not provide `constexpr` variant

if constexpr Example

```
constexpr int fib(int n) {
    return (n == 0 || n == 1) ? 1 : fib(n - 1) + fib(n - 2);
}

int main() {
    if constexpr (sizeof(void*) == 8)
        return fib(5);
    else
        return fib(3);
}
```

Generated assembly code (x64 OS):

```
main:
    mov eax, 8
    ret
```

`if constexpr` Pitfalls

`if constexpr` only works with *explicit* `if/else` statements

```
auto f1() {
    if constexpr (my_constexpr_fun() == 1)
        return 1;
//  return 2.0; compile error   // this is not part of constexpr
}
```

`else if` branch requires `constexpr`

```
auto f2() {
    if constexpr (my_constexpr_fun() == 1)
        return 1;
    else if (my_constexpr_fun() == 2) // -> else if constexpr
//      return 2.0; compile error     // this is not part of constexpr
    else
        return 3L;
}
```

std::is_constant_evaluated()

C++20 provides `std::is_constant_evaluated()` utility to evaluate if the current function is evaluated at compile time

```cpp
#include <type_traits> // std::is_constant_evaluated

constexpr int f(int n) {
    if (std::is_constant_evaluated())
       return 0;
    return 4;
}

f(3); // return 0

int v = 3;
f(v); // return = 4
```

`std::is_constant_evaluated()` has two problems that `if consteval` C++23 solves:

(1) Calling a `consteval` function cannot be used within a `constexpr` function if it is called with a run-time parameter

```
consteval int g(int n) { return n * 3; }

constexpr int f(int n) {
    if (std::is_constant_evaluated())  // it works with if consteval
       return g(n);
    return 4;
}

// f(3); compiler error
```

(2) `if constexpr (std::is_constant_evaluated())` is a bug because it is always evaluated to `true`

```
constexpr int f(int x) {
    if constexpr (std::is_constant_evaluated())  // if consteval avoids this error
       return 3;
    return 4;
}

constexpr int g(int x) {
    if consteval {
       return 3;
    }
    return 4;
}
```

`volatile` Keyword ★

`volatile` Keyword

`volatile`

`volatile` is a hint to the compiler to avoid aggressive memory optimizations involving a pointer or an object

Use cases:

- *Low-level programming*: driver development, interaction with assembly, etc. (force writing to a specific memory location)
- *Multi-thread program*: variables shared between threads/processes to communicate (don't optimize, delay variable update)
- *Benchmarking*: some operations need to not be optimized away

Note: `volatile` reads/writes can still be reordered with respect to non-volatile ones

`volatile` Keyword - Example

The following code compiled with `-O3` (full optimization) and without `volatile` works fine

```
volatile int* ptr = new int[1];              // actual alloction size is much
int           pos = 128 * 1024 / sizeof(int); // larger, typically 128 KB
ptr[pos]          = 4;                       // ☠ segfault
```

Explicit Type Conversion

Old style cast: `(type) value`

New style cast:

- `static_cast` performs compile-time (not run-time) type check. This is the safest cast as it prevents accidental/unsafe conversions between types
- `const_cast` can add or cast away (remove) constness or volatility
- `reinterpret_cast`
  `reinterpret_cast<T*>(v)` equal to `(T*) v`
  `reinterpret_cast<T&>(v)` equal to `*((T*) &v)`

`const_cast` and `reinterpret_cast` do not compile to any CPU instruction

Static cast vs. old style cast:

```
char a[] = {1, 2, 3, 4};
int* b   = (int*) a;              // ok
cout << b[0];                     // print 67305985 not 1!!
//int* c = static_cast<int*>(a); // compile error unsafe conversion
```

Const cast:

```
const int       a  = 5;
const_cast<int>(a) = 3; // ok, but undefined behavior
```

Reinterpret cast: (bit-level conversion)

```
float b = 3.0f;
// bit representation of b: 01000000010000000000000000000000
int   c = reinterpret_cast<int&>(b);
// bit representation of c: 01000000010000000000000000000000
```

Print the value of a pointer

```
int*  ptr = new int;
//int x1  = static_cast<size_t>(ptr);      // compile error unsafe
int   x2  = reinterpret_cast<size_t>(ptr); // ok, same size

// but
unsigned v;
//int x3 = reinterpret_cast<int>(v); // compile error
                                     // invalid conversion
```

Array reshaping

```
int a[3][4];
int (&b)[2][6] = reinterpret_cast<int (&)[2][6]>(a);
int (*c)[6]    = reinterpret_cast<int (*)[6]>(a);
```

Pointer Aliasing

One pointer **aliases** another when they both point to the same memory location

Type Punning

Type punning refers to circumvent the type system of a programming language to achieve an effect that would be difficult or impossible to achieve within the bounds of the formal language

The compiler assumes that the ***strict aliasing rule*** *is never violated*: Accessing a value using a type which is different from the original one is not allowed and it is classified as *undefined behavior*

```cpp
// slow without optimizations. The branch breaks the CPU instruction pipeline
float abs(float x) {
    return (x < 0.0f) ? -x : x;
}

// optimized by hand
float abs(float x) {
    unsigned uvalue = reinterpret_cast<unsigned&>(x);
    unsigned tmp    = uvalue & 0x7FFFFFFF; // clear the last bit
    return reinterpret_cast<float&>(tmp);
}
// this is undefined behavior!!
```

GCC warning (not clang): `-Wstrict-aliasing`

- blog.qt.io/blog/2011/06/10/type-punning-and-strict-aliasing
- What is the Strict Aliasing Rule and Why do we care?

memcpy and std::bit_cast

The right way to avoid undefined behavior is using `memcpy`

```
float    v1 = 32.3f;
unsigned v2;
std::memcpy(&v2, &v1, sizeof(float));
// v1, v2 must be trivially copyable
```

C++20 provides `std::bit_cast` safe conversion for replacing `reinterpret_cast`

```
float    v1 = 32.3f;
unsigned v2 = std::bit_cast<unsigned>(v1);
```

`sizeof` Operator

`sizeof` operator

`sizeof`

The `sizeof` is a compile-time operator that determines the size, in bytes, of a variable or data type

- `sizeof` returns a value of type `size_t`
- `sizeof(anything)` never returns 0 (*except for arrays of size 0)
- `sizeof(char)` always returns 1
- When applied to structures, it also takes into account the internal padding
- When applied to a reference, the result is the size of the referenced type
- `sizeof(incomplete type)` produces compile error, e.g. `void`
- `sizeof(bitfield member)` produces compile error

* `gcc` allows array of size 0 (not allowed by the C++ standard)

```
sizeof(int);   // 4 bytes
sizeof(int*)   // 8 bytes on a 64-bit OS
sizeof(void*)  // 8 bytes on a 64-bit OS
sizeof(size_t) // 8 bytes on a 64-bit OS
```

```
int f(int[] array) {       // dangerous!!
    cout << sizeof(array);
}

int  array1[10];
int* array2 = new int[10];
cout << sizeof(array1); // sizeof(int) * 10 = 40 bytes
cout << sizeof(array2); // sizeof(int*) = 8 bytes
f(array1);              // 8 bytes (64-bit OS)
```

```c
struct A {
    int  x; // 4-byte alignment
    char y; // offset 4
};
sizeof(A); // 8 bytes: 4 + 1 (+ 3 padding), must be aligned to its largest member

struct B {
    int   x; // offset 0 -> 4-byte alignment
    char  y; // offset 4 -> 1-byte alignment
    short z; // offset 6 -> 2-byte alignment
};
sizeof(B); // 8 bytes : 4 + 1 (+ 1 padding) + 2

struct C {
    short z; // offset 0 -> 2-byte alignment
    int   x; // offset 4 -> 4-byte alignment
    char  y; // offset 8 -> 1-byte alignment
};
sizeof(C); // 12 bytes : 2 (+ 2 padding) + 4 + 1 + (+ 3 padding)
```

```
char  a;
char& b = a;
sizeof(&a);      // 8 bytes in a 64-bit OS (pointer)
sizeof(b);       // 1 byte, equal to sizeof(char)
                 // NOTE: a reference is not a pointer
struct S1 {
    void* p;
};
sizeof(S1);      // 8 bytes

struct S2 {
    char& c;
};
sizeof(S2);      // 8 bytes, same as sizeof(void*)
sizeof(S2{}.c); // 1 byte
```

```
struct A {};
sizeof(A);      // 1 : sizeof never return 0

A array1[10];
sizeof(array1); // 1 : array of empty structures

int array2[0];  // only gcc, compiler error for other compiler
sizeof(array2); // 0 : special case
```

C++20 `[[no_unique_address]]` allows a structure member to be overlapped with other data members of a different type

```
struct Empty {}; // empty class, sizeof(Empty) == 1

struct A {       // sizeof(A) == 5 (4 + 1)
    int   i;
    Empty e;
};

struct B {       // sizeof(B) == 4, 'e' overlaps with 'i'
    int i;
    [[no_unique_address]] Empty e;
};
```

Notes: `[[no_unique_address]]` is ignored by MSVC even in C++20 mode; instead, `[[msvc::no_unique_address]]` is provided

`sizeof` and Size of a Byte

Interesting: C++ does not explicitly define the size of a byte (see `Exotic architectures the standards committees care about`)

Modern C++ Programming

6. Basic Concepts V

Functions and Preprocessing

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

4 Preprocessing

Functions

Overview

A **function** (**procedure** or **routine**) is a piece of code that performs a *specific task*

Purpose:

- **Avoiding code duplication**: less code for the same functionality $\rightarrow$ less bugs
- **Readability**: better express what the code does
- **Organization**: break the code in separate modules

Function Parameter and Argument

Function Parameter [formal]

A **parameter** is the variable which is part of the method signature

Function Argument [actual]

An **argument** is the actual value (instance) of the variable that gets passed to the function

```
void f(int a, char* b); // parameters: int a, char* b
                        // return type: void

f(3, "abc");            // arguments: 3, "abc"
```

Pass by-Value

Call-by-value

The object is copied and assigned to input arguments of the method `f(T x)`

Advantages:

- Changes made to the parameter inside the function have no effect on the argument

Disadvantages:

- Performance penalty if the copied arguments are large (e.g. a structure with several data members)

When to use:

- Built-in data type and small objects ($\leq$ 8 bytes)

When not to use:

- Fixed size arrays which decay into pointers
- Large objects

Pass by-Pointer

Call-by-pointer

The address of a variable is copied and assigned to input arguments of the method `f(T* x)`

Advantages:

- Allows a function to change the value of the argument
- The argument is not copied (fast)

Disadvantages:

- The argument may be a null pointer
- Dereferencing a pointer is slower than accessing a value directly

When to use:

- *Raw* arrays (use `const T*` if read-only)

When not to use:

- All other cases

Pass by-Reference

Call-by-reference

The reference of a variable is copied and assigned to input arguments of the method `f(T& x)`

Advantages:

- Allows a function to change the value of the argument (better readability compared with pointers)
- The argument is not copied (fast)
- References must be initialized (no null pointer)
- Avoid implicit conversion (without `const T&`)

When to use:

- All cases except raw pointers

When not to use:

- Pass by-value *could* give performance advantages and improve the readability with built-in data type and small objects that are trivially copyable

Examples

```
struct MyStruct;

void f1(int a);         // pass by-value
void f2(int& a);        // pass by-reference
void f3(const int& a); // pass by-const reference
void f4(MyStruct& a);  // pass by-reference

void f5(int* a);        // pass by-pointer
void f6(const int* a); // pass by-const pointer
void f7(MyStruct* a);  // pass by-pointer

void f8(int*& a);       // pass a pointer by-reference
//-------------------------------------------------------------
char c = 'a';
f1(c);    // ok, pass by-value (implicit conversion)
// f2(c); // compile error different types
f3(c);    // ok, pass by-value (implicit conversion)
```

Signature

Function signature defines the *input types* for a (specialized) function and the *inputs + outputs types* for a template function

A function signature includes the number of arguments, the types of arguments, and the order of the arguments

- The C++ standard prohibits a function declaration that only differs in the return type
- Function declarations with different signatures can have distinct return types

Overloading

Function overloading allows having distinct functions with the same name but with different *signatures*

```
void f(int a, char* b);          // signature:  (int, char*)

// char f(int a, char* b);       // compile error same signature
                                 // but different return types

void f(const int a, char* b);    // same signature, ok
                                 // const int == int

void f(int a, const char* b);    // overloading with signature: (int, const char*)

int f(float);                    // overloading with signature: (float)
                                 // the return type is different
```

Overloading Resolution Rules

- An exact match
- A promotion (e.g. `char` to `int`)
- A standard type conversion (e.g. `float` and `int`)
- A constructor or user-defined type conversion $\rightsquigarrow$

```
void f(int   a);
void f(float b);          // overload
void f(float b, char c); // overload
//-------------------------------------------------------------
   f(0);         // exact match
   f('a');       // promotion from char to int (promotion)
// f(3LL);       // compile error ambiguous match
   f(2.3f);      // exact match
// f(2.3);       // compile error ambiguous match
   f(2.3, 'a'); // standard type conversion, ambiguity is not possible here
```

Overloading and `=delete`

`=delete` can be used to prevent calling the wrong overload

```
void g(int) {}

void g(double) = delete;

g(3);   // ok
g(3.0); // compile error
```

```
#include <cstddef> // std::nullptr_t

void f(int*) {}

void f(std::nullptr_t) = delete;

f(nullptr); // compile error
```

Function Default Parameters

Default/Optional parameter

A **default parameter** is a function parameter that has a default value

- If the user does not supply a value for this parameter, the default value will be used
- All default parameters must be the rightmost parameters
- Default parameters must be declared only once
- Default parameters can improve compile time and avoid redundant code because they avoid defining other overloaded functions

```
void f(int a, int b = 20);          // declaration

//void f(int a, int b = 10) { ... } // compile error, already set in the declaration

void f(int a, int b) { ... }        // definition, default value of "b" is already set

f(5);  // b is 20
```

Function Attributes `[[attribute]]` 1/2

C++ allows marking functions with standard properties to better express their intent:

- C++11 `[[noreturn]]` indicates that a function does not return for optimization purposes or compiler warnings
- C++14 `[[deprecated]]`, `[[deprecated("reason")]]` indicates the use of a function is discouraged. It issues a compiler warning if used
- C++17 `[[nodiscard]]`
  C++20 `[[nodiscard("reason")]]` issues a warning if the return value of a function is discarded (not handled)

```
[[deprecated]] void my_rand() { ... }

[[deprecated("please use rnd()")]] void my_rand2() { ... }

[[nodiscard]] int f() { return 3; }

[[noreturn]] void g() { std::exit(0); }

my_rand();   // WARNING: "my_rand() is deprecated"
my_rand2();  // WARNING: "my_rand2() is deprecated, please use rnd()"
f();         // WARNING "discard return value"
int z = f(); // no warning
g();         // no code after calling this function
```

Function Pointers and Function Objects

Standard C achieves generic programming capabilities and composability through the concept of **function pointer**

A function can be passed as a pointer to another function and behaves as an *"indirect call"*

```c
#include <stdlib.h> // qsort

int descending(const void* a, const void* b) {
    return *((const int*) a) > *((const int*) b);
}

int array[] = {7, 2, 5, 1};
qsort(array, 4, sizeof(int), descending);
// array: { 7, 5, 2, 1 }
```

```
int eval(int a, int b, int (*f)(int, int)) {
    return f(a, b);
}
// type: int (*)(int, int)
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

cout << eval(4, 3, add); // print 7
cout << eval(4, 3, sub); // print 1
```

Problems:

Safety There is no check of the argument type in the generic case (e.g. `qsort`)

Performance Any operation requires an indirect call to the original function. Function inlining is not possible

Function Object

A **function object**, or **functor**, is a *callable* object that can be treated as a parameter

C++ provides a more efficient and convenience way to pass *"procedure"* to other functions called **function object**

```
#include <algorithm> // for std::sort

struct Descending {                    // <-- function object
    bool operator()(int a, int b) {  // function call operator
        return a > b;
    }
};
int array[] = {7, 2, 5, 1};
std::sort(array, array + 4, Descending{});
// array: { 7, 5, 2, 1 }
```

Advantages:

Safety Argument type checking is always possible. It could involve templates

Performance The compiler injects `operator()` in the code of the destination function and then compile the routine. Operator inlining is the standard behavior

C++11 simplifies the concept by providing less verbose `function objects` called **lambda expressions**

Lambda Expressions

Lambda Expression

A C++11 **lambda expression** is an *inline local-scope* function object

```
auto x = [capture clause] (parameters) { body }
```

- The `[capture clause]` marks the declaration of the lambda and how the local scope arguments are captured (by-value, by-reference, etc.)
- The `parameters` of the lambda are normal function parameters (optional)
- The `body` of the lambda is a normal function body

The expression to the right of `=` is the **lambda expression**, and *the runtime object* `x` created by that expression is the **closure**

Lambda Expression

```cpp
#include <algorithm> // for std::sort

int array[] = {7, 2, 5, 1};
auto lambda = [](int a, int b){ return a > b; }; // named lambda

std::sort(array, array + 4, lambda);
// array: { 7, 5, 2, 1 }

// in alternative, in one line of code:          // unnamed lambda
std::sort(array, array + 4, [](int a, int b){ return a > b; });
// array: { 7, 5, 2, 1 }
```

Capture List

Lambda expressions *capture* external variables used in the body of the lambda in two ways:

- Capture *by-value*
- Capture *by-reference* (can modify external variable values)

Capture list can be passed as follows

- `[]` no capture
- `[=]` captures all variables *by-value*
- `[&]` captures all variables *by-reference*
- `[var1]` captures only `var1` *by-value*
- `[&var2]` captures only `var2` *by-reference*
- `[var1, &var2]` captures `var1` *by-value* and `var2` *by-reference*

Capture List Examples

```cpp
// GOAL: find the first element greater than "limit"
#include <algorithm> // for std::find_if
int limit = ...

auto lambda1 = [=](int value)      { return value > limit; }; // by-value
auto lambda2 = [&](int value)      { return value > limit; }; // by-reference
auto lambda3 = [limit](int value)  { return value > limit; }; // "limit" by-value
auto lambda4 = [&limit](int value) { return value > limit; }; // "limit" by-reference
// auto lambda5 = [](int value)    { return value > limit; }; // no capture
                                                              // compile error

int array[] = {7, 2, 5, 1};
std::find_if(array, array + 4, lambda1);
```

Capture List - Other Cases

- `[=, &var1]` captures all variables used in the body of the lambda **by-value**, except `var1` that is captured **by-reference**
- `[&, var1]` captures all variables used in the body of the lambda **by-reference**, except `var1` that is captured **by-value**
- A lambda expression can read a variable without capturing it if the variable is `constexpr`

```
constexpr int limit = 5;
int var1 = 3, var2 = 4;

auto lambda1 = [](int value){ return value > limit; };

auto lambda2 = [=, &var2]() { return var1 > var2;   };
```

Lambda Expressions - Parameters

C++14 Lambda expression parameters can be automatically deduced

```
auto x = [](auto value) { return value + 4; };
```

C++14 Lambda expression parameters can be initialized

```
auto x = [](int i = 6) { return i + 4; };
```

Lambda Expressions - Composability

Lambda expressions can be composed

```
auto lambda1 = [](int value){ return value + 4; };
auto lambda2 = [](int value){ return value * 2; };

auto lambda3 = [&](int value){ return lambda2(lambda1(value)); };
// returns (value + 4) * 2
```

A function can return a lambda (dynamic dispatch is also possible)

```
auto f() {
    return [](int value){ return value + 4; };
}

auto lambda = f();
cout << lambda(2); // print "6"
```

constexpr/consteval Lambda Expression

C++17 Lambda expression supports `constexpr`

C++20 Lambda expression supports `consteval`

```cpp
// constexpr lambda
auto factorial = [](int value) constexpr {
    int ret = 1;
    for (int i = 2; i <= value; i++)
        ret *= i;
    return ret;
};
auto mul = [](int v) consteval { return v * 2; };

constexpr int v1 = factorial(4) + mul(5); // '24' + '10'
```

`template` Lambda Expression ⇝

C++20 Lambda expression supports `template` and `requires` clause

```
auto lambda = []<typename T>(T value)
              requires std::is_arithmetic_v<T> {
    return value * 2;
};

auto v = lambda(3.4); // v: 6.8 (double)

struct A{} a;
// auto v = lambda(a); // compiler error
```

`mutable` Lambda Expression

Lambda capture is *by-const-value*

`mutable` specifier allows the lambda to modify the parameters captured *by-value*

```
int  var = 1;

auto lambda1 = [&](){ var = 4; };          // ok
lambda1();
cout << var; // print '4'

// auto lambda2 = [=](){ var = 3; };       // compile error
// lambda operator() is const

auto lambda3 = [=]() mutable { var = 3; }; // ok
lambda3();
cout << var;  // print '4', lambda3 captures by-value
```

[[nodiscard]] Attribute

C++23 allows adding the `[[nodiscard]]` attribute to lambda expressions

```
auto lambda = [] [[nodiscard]] (){ return 4; };

lambda();          // compiler warning

auto x = lambda(); // ok
```

Capture List and Classes ⇝

- `[this]` captures the current object `(*this)` *by-reference* (implicit in C++17)
- `[x = x]` captures the current object member `x` *by-value* C++14
- `[&x = x]` captures the current object member `x` *by-reference* C++14
- `[=]` default capture of `this` pointer by value has been deprecated C++20

```
class A {
    int data = 1;

    void f() {
        int  var     = 2;                                  // <-- local variable
        auto lambda1 = [=]()     { return var;  };         // copy by-value, return 2
        auto lambda2 = [=]()     { int var = 3; return var; }; // return 3 (nearest scope)
        auto lambda3 = [this]()  { return data; };         // copy by-reference, return 1
        auto lambda4 = [*this]() { return data; };         // copy by-value (C++17), return 1
//      auto lambda5 = [data]()  { return data; };         // compile error 'data' is not visible
        auto lambda6 = [data = data]() { return data; }; // return 1
    }
};
```

Preprocessing

Preprocessing and Macro

A **preprocessor directive** is any line preceded by a *hash* symbol (#) which tells the compiler how to interpret the source code *before* compiling it

Macro are preprocessor directives which substitute any occurrence of an *identifier* in the rest of the code by replacement

Macro are evil:

Do not use macro expansion!!

...or use as little as possible

- Macro cannot be directly debugged
- Macro expansions can have unexpected side effects
- Macro have no namespace or scope

All statements starting with `#`

- `#include "my_file.h"`
  Inject the code in the current file
- `#define MACRO <expression>`
  Define a new macro
- `#undef MACRO`
  Undefine a macro
  (a macro should be undefined as early as possible for safety reasons)

Multi-line Preprocessing: `\` at the end of the line

Indent: `#  define`

Conditional Compiling

- `#if <condition>`
  `code`
  `#elif <condition>`
  `code`
  `#else`
  `code`
  `#endif`
- `#if defined(MACRO)` equal to `#ifdef MACRO`
  `#elif defined(MACRO)` equal to `#elifdef MACRO` C++23
  Check if a macro is defined
- `#if !defined(MACRO)` equal to `#ifndef MACRO`
  `#elif !defined(MACRO)` equal to `#elifndef MACRO` C++23
  Check if a macro is not defined

Common Error 1

Define macros in header files and before includes!!

```
#include <iostream>

#define value    // very dangerous!!
#include "big_lib.hpp"

int main() {
    std::cout << f(4); // should print 7, but it always prints 3
}
```

big_lib.hpp:

```
int f(int value) {    // 'value' disappears
    return value + 3;
}
```

It is very hard to see this problem when the macro is in a header

Common Error 2

`#if defined` can introduce bugs related to macro visibility

```
// #include "macro_definition.hpp" // forget to add the header that defines ENABLE_DEBUG

#if defined(ENABLE_DEBUG)
    void f(int v) { cout << v << endl; return v * 3; }
#else
    void f(int v) { return v * 3; }
#endif
```

```
#if ENABLE_DEBUG // evaluated to 0 or 1
    void f(int v) { cout << v << endl; return v * 3; }
#else
    void f(int v) { return v * 3; }
#endif
```

Common Error 3

Forget to use parenthesis in macro definitions!!

```cpp
#include <iostream>

#define SUB1(a, b)  a - b       // WRONG
#define SUB2(a, b) (a - b)      // WRONG
#define SUB3(a, b) ((a) - (b))  // correct

int main() {
    std::cout << (5 * SUB1(2, 1));   // print 9 not 5!!
    std::cout << SUB2(3 + 3, 2 + 2); // print 6 not 2!!
    std::cout << SUB3(3 + 3, 2 + 2); // print 2
}
```

Common Error 4

Macros make hard to find compile errors!!

```
 1: #include <iostream>
 2:
 3: #define F(a) {        \
 4:         ...            \
 5:         ...            \
 6:         return v;
 7:
 8: int main() {
 9:     F(3);      // compile error at line 9!!
10: }
```

- **In which line is the error??!***

*modern compilers are able to roll out the macro

Common Error 5

Macro can introduce bugs related to the evaluation of their expressions!!

```
#if defined(DEBUG)
#   define CHECK(EXPR)   // do something with EXPR
    void check(bool b) { /* do something with b */ }
#else
#   define CHECK(EXPR)   // do nothing
    void check(bool) {}  // do nothing
#endif
bool f() { /* return a boolean value */ }

check( f() )
CHECK( f() ) // <-- problem here
```

- **What happens when `DEBUG` is not defined?**

  `f()` is not evaluated by using the macro

Common Error 6

Forget curly brackets in multi-lines macros!!

```
#include <iostream>
#include <nuclear_explosion.hpp>

#define NUCLEAR_EXPLOSION                    \  // {
    std::cout << "start nuclear explosion"; \
    nuclear_explosion();
                                                // }
int main() {
    bool never_happen = false;
    if (never_happen)
        NUCLEAR_EXPLOSION
} // BOOM!! ☠
```

The second line is executed!!

Common Error 7

Macros do not have scope!!

```cpp
#include <iostream>

void f() {
    #define value 4
    std::cout << value;
}

int main() {
    f();                  // 4
    std::cout << value;   // 4
    #define value 3
    f();                  // 4
    std::cout << value;   // 3
}
```

Common Error 8

Macros can have side effect!!

```c
#define MIN(a, b) ((a) < (b) ? (a) : (b))

int main() {
    int array1[] = { 1, 5, 2 };
    int array2[] = { 6, 3, 4 };
    int i = 0;
    int j = 0;
    int v1 = MIN(array1[i++], array2[j++]); // v1 = 5!!
    int v2 = MIN(array1[i++], array2[j++]); // undefined behavior/
}                                           // segmentation fault
```

Common Error 9

Macros can have undefined behavior themselves!!

```
#define MY_MACRO defined(EXTERNAL_MACRO)

#if MY_MACRO
#   define MY_VALUE 1
#else
#   define MY_VALUE 0
#endif

int f() { return MY_VALUE; } // undefined behavior
```

When Preprocessors are Necessary

- **Conditional compiling**: different architectures, compiler features, etc.
- **Mixing different languages**: code generation (example: `asm assembly`)
- **Complex name replacing**: see template programming

Otherwise, prefer `const` and `constexpr` for constant values and functions

```
#define SIZE 3      // replaced with
const int SIZE = 3; // only C++11 at global scope

#define SUB(a, b) ((a) - (b)) // replaced with
constexpr int sub(int a, int b) {
    return a - b;
}
```

`__LINE__` Integer value representing the current line in the source code file being compiled

`__FILE__` A string literal containing the name of the source file being compiled

`__FUNCTION__` (non-standard, `gcc`, `clang`) A string literal containing the name of the function in the 'macro scope'

`__PRETTY_FUNCTION__` (non-standard, `gcc`, `clang`) A string literal containing the full signature of the function in the 'macro scope'

`__func__` (C++11 keyword) A string containing the name of the function in the 'macro scope'

source.cpp:

```cpp
#include <iostream>

void f(int p) {
    std::cout << __FILE__ << ":" << __LINE__;  // print 'source.cpp:4'
    std::cout << __FUNCTION__;                 // print 'f'
    std::cout << __func__;                     // print 'f'
}

// see template lectures
template<typename T>
float g(T p) {
    std::cout << __PRETTY_FUNCTION__;          // print 'float g(T) [T = int]'
    return 0.0f;
}

void g1() { g(3); }
```

C++20 provides source location utilities for replacing macro-based approach

`#include <source_location>`

`current()`	get source location info (`static member`)
`line()`	source code line
`column()`	line column
`file_name()`	current file name
`function_name()`	current function name

```cpp
#include <source_location>

void f(std::source_location s = std::source_location::current()) {
    cout << "function: " << s.function_name() << ", line " << s.line();
}
f(); // print: "function: f, line 6"
```

Select code depending on the C/C++ version

- `#if defined(__cplusplus)` C++ code
- `#if __cplusplus == 199711L` ISO C++ 1998/2003
- `#if __cplusplus == 201103L` ISO C++ 2011*
- `#if __cplusplus == 201402L` ISO C++ 2014*
- `#if __cplusplus == 201703L` ISO C++ 2017

Select code depending on the compiler

- `#if defined(__GNUG__)` The compiler is gcc/g++ †
- `#if defined(__clang__)` The compiler is clang/clang++
- `#if defined(_MSC_VER)` The compiler is Microsoft Visual C++

---

* MSVC defines __cplusplus == 199711L even for C++11/14

† __GNUC__ is defined by many compilers, e.g clang

Select code depending on the operating system or environment

- `#if defined(_WIN64)` OS is `Windows 64-bit`
- `#if defined(__linux__)` OS is `Linux`
- `#if defined(__APPLE__)` OS is `Mac OS`
- `#if defined(__MINGW32__)` OS is `MinGW 32-bit`
- ...and many others

Other Macros

`__DATE__` A string literal in the form "MMM DD YYYY" containing the date in which the compilation process began

`__TIME__` A string literal in the form "hh:mm:ss" containing the time at which the compilation process began

Very comprehensive macro list:

- `sourceforge.net/p/predef/wiki/Home/`
- `Compiler predefined macros`
- `Abseil platform macros`

Feature Testing Macro

C++17 introduces `__has_include` macro which returns `1` if header or source file with the specified name exists

```
#if __has_include(<iostream>)
#   include <iostream>
#endif
```

C++20 introduces a set of macros to evaluate if a given feature is supported by the compiler

```
#if __cpp_constexpr
constexpr int square(int x) { return x * x; }
#endif
```

Common Error 10 ⇝

Macros depend on compilers and environment!!

```cpp
struct A {
    int x;  // enable C++11 code
#if __cplusplus >= 201103
    A() = default;
#else
    A() {}
#endif
};
```

```cpp
// should return ≈ 10.0f
float safe_function() {
    A a{}; // zero-initialization
    for (int i = 0; i < 10; i++)
        a.x += 1.0f;
    return a.x;
}
// what is the behavior ???
```

The code works fine on Linux, but not under Windows MSVC. MSVC sets `__cplusplus` to `199711` even if C++11/14/17 flag is set!! in this case the code can return `NaN`

---

see Lecture "Object-Oriented Programming II - Zero Initialization" and `MSVC now correctly reports __cplusplus`

Stringizing Operator (#)

The **stringizing macro operator** (`#`) causes the corresponding actual argument to be enclosed in double quotation marks `"`

```
#define STRING_MACRO(string) #string

cout << STRING_MACRO(hello);  // equivalent to "hello"
```

```
#define INFO_MACRO(my_func)                          \
{                                                    \
    my_func                                          \
    cout << "call " << #my_func << " at "            \
         << __FILE__ << ":" __LINE__;                \
}
void g(int) {}

INFO_MACRO( g(3) ) // print: "call g(3) at my_file.cpp:7"
```

Common Error 11

Code injection

```cpp
#include <cstdio>

#define CHECK_ERROR(condition)                                  \
{                                                               \
   if (condition) {                                             \
      std::printf("expr: " #condition " failed at line %d\n",\
                  __LINE__);                                    \
   }                                                            \
}

int t = 6, s = 3;
CHECK_ERROR(t > s) // print "expr: t > s failed at line 13"
CHECK_ERROR(t % s == 0) // segmentation fault!!! ☠
// printf interprets "% s" as a format specifier
```

`#error` and `#warning`

- `#error "text"` The directive emits a user-specified error message at compile time when the compiler parse it and stop the compilation process

- C++23 `#warning "text"` The directive emits a user-specified warning message at compile time when the compiler parse it without stopping the compilation process

#pragma

The `#pragma` directive controls implementation-specific behavior of the compiler. In general, it is not portable

- `#pragma message "text"` Display informational messages at compile time (every time this instruction is parsed)
- `#pragma GCC diagnostic warning "-Wformat"` Disable a GCC warning
- `_Pragma(<command>)` (C++11)
  It is a keyword and can be embedded in a `#define`

```
#define MY_MESSAGE \
    _Pragma("message(\"hello\")")
```

Token-Pasting Operator (##) ★

The **token-concatenation (or pasting) macro operator** (`##`) allows combining two tokens (without leaving no blank spaces)

```
#define FUNC_GEN_A(tokenA, tokenB)  \
    void tokenA##tokenB() {}

#define FUNC_GEN_B(tokenA, tokenB)  \
    void tokenA##_##tokenB() {}

FUNC_GEN_A(my, function)
FUNC_GEN_B(my, function)

myfunction();  // ok, from FUNC_GEN_A
my_function(); // ok, from FUNC_GEN_B
```

Variadic Macro ★

A **variadic macro** C++11 is a special macro accepting a variable number of arguments (separated by comma)

Each occurrence of the special identifier `__VA_ARGS__` in the macro replacement list is replaced by the passed arguments

Example:

```
void f(int a)                { printf("%d", a);             }
void f(int a, int b)         { printf("%d %d", a, b);       }
void f(int a, int b, int c) { printf("%d %d %d", a, b, c); }

#define PRINT(...) \
    f(__VA_ARGS__);

PRINT(1, 2)
PRINT(1, 2, 3)
```

Macro Trick ★

Convert a number literal to a string literal

```
#define TO_LITERAL_AUX(x) #x
#define TO_LITERAL(x)     TO_LITERAL_AUX(x)
```

Motivation: avoid integer to string conversion (performance)

```
int main() {
   int  x1   = 3 * 10;
   int  y1   = __LINE__ + 4;
   char x2[] = TO_LITERAL(3);
   char y2[] = TO_LITERAL(__LINE__);
}
```

Modern C++ Programming

7. Object-Oriented Programming I

Class Concepts

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

C++ Classes

C++ Classes

C/C++ Structure

A **structure** (`struct`) is a collection of variables of the same or different data types under a single name

C++ Class

A **class** (`class`) extends the concept of structure to hold functions as members

`struct` vs. `class`

Structures and *classes* are *semantically* equivalent.

- `struct` represents *passive* objects, namely the *physical state* (set of data)
- `class` represents *active* objects, namely the *logical state* (data abstraction)

Class Members - Data and Function Members

Data Member

Data within a class are called **data members** or **class fields**

Function Member

Functions within a class are called **function members** or **methods**

Holding a resource is a class invariant, and is tied to object lifetime

RAII Idiom consists in three steps:

- Encapsulate a resource into a class (*constructor*)
- Use the resource via a local instance of the class
- The resource is automatically released when the object gets out of scope (*destructor*)

Implication 1: C++ programming language does not require the garbage collector!!

Implication 2 :The programmer has the responsibility to manage the resources

`struct/class` Declaration and Definition

`struct` declaration and definition

```
struct A;      // struct declaration

struct A {     // struct definition
    int x;     // data member
    void f(); // function member
};
```

`class` declaration and definition

```
class A;       // class declaration

class A {      // class definition
    int x;     // data member
    void f(); // function member
};
```

struct/class Function Declaration and Definition

```
struct A {
   void g();          // function member declaration

   void f() {         // function member declaration
       cout << "f"; // inline definition
   }
};

void A::g() {         // function member definition
    cout << "g";     // out-of-line definition
}
```

struct/class Members

```
struct B {
    void g() { cout << "g"; } // function member
};

struct A {
    int  x;                   // data member
    B    b;                   // data member
    void f() { cout << "f"; } // function member
};

A a;
a.x;
a.f();
a.b.g();
```

Class Hierarchy

Child/Derived Class or Subclass

A new class that inheriting variables and functions from another class is called a **derived** or **child** class

Parent/Base Class

The *closest* class providing variables and functions of a derived class is called **parent** or **base** class

Extend a *base class* refers to creating a new class which retains characteristics of the base class and *on top it can add* (and never remove) its own members

Syntax:

```
class DerivedClass : [<inheritance attribute>] BaseClass {
```

```
struct A {          // base class
    int value = 3;

    void g() {}
};

struct B : A {      // B is a derived class of A (B extends A)
    int data = 4;   // B inherits from A

    int f() { return data; }
};

A a;
B b;
a.value;
b.g();
```

```
struct A {};
struct B : A {};

void f(A a) {}      // copy
void g(B b) {}      // copy

void f_ref(A& a) {} // the same for A*
void g_ref(B& b) {} // the same for B*

A a;
B b;
f(a); // ok, also f(b), f_ref(a), g_ref(b)
g(b); // ok, also g_ref(b), but not g(a), g_ref(a)

A a1 = b;    // ok, also  A& a2 = b
// B b1 = a; // compile error
```

Access specifiers

The **access specifiers** define the visibility of inherited members of the subsequent base class. The keywords `public`, `private`, and `protected` specify the sections of visibility

The goal of the *access specifiers* is to prevent direct access to the internal representation of the class for avoiding wrong usage and potential inconsistency (access control)

- **`public:`** No restriction (*function members*, *derived classes*, *outside the class*)
- **`protected:`** *Function members* and *derived classes* access
- **`private:`** *Function members* only access (internal)

`struct` has *default* `public` members
`class` has *default* `private` members

Access specifiers 2/2

```
struct A1 {
    int value;   // public (by default)
protected:
    void f1() {} // protected
private:
    void f2() {} // private
};

class A2 {
    int data;    // private (by default)
};
struct B : A1 {
   void h1() { f1(); } // ok, "f1" is visible in B
// void h2() { f2(); } // compile error "f2" is private in A1
};

A1 a;
a.value;  // ok
// a.f1() // compile error protected
// a.f2() // compile error private
```

The **access specifiers** are also used for defining how the visibility is propagated from the *base class* to a *specific derived class* in the inheritance

Member declaration		Inheritance		Derived classes
`public`				`public`
`protected`	→	**`public`**	→	`protected`
`private`				\
`public`				`protected`
`protected`	→	**`protected`**	→	`protected`
`private`				\
`public`				`private`
`protected`	→	**`private`**	→	`private`
`private`				\

```
struct A {
    int var1; // public
protected:
    int var2; // protected
};

struct B : protected A {
    int var3; // public
};

B b;
// b.var1; // compile error, var1 is protected in B
// b.var2; // compile error, var2 is protected in B
b.var3;    // ok, var3 is public in B
```

```
class A {
public:
    int var1;
protected:
    int var2;
};

class B1 : A {};         // private inheritance

class B2 : public A {}; // public inheritance

B1 b1;
// b1.var1; // compile error, var1 is private in B1
// b1.var2; // compile error, var2 is private in B1

B2 b2;
b2.var1;    // ok, var1 is public in B2
```

When Use `public/protected/private/` for Data Members?

When use `protected/private` data members:

- They are not part of the interface, namely the *logical state* of the object (not useful for the user)
- They must preserve the `const` correctness (e.g. for pointer), see `Advanced Concepts I`

When use `public` data members:

- They can potentially change any time
- `const` correctness is preserved for values and references, as opposite to pointers. *Data members* should be preferred to *member functions* in this case

Class Constructor

Class Constructor

Constructor [ctor]

A **constructor** is a *special* member function of a class that is executed when a new instance of that class is created

Goals: *initialization* and *resource acquisition*

Syntax: `T(...)` same named of the class and no return type

- A *constructor* is supposed to initialize *all* data members
- We can define *multiple constructors* with different signatures
- Any *constructor* can be `constexpr`

Default Constructor

The **default constructor** `T()` is a constructor with no argument

Every class has always either an *implicit*, *explicit*, or *deleted* default constructor

```
struct A {
    A()    {} // explicit default constructor
    A(int) {} // user-defined (non-default) constructor
};
```

```
struct A {
    int x = 3; // implicit default constructor
};
A a{}; // call the default constructor, equivalent to: A a;
```

Note: an *implicit* default constructor is `constexpr`

Default Constructor Examples

```
struct A {
    A() { cout << "A"; }  // default constructor
};

A  a1;              // call the default constructor
// A a2();          // interpreted as a function declaration!!
A a3{};             // ok, call the default constructor
                    // direct-list initialization (C++11)

A  array[3];        // print "AAA"

A* ptr = new A[4]; // print "AAAA"
```

The *implicit* default constructor of a class is marked as **deleted** if (simplified):

- It has any user-defined constructor

```
struct A {
    A(int x) {}
};
// A a; // compile error
```

- It has a non-static member/base class of reference/`const` type

```
struct NoDefault { // deleted default constructor
    int&       x;
    const int y;
};
```

- It has a non-static member/base class which has a deleted (or inaccessible) default constructor

```
struct A {
    NoDefault var;       // deleted default constructor
};
struct B : NoDefault {}; // deleted default constructor
```

- It has a non-static member/base class with a deleted or inaccessible destructor

```
struct A {
private:
    ~A() {}
};
```

Initializer List

The **Initializer list** is used for *initializing the data members* of a class or explicitly call the base class constructor before entering the constructor body

(Not to be confused with `std::initializer_list`)

```
struct A {
    int x, y;

    A(int x1) : x(x1) {}  // ": x(x1)" is the Initializer list
                          // direct initialization syntax

    A(int x1, int y1) :   // ": x{x1}, y{y1}"
        x{x1},            // is the Initializer list
        y{y1} {}          // direct-list initialization syntax
};                        // (C++11)
```

In-Class Member Initializer

C++11 **In-class non-static data members initialization** (NSDMI) allows initializing the data members where they are declared. A user-defined constructor can be used to override their default values

```
struct A {
    int         x   = 0;       // in-class member initializer
    const char* str = nullptr; // in-class member initializer

    A() {} // "x" and "str" are well-defined if
           // the default constructor is called

    A(const char* str1) : str{str1} {}
};
```

Data Member Initialization

const and **reference** data members must be initialized by using the *initialization list* or by using in-class *brace-or-equal-initializer* syntax (C++11)

```
struct A {
    int         x;
    const char y;      // must be initialized
    int&        z;     // must be initialized

    int&        v = x; // equal-initializer (C++11)
    const int  w{4};   // brace initializer (C++11)

    A() : x(3), y('a'), z(x) {}
};
```

Initialization Order

Class member initialization follows the order of declarations and *not* the order in the initialization list

```
struct ArrayWrapper {
    int* array;
    int  size;

    ArrayWrapper(int user_size) :
        size{user_size},
        array{new int[size]} {}
        // wrong!!: "size" is still undefined
};

ArrayWrapper a(10);
cout << a.array[4]; // segmentation fault
```

Uniform Initialization for Objects

Uniform Initialization (C++11)

Uniform Initialization $\{\}$, also called *list-initialization*, is a way to fully initialize any object independently of its data type

- **Minimizing Redundant Typenames**
  - In function arguments
  - In function returns
- Solving the **"Most Vexing Parse" problem**
  - Constructor interpreted as function prototype

mbevin.wordpress.com/2012/11/16/uniform-initialization

Minimizing Redundant Typenames

```
struct Point {
    int x, y;
    Point(int x1, int y1) : x(x1), y(y1) {}
};
```

C++03

```
Point add(Point a, Point b) {
    return Point(a.x + b.x, a.y + b.y);
}
Point c = add(Point(1, 2), Point(3, 4));
```

C++11

```
Point add(Point a, Point b) {
    return { a.x + b.x, a.y + b.y }; // here
}
auto c = add({1, 2}, {3, 4});        // here
```

```
struct A {
   A(int) {}
};

struct B {
// A a(1); // compile error It works in a function scope
   A a{2}; // ok, call the constructor
};
```

```
struct A {};

struct B {
    B(A a) {}
    void f() {}
};

B b( A() );  // "b" is interpreted as function declaration
             //  with a single argument A (*)() (func. pointer)
// b.f()     // compile error "Most Vexing Parse" problem
             // solved with B b{ A{} };
```

Constructors and Inheritance

Class constructors are never inherited

A *Derived* class must call *implicitly* or *explicitly* a *Base* constructor before the current class constructor

Class constructors are called in order from the top Base class to the most Derived class (C++ objects are constructed like onions)

```
struct A {
    A() { cout << "A" };
};
struct B1 : A { // call "A()" implicitly
    int y = 3;  // then, "y = 3"
};
struct B2 : A { // call "A()" explicitly
    B2() : A() { cout << "B"; }
};
B1 b1; // print "A"
B2 b2; // print "A", then print "B"
```

Delegate Constructor

The problem:

Most constructors usually perform identical initialization steps before executing individual operations

C++11 A **delegate constructor** calls another constructor of the same class to reduce the repetitive code by adding a function that does all the initialization steps

```
struct A {
    int   a;
    float b;
    bool  c;
    // standard constructor:
    A(int a1, float b1, bool c1) : a(a1), b(b1), c(c1) {
        // do a lot of work
    }

    A(int a1, float b1) : A(a1, b1, false)  {} // delegate construtor
    A(float b1)         : A(100, b1, false) {} // delegate construtor
};
```

explicit

The `explicit` keyword specifies that a *constructor* or *conversion operator* (C++11) does not allow implicit conversions or copy-initialization from single arguments or braced initializers

The problem:

```
struct MyString {
    MyString(int n);          // (1) allocate n bytes for the string
    MyString(const char *p); // (21) initializes starting from a raw string
};
MyString string = 'a';        // call (1), implicit conversion!!
```

`explicit` cannot be applied to *copy/move-constructors*

Most C++ constructors should be explicit

```cpp
struct A {
    A() {}
    A(int) {}
    A(int, int) {}
};
void f(const A&) {}

A a1 = {};      // ok
A a2(2);        // ok
A a3 = 1;       // ok (implicit)
A a4{4, 5};     // ok. Selected A(int, int)
A a5 = {4, 5}; // ok. Selected A(int, int)
f({});          // ok
f(1);           // ok
f({1});         // ok
```

```cpp
struct B {
    explicit B() {}
    explicit B(int) {}
    explicit B(int, int) {}
};
void f(const B&) {}

// B b1 = {};      // error implicit conversion
B b2(2);           // ok
// B b3 = 1;       // error implicit conversion
B b4{4, 5};        // ok. Selected B(int, int)
// B b5 = {4, 5}; // error implicit conversion
B b6 = (B) 1;      // OK: explicit cast
// f({});          // error implicit conversion
// f(1);           // error implicit conversion
// f({1});         // error implicit conversion
f(B{1});           // ok
```

[[nodiscard]] and Classes

C++17 allows setting `[[nodiscard]]` for the entire `class/struct`

```
[[nodiscard]] struct A {};
A f() { return A{}; }

auto x = f(); // ok
f();          // compiler warning
```

C++20 allows to set `[[nodiscard]]` for constructors

```
struct A {
    [[nodiscard]] A() {} // C++20 also allows [[nodiscard]] with a reason
};
void f(A {})

A a{};  // ok
f(A{}); // ok
A{};    // compiler warning
```

Copy Constructor

Copy Constructor

Copy Constructor

A **copy constructor** `T(const T&)` creates a new object as a *deep copy* of an existing object

```
struct A {
    A()          {} // default constructor
    A(int)       {} // non-default constructor
    A(const A&) {} // copy constructor → direct initialization
}
```

Copy Constructor Details

- Every class always defines an *implicit* or *explicit* copy constructor, potentially *deleted*
- The copy constructor implicitly calls the *default* Base class constructor
- Even the copy constructor is considered a *user-defined* constructor
- The copy constructor doesn't have template parameters, otherwise it is a standard member function
- The copy constructor must not be confused with the assignment operator `operator=`

```
MyStruct x;
MyStruct y{x}; // copy constructor
y = x;         // call the assignment operator=, not the copy constructor
               // → copy initialization, see next lecture
```

Copy Constructor Example

```
struct Array {
    int  size;
    int* array;

    Array(int size1) : size{size1} {
        array = new int[size];
    }
    // copy constructor, ": size{obj.size}" initializer list
    Array(const Array& obj) : size{obj.size} {
        array = new int[size];
        for (int i = 0; i < size; i++)
            array[i] = obj.array[i];
    }
};
Array x{100}; // do something with x.array ...
Array y{x};   // call "Array::Array(const Array&)"
```

Copy Constructor Usage

The copy constructor is used to:

- Initialize one object from another one having the same type
  - Direct constructor
  - Assignment operator

```
A a1;
A a2(a1);     // Direct copy initialization
A a3{a1};     // Direct copy initialization
A a4 = a1;    // Copy initialization
A a5 = {a1}; // Copy list initialization
```

- Copy an object which is *passed by-value* as input parameter of a function

```
void f(A a);
```

- Copy an object which is returned as result from a function*

```
A f() { return A(3); } // * see RVO optimization
```

Copy Constructor Usage Examples

```
struct A {
    A() {}
    A(const A& obj) { cout << "copy"; }
};

void f(A a) {} // pass by-value

A g1(A& a)  { return a; }

A g2()      { return A(); }

A a;
A b = a;    // copy constructor (assignment)   "copy"
A c(b);     // copy constructor (direct)       "copy"
f(b);       // copy constructor (argument)     "copy"
g1(a);      // copy constructor (return value) "copy"
A d = g2(); // * see RVO optimization (Advanced Concepts I)
```

Pass by-value and Copy Constructor

```
struct A {
    A() {}
    A(const A& obj) { cout << "expensive copy"; }
};

struct B : A {
    B() {}
    B(const B& obj) { cout << "cheap copy"; }
};

void f1(B b) {}
void f2(A a) {}

B b1;
f1(b1); // cheap copy
f2(b1); // expensive copy!! It calls A(const A&) implicitly
```

Deleted Copy Constructor

The *implicit* copy constructor of a class is marked as **deleted** if (simplified):

- It has a non-static member/base class of reference/`const` type

```
struct NonDefault {  int& x; }; // deleted copy constructor
```

- It has a non-static member/base class which has a deleted (or inaccessible) copy constructor

```
struct B { // deleted copy constructor
    NonDefault a;
};
struct B : NonDefault {}; // delete copy constructor
```

- It has a non-static member/base class with a deleted or inaccessible destructor
- The class has the move constructor (next lectures)

Class Destructor

Destructor [dtor]

A **destructor** is a special member function that is executed whenever an object is out-of-scope or whenever the `delete/delete[]` expression is applied to a pointer of that class

Goals: *resources releasing*

Syntax: `~T()` same name of the class and no return type

- Any object has exactly one *destructor*, which is always *implicitly* or *explicitly* declared
- C++20 The *destructor* can be `constexpr`

```
struct Array {
    int* array;

    Array() {   // constructor
        array = new int[10];
    }

    ~Array() {  // destructor
        delete[] array;
    }
};
int main() {
   Array a;     // call the constructor
   for (int i = 0; i < 5; i++)
       Array b; // call 5 times the constructor + destructor
}  // call the destructor of "a"
```

Class destructor is never inherited. *Base* class destructor is invoked *after* the current class destructor

Class destructors are called in reverse order. From the most Derived to the top Base class

```
struct A {
    ~A() { cout << "A"; }
};
struct B {
    ~B() { cout << "B"; }
};
struct C : A {
    B b;              // call ~B()
    ~C() { cout << "C"; }
};
int main() {
    C b;  // print "C", then "B", then "A"
}
```

Defaulted Constructors, Destructor, and Operators (=default)

C++11 The compiler can automatically generate

- **default/copy/move constructors**
  ```
  A() = default
  A(const A&) = default
  A(A&&) = default
  ```
- **destructor**
  ```
  ~A() = default
  ```
- **copy/move assignment operators** `A& operator=(const A&) = default`
  ```
  A& operator=(A&&) = default
  ```
- **spaceship operator**
  ```
  auto operator<=>(const A&) const = default
  ```

`= default` implies `constexpr`, but not `noexcept` or `explicit`

When the compiler-generated constructors, destructors, and operators are useful:

- Change the visibility of non-user provided constructors and assignment operators (`public`, `protected`, `private`)
- Make visible the declarations of such members

The **defaulted** default constructor has a similar effect as a user-defined constructor with empty body and empty initializer list

When the compiler-generated constructor is useful:

- Any user-provided constructor disables implicitly-generated default constructor
- Force the default values for the class data members

```
struct A {
   A(int v1) {}   // delete implicitly-defined default ctor because
                  // a user-provided constructor is defined

   A() = default; // now, A has the default constructor
};
```

```
struct B {
protected:
    B() = default; // now it is protected
};
```

```
struct C {
    int x;
//  C() {}         // 'x' is undefined
    C() = default; // 'x' is zero
};
```

Class Keywords

this Keyword

this

Every object has access to its own address through the pointer `this`

Explicit usage is not mandatory (and not suggested)

`this` is necessary when:

- The name of a local variable is equal to some member name
- Return reference to the calling object

```
struct A {
    int x;
    void f(int x) {
        this->x = x; // without "this" has no effect
    }
    const A& g() {
        return *this;
    }
};
```

static Keyword

The keyword `static` declares members (fields or methods) that are not bound to class instances. A **static** member is shared by all objects of the class

```
struct A {
    int x;

    int f() { return x; }

    static int g() { return 3; }  // g() cannot access 'x' as it is associated
};                                // with class instances
A a{4};
a.f();  // call the class instance method
A::g(); // call the static class method
a.g();  // as an alternative, a class instance can access static class members
```

```
struct A {
   static const int       a = 4;          // C++03
   static constexpr float b = 4.2f;       // better, C++11
// static const float     c = 4.2f;       // only GNU extension (GCC)

   static constexpr int f() { return 1; } // ok, C++11
// static const int     g() { return 1; } // 'const' refers to the return type
};
```

Non-`const` `static` data members cannot be *directly* initialized inline (see `Translation Units` lecture)...before C++17

```
struct A {
// static int        a = 4; // compiler error
   static int        a;     // ok, declaration only
   static inline int b = 4; // ok from C++17

   static int f() { return 2; }
   static int g();          // ok, declaration only
};

int A::a = 4;               // ok, undefined reference without this definition
int A::g() { return 3; }    // ok, undefined reference without this definition
```

```
struct A {
    static int x; // declaration

    static int f() { return x; }

    static int& g() { return x; }
};
int A::x = 3; // definition

//-----------------------------------------------------------------------------

A::f();     // return 3
A::x++;
A::f();     // return 4
A::g() = 7;
A::f();     // return 7
```

- A `static` member function can only access `static` class members
- A non-`static` member function can access `static` class members

```
struct A {
    int                x = 3;
    static inline int y = 4;

    int         f1() { return x; }  // ok
//  static int f2() { return x; }  // compiler error, 'x' is not visible
    int         g1() { return y; }  // ok
    static int g2() { return y; }  // ok

    struct B {
        int h() { return y + g2(); } // ok
    }; // 'x', 'f1()', 'g1()' are not visible within 'B'
};
```

Const member functions

Const member functions (**inspectors** or **observers**) are functions marked with `const` that are not allowed to change the object logical state

The compiler prevents from inadvertently mutating/changing the data members of *observer* functions → All data members are marked `const` within an **observer** method, including the `this` pointer

- The *physical state* can still be modified, see `mutable` member functions ⇝
- Member functions without a `const` suffix are called *non-const member functions* or **mutators/modifiers**

```
struct A {
    int  x = 3;
    int* p;

    int get() const {
     // x = 2;         // compile error class variables cannot be modified
     // p = nullptr;   // compile error class variables cannot be modified
        p[0] = 3;      // ok, p is 'int* const' -> its content is
                       // not protected
        return x;
    }
};
```

A common case where `const` member functions are useful is to enforce const correctness when accessing pointers, see `Advanced Concepts I, Const Correctness`

The `const` keyword is part of the function signature. Therefore, a class can implement two similar methods, one which is called when the object is `const`, and one that is not

```
class A {
    int x = 3;
public:
    int& get1()       { return x; } // read and write
    int  get1() const { return x; } // read only
    int& get2()       { return x; } // read and write
};

A a1;
cout << a1.get1();    // ok
cout << a1.get2();    // ok
a1.get1() = 4;        // ok
const A a2;
cout << a2.get1();    // ok
// cout << a2.get2(); // compile error "a2" is const
//a2.get1() = 5;      // compile error only "get1() const" is available
```

`mutable` Keyword

mutable

`mutable` data members of *const* class instances are modifiable. They should be part of the object *physical state*, but not of the *logical state*

- It is particularly useful if most of the members should be constant but a few need to be modified
- *Conceptually,* `mutable` *members should not change anything that can be retrieved from the class interface*

```
struct A {
    int         x = 3;
    mutable int y = 5;
};
const A a;
// a.x = 3; // compiler error const
a.y = 5;    // ok
```

`using` Keyword for type declaration

The `using` keyword is used to declare a *type alias* tied to a specific class

```
struct A {
    using type = int;
};

typename A::type x = 3; // "typename" keyword is needed when we refer to types

struct B : A {};

typename B::type x = 4; // B can use "type" as it is public in A
```

`using` Keyword for Inheritance

The `using` keyword can be also used to change the *inheritance attribute* of member data or functions

```
struct A {
protected:
    int x = 3;
};

struct B : A {
public:
    using A::x;
};

B b;
b.x = 3;  // ok, "b.x" is public
```

friend Class

A `friend` class can access the `private` and `protected` members of the class in which it is declared as a `friend`

Friendship properties:

- **Not Symmetric**: if class `A` is a friend of class `B`, class `B` is not automatically a friend of class `A`
- **Not Transitive**: if class `A` is a friend of class `B`, and class `B` is a friend of class `C`, class `A` is not automatically a friend of class `C`
- **Not Inherited**: if class `Base` is a friend of class `X`, subclass `Derived` is not automatically a friend of class `X`; and if class `X` is a friend of class `Base`, class `X` is not automatically a friend of subclass `Derived`

```
class B;   // class declaration

class A {
    friend class B;
    int x;      // private
};

class B {
    int f(A a) { return a.x; } // ok, B is friend of A
};

class C : B {
//  int f(A a) { return a.x; } // compile error not inherited
};
```

`friend` Method

A *non-member function* can access the `private` and `protected` members of a class if it is declared a `friend` of that class

```
class A {
    int x = 3;  // private

    friend int f(A a); // friendship declaration, no implementation
};

//'f' is not a member function of any class
int f(A a) {
    return a.x;  // A is friend of f(A)
}
```

`friend` methods are commonly used for implementing the stream operator `operator<<`

delete Keyword (C++11)

The `delete` keyword explicitly marks a member function as deleted and any use results in a compiler error. When it is applied to *copy/move constructor* or *assignment*, it prevents the compiler from implicitly generating these functions

The default copy/move functions for a class can produce unexpected results. The keyword `delete` prevents these errors

```
struct A {
    A()         = default;
    A(const A&) = delete; // e.g. deleted because unsafe or expensive
};
void f(A a) {}  // implicit call to copy constructor

A a;
// f(a);        // compile error marked as deleted
```

Modern C++ Programming

8. Object-Oriented Programming II

Polymorphism and Operator Overloading

Federico Busato

2024-03-29

Table of Contents

3 Operator Overloading

Table of Contents

Polymorphism

Polymorphism

In Object-Oriented Programming (OOP), **polymorphism** (meaning "having multiple forms") is the capability of an object of *mutating* its behavior in accordance with the specific usage *context*

- At run-time, objects of a *base class* behaves as objects of a *derived class*
- A **Base** class may define and implement polymorphic methods, and **derived** classes can **override** them, which means they provide their own implementations, invoked at run-time depending on the context

Polymorphism vs. Overloading

Overloading is a form of static polymorphism (compile-time polymorphism)

In C++, the term **polymorphic** is strongly associated with dynamic polymorphism (*overriding*)

```
// overloading example
void f(int a)    {}

void f(double b) {}

f(3);   // calls f(int)
f(3.3); // calls f(double)
```

Function Binding

Connecting the function call to the function body is called *Binding*

- In **Early Binding** or *Static Binding* or *Compile-time Binding*, the compiler identifies the type of object at compile-time
  - the program can jump directly to the function address
- In **Late Binding** or *Dynamic Binding* or *Run-time binding*, the run-time identifies the type of object at execution-time and *then* matches the function call with the correct function definition
  - the program has to read the address held in the pointer and then jump to that address (less efficient since it involves an extra level of indirection)

C++ achieves **late binding** by declaring a `virtual` function

Polymorphism - The problem

```
struct A {
    void f()  { cout << "A"; }
};

struct B : A {
    void f() { cout << "B"; }
};

void g(A& a) { a.f(); } // accepts A and B

void h(B& b) { b.f(); } // accepts only B

A a;
B b;
g(a);    // print "A"
g(b);    // print "A" not "B"!!!
```

Polymorphism - `virtual` method

```
struct A {
    virtual void f() { cout << "A"; }
}; // now "f()" is virtual, evaluated at run-time

struct B : A {
    void f() { cout << "B"; }
}; // now "B::f()" overrides "A::f()", evaluated at run-time

void g(A& a) { a.f(); } // accepts A and B

A a;
B b;
g(a); // print "A"
g(b); // NOW, print "B"!!!
```

The `virtual` keyword is *not necessary* in derived classes, but it improves *readability* and clearly advertises the fact to the user that the function is `virtual`

When `virtual` works

```
struct A {
    virtual void f() { cout << "A"; }
};

struct B : A {
    void f() { cout << "B"; }
};

void f(A& a) { a.f();  } // ok, print "B"
void g(A* a) { a->f(); } // ok, print "B"
void h(A  a) { a.f();  } // does not work!! print "A"

B b;
f(b);  // print "B"
g(&b); // print "B"
h(b);  // print "A" (cast to A)
```

Polymorphism Dynamic Behavior

```
struct A {
    virtual void f() { cout << "A"; }
};

struct B : A {
    void f() { cout << "B"; }
};

A* get_object(bool selectA) {
    return (selectA) ? new A() : new B();
}

get_object(true)->f();  // print "A"
get_object(false)->f(); // print "B"
```

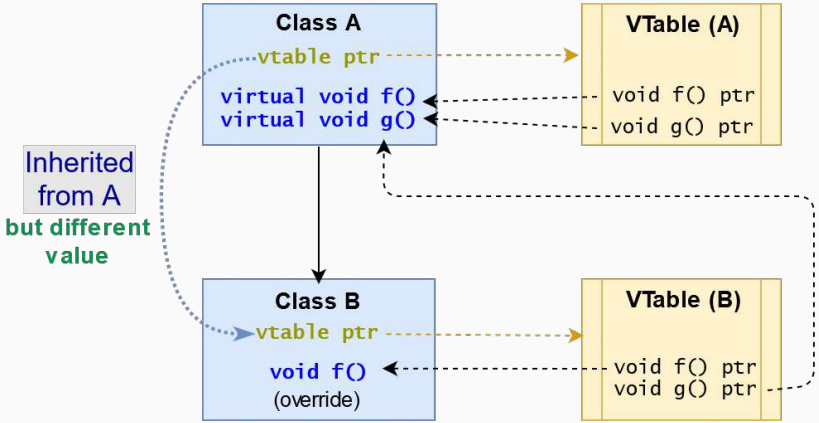
vtable

The **virtual table** (`vtable`) is a lookup table of functions used to resolve function calls and support *dynamic dispatch* (late binding)

A *virtual table* contains one entry for each `virtual` function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the *most-derived* function accessible by that class

The compiler adds a *hidden* pointer to the base class which points to the virtual table for that class (`sizeof` considers the `vtable` pointer)

```
struct A {
    virtual void f();
    virtual void g();
};

struct B : A {
    void f();
};
```



Does the `vtable` really exist? (answer: YES)

```
struct A {
    int x = 3;
    virtual void f() { cout << "abc"; }
};

A* a1 = new A;
A* a2 = (A*) malloc(sizeof(A));

cout << a1->x;  // print "3"
cout << a2->x;  // undefined value!!
a1->f();        // print "abc"
a2->f();        // segmentation fault
```

Lesson learned: Never use `malloc` in C++

Virtual Method Notes

`virtual` classes allocate one extra pointer (hidden)

```
struct A {
    virtual void f1();
    virtual void f2();
};

class B : A {};

cout << sizeof(A); // 8 bytes (vtable pointer)
cout << sizeof(B); // 8 bytes (vtable pointer)
```

override Keyword (C++11)

The override keyword ensures that the function is virtual and is overriding a virtual function from a base class

It forces the compiler to check the base class to see if there is a `virtual` function with this exact signature

`override` implies `virtual` (`virtual` should be omitted)

```
struct A {
    virtual void f(int a);          // a "float" value is casted to "int"
};                                  // ***

struct B : A {
    void f(int   a) override;       // ok
    void f(float a);                // (still) very dangerous!!
                                    // ***

//  void f(float a) override;       // compile error not safe
//  void f(int   a) const override; // compile error not safe
};

// *** f(3.3f) has a different behavior between A and B
```

final Keyword (C++11)

The `final` keyword prevents inheriting from classes or overriding methods in derived classes

```
struct A {
    virtual void f(int a) final;  // "final" method
};

struct B : A {
//  void f(int a);   // compile error f(int) is "final"
    void f(float a); // dangerous (still possible)
};                   // "override" prevents these errors

struct C final {     // cannot be extended
};
// struct D : C {    // compile error C is "final"
// };
```

Virtual Methods (Common Error 1)

All classes with at least one `virtual` method should declare a `virtual` *destructor*

```
struct A {
    ~A() { cout << "A"; }   // <-- here the problem (not virtual)
    virtual void f(int a) {}
};
struct B : A {
    int* array;
    B()   { array = new int[1000000]; }
    ~B() { delete[] array;           }
};
//-----------------------------------------------------------------------
void destroy(A* a) {
    delete a;    // call ~A()
}
B* b = new B;
destroy(b); // without virtual, ~B() is not called
            // destroy() prints only "A" -> huge memory leak!!
```

Virtual Methods (Common Error 2)

Do not call virtual methods in constructor and destructor

- *Constructor*: The derived class is not ready until constructor is completed
- *Destructor*: The derived class is already destroyed

```
struct A {
    A() { f(); }  // what instance is called? "B" is not ready
                  // it calls A::f(), even though A::f() is virtual
    virtual void f() { cout << "Explosion"; }
};
struct B : A {
    B() = default; // call A(). Note: A() may be also implicit

    void f() override { cout << "Safe"; }
};

B b; // call B(), print "Explosion", not "Safe"!!
```

Virtual Methods (Common Error 3)

Do not use default parameters in virtual methods

Default parameters are not inherited

```
struct A {
    virtual void f(int i = 5) { cout << "A::" << i << "\n"; }
    virtual void g(int i = 5) { cout << "A::" << i << "\n"; }
};
struct B : A {
    void f(int i = 3) override { cout << "B::" << i << "\n"; }
    void g(int i)     override { cout << "B::" << i << "\n"; }
};
A a; B b;
a.f();      // ok, print "A::5"
b.f();      // ok, print "B::3"

A& ab = b;
ab.f();     // !!! print "B::5"  // the virtual table of A
                                 // contains f(int i = 5) and
ab.g();     // !!! print "B::5"  // g(int i = 5) but it points
                                 // to B implementations
```

Pure Virtual Method

A **pure virtual method** is a function that must be implemented in derived classes (concrete implementation)

Pure virtual functions can have or not have a body

```
struct A {
    virtual void f() = 0; // pure virtual without body
    virtual void g() = 0; // pure virtual with body
};
void A::g() {} // pure virtual implementation (body) for g()

struct B : A {
    void f() override {} // must be implemented
    void g() override {} // must be implemented
};
```

A class with one *pure virtual function* cannot be instantiated

```
struct A {
    virtual void f() = 0;
};

struct B1 : A {
//  virtual void f() = 0; // implicitly declared
};

struct B2 : A {
    void f() override {}
};

// A  a;  // "A"  has a pure virtual method
// B1 b1; // "B1" has a pure virtual method
B2 b2;    //  ok
```

Abstract Class and Interface

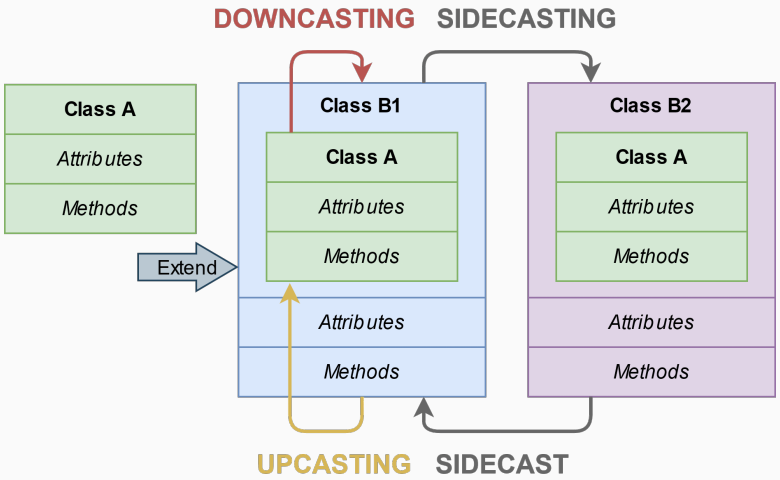
- A class is **interface** if it has only *pure virtual* functions and optionally (*suggested*) a virtual destructor. Interfaces do not have implementation or data
- A class is **abstract** if it has at least one *pure virtual* function

```
struct A {          // INTERFACE
    virtual ~A();   // to implement
    virtual void f() = 0;
};

struct B {          // ABSTRACT CLASS
    B() {}          // abstract classes may have a contructor
    virtual void g() = 0; // at least one pure virtual
protected:
    int x;          // additional data
};
```

Inheritance Casting and Run-time Type Identification ★

Hierarchy Casting

Class-casting allows implicit or explicit conversion of a class into another one across its hierarchy



Hierarchy Casting

Upcasting Conversion between a derived class reference or pointer to a base class

- It can be *implicit* or *explicit*
- It is safe
- `static_cast` or `dynamic_cast` // see next slides

Downcasting Conversion between a base class reference or pointer to a derived class

- It is only *explicit*
- It can be dangerous
- `static_cast` or `dynamic_cast`

Sidecasting (*Cross-cast*) Conversion between a class reference or pointer to another class of the same hierarchy level

- It is only *explicit*
- It can be dangerous
- `dynamic_cast`

Upcasting and Downcasting Example

```
struct A {
   virtual void f() { cout << "A"; }
};

struct B : A {
   int var = 3;
   void f() override { cout << "B"; }
};

A  a;
B  b;
A& a1 = b; // implicit cast upcasting

static_cast<A&>(b).f();          // print "B" upcasting
static_cast<B&>(a).f();          // print "A" downcasting
cout << b.var;                   // print 3 (no cast)
cout << static_cast<B&>(a).var;  // potential segfault!!! downcasting
```

Sidecasting Example

```
struct A {
   virtual void f() { cout << "A"; }
};

struct B1 : A {
   void f() override { cout << "B1"; }
};
struct B2 : A {
   void f() override { cout << "B2"; }
};

B1 b1;
B2 b2;
dynamic_cast<B2&>(b1).f();     // sidecasting, throw std::bad_cast
dynamic_cast<B1&>(b2).f();     // sidecasting, throw std::bad_cast
// static_cast<B1&>(b2).f(); // compile error
```

Run-time Type Identification

RTTI

Run-Time Type Information (**RTTI**) is a mechanism that allows the type of object to be *determined at runtime*

C++ expresses RTTI through three features:

- `dynamic_cast` keyword: conversion of polymorphic types
- `typeid` keyword: identifying the exact type of object
- `type_info` class: type information returned by the `typeid` operator

RTTI is available only for classes that are *polymorphic*, which means they have *at least one* virtual method

type_info and typeid

`type_info` class has the method `name()` which returns the name of the type

```
struct A {
   virtual void f() {}
};

struct B : A {};

A  a;
B  b;
A& a1 = b; // implicit upcasting
cout << typeid(a).name();  // print "1A"
cout << typeid(b).name();  // print "1B"
cout << typeid(a1).name(); // print "1B"
```

dynamic_cast

`dynamic_cast`, differently from `static_cast`, uses *RTTI* for deducing the correctness of the output type
This operation happens at run-time and it is expensive

`dynamic_cast<New>(Obj)` has the following properties:

- Convert between a derived class `Obj` to a base class `New` $\rightarrow$ *upcasting*. `New/Obj` are both pointers or references
- Throw `std::bad_cast` if `New/Obj` are *references* and `New/Obj` cannot be converted
- Returns `NULL` if `New/Obj` are *pointers* and `New/Obj` cannot be converted

dynamic_cast Example 1

```
struct A {
   virtual void f() { cout << "A"; }
};

struct B : A {
   void f() override { cout << "B"; }
};

A a;
B b;
dynamic_cast<A&>(b).f();     // print "B" upcasting

// dynamic_cast<B&>(a).f(); // throw std::bad_cast
                            // wrong downcasting

dynamic_cast<B*>(&a);        // returns nullptr
                            // wrong downcasting
```

dynamic_cast Example 2

```
struct A {
    virtual void f() { cout << "A"; }
};
struct B : A {
    void f() override { cout << "B"; }
};

A* get_object(bool selectA) {
    return (selectA) ? new A() : new B();
}

void g(bool value) {
    A* a = get_object(value);
    B* b = dynamic_cast<B*>(a); // downcasting + check
    if (b != nullptr)
        b->f();       // exectuted only when it is safe
}
```

Operator Overloading

Operator Overloading

Operator overloading is a special case of polymorphism in which some *operators* are treated as polymorphic functions and have different behaviors depending on the type of its arguments

```
struct Point {
    int x, y;

    Point operator+(const Point& p) const {
        return {x + p.x, y + p.y};
    }
};

Point a{1, 2};
Point b{5, 3};
Point c = a + b; // "c" is (6, 5)
```

Operator Overloading

Category	Operators
Arithmetic	`+ - * / % ++ --`
Comparison	`== != < <= > >= <=>`
Bitwise	`| & ^ ~ << >>`
Logical	`! && ||`
Compound Assignment Arithmetic	`+= -= *= /= %=`
Compound Assignment Bitwise	`>>= <<= |= &= ^=`
Subscript	`[]`
Function call	`()`
Address-of, Reference, Dereferencing	`& -> ->* *`
Memory	`new new[] delete delete[]`
Comma	`,`

- Categories not in bold are rarely used in practice
- Operators that cannot be overloaded: `? . .* :: sizeof typeof`

Comparison Operator `operator<`

Relational and comparison operators `operator<, <=, ==, >= >` are used for comparing two objects

In particular, the `operator<` is used to determine the ordering of a set of objects (e.g. sort)

```
#include <algorithm>
struct A {
    int x;

    bool operator<(A a) const {
        return x * x < a.x * a.x;
    }
};
A array[] = {5, -1, 4, -7};
std::sort(array, array + 4);
// array: {-1, 4, 5, -7}
```

C++20 allows overloading the **spaceship operator** `<=>` for replacing all comparison operators `operator<, <=, ==, >= >`

```
struct A {
    bool operator==(const A&) const;
    bool operator!=(const A&) const;
    bool operator<(const A&)  const;
    bool operator<=(const A&) const;
    bool operator>(const A&)  const;
    bool operator>=(const A&) const;
};

// replaced by
struct B {
    int operator<=>(const B&) const;
};
```

```cpp
#include <compare>

struct Obj {
    int x;

    auto operator<=>(const Obj& other) {
        return x - other.x; // or even better "x <=> other.x"
    }
};

Obj a{3};
Obj b{5};
a < b;        // true, even if the operator< is not defined
a == b;       // false
a <=> b < 0; // true
```

The compiler can also generate the code for the *spaceship operator* `= default`, even for multiple fields and arrays, by using the default comparison semantic of its members

```cpp
#include <compare>

struct Obj {
    int  x;
    char y;
    short z[2];

    auto operator<=>(const Obj&) const = default;
    // if x == other.x, then compare y
    // if y == other.y, then compare z
    // if z[0] == other.z[0], then compare z[1]
};
```

The *spaceship operator* can use one of the following ordering:

strong ordering
- if `a` is equivalent to `b`, `f(a)` is also equivalent to `f(b)`
- exactly one of `<`, `==`, or `>` must be true
- integral types, e.g. `int`, `char`

weak ordering
- if `a` is equivalent to `b`, `f(a)` may not be equivalent to `f(b)`
- exactly one of `<`, `==`, or `>` must be true
- rectangles, e.g. `R{2, 5} == R{5, 2}`

partial ordering
- if `a` is equivalent to `b`, `f(a)` may not be equivalent to `f(b)`
- `<`, `==`, or `>` may all be false
- floating-point `float`, e.g. `NaN`

Subscript Operator `operator[]`

The **array subscript** `operator[]` allows accessing to an object in an array-like fashion

The operator accepts everything as parameter, not just integers

```
struct A {
    char permutation[] {'c', 'b', 'd', 'a', 'h', 'y'};

    char& operator[](char c) { // read/write
        return permutation[c - 'a'];
    }
    char operator[](char c) const { // read only
        return permutation[c - 'a'];
    }
};

A a;
a['d'] = 't';
```

Multidimensional Subscript Operator `operator[]`

C++23 introduces the *multidimensional subscript operator* and replaces the standard behavior of the *comma operator*

```
struct A {
    int operator[](int x) { return x; }
};
struct B {
    int operator[](int x, int y) { return x * y; } // not allowed before C++23
};

int main() {
    A a;
    cout << a[3, 4]; // return 4 (bug)
    B b;
    cout << b[3, 4]; // return 12, C++23
}
```

Function Call Operator `operator()`

The **function call operator** `operator()` is generally overloaded to create objects which behave like functions, or for classes that have a primary operation (see `Basic Concepts IV` lecture)

```
#include <numeric>  // for std::accumulate

struct Multiply {
    int operator()(int a, int b) const {
        return a * b;
    }
};

int array[]   = { 2, 3, 4 };
int factorial = std::accumulate(array, array + 3, 1, Multiply{});
cout << factorial;  // 24
```

`static operator()` and `static operator[]`

C++23 introduces the `static` version of the *function call operator* `operator()` and the *subscript operator* `operator[]` to avoid passing the `this` pointer

```
#include <numeric>  // for std::accumulate

struct Multiply {
//  int         operator()(int a, int b); // declaration only
    static int operator()(int a, int b); // best efficiency, no need to access
};                                       // internal data members

struct MyArray {
//  int         operator[](int x);
    static int operator[](int x);        // best efficiency
};
int array[]   = { 2, 3, 4 };
int factorial = std::accumulate(array, array + 3, 1, Multiply{});
```

The **conversion operator** `operator T()` allows objects to be either implicitly or explicitly (casting) converted to another type

```
class MyBool {
    int x;
public:
    MyBool(int x1) : x{x1} {}

    operator bool() const {  // implicit return type
        return x == 0;
    }
};

MyBool my_bool{3};
bool b = my_bool;  // b = false, call operator bool()
```

C++11 **Conversion operators** can be marked `explicit` to prevent implicit conversions. It is a good practice as for class constructors

```
struct A {
    operator bool() { return true; }
};

struct B {
    explicit operator bool() { return true; }
};

A a;
B b;
bool    c1 = a;
// bool c2 = b;  // compile error: explicit
bool    c3 = static_cast<bool>(b);
```

Return Type Overloading Resolution ★

```
struct A {
    operator float() { return 3.0f; }
    operator int()   { return 2;    }
};

auto f() {
    return A{};
}

float x = f();
int   y = f();
cout << x << " " << y; // x=3.0f, y=2
```

Increment and Decrement Operators `operator++/--`

The increment and decrement operators `operator++`, `operator--` are used to update the value of a variable by one unit

```
struct A {
    int* ptr;
    int  pos;
    A& operator++() {      // Prefix notation (++var):
        ++ptr;             //  returns the new copy of the object by-reference
        ++pos;
        return *this;
    }
    A operator++(int a) {  // Postfix notation (var++):
        A tmp = *this;     //  returns the old copy of the object by-value
        ++ptr;
        ++pos;
        return tmp;
    }
};
```

The **assignment operator** operator= is used to copy values from one object to another *already existing* object

```
#include <algorithm>  //std::fill, std::copy
struct Array {
    char* array;
    int   size;

    Array(int size1, char value) : size{size1} {
         array = new char[size];
         std::fill(array, array + size, value);
    }
    ~Array() { delete[] array; }

    Array& operator=(const Array& x) { .... } // --> see next slide
};
Array a{5, 'o'}; // ["ooooo"]
Array b{3, 'b'}; // ["bbb"]
a = b;           // a = ["bbb"] <-- goal
```

- First option:

```
Array& operator=(const Array& x) {
    if (this == &x)          // (1) Check for self assignment
        return *this;
    delete[] array;          // (2) Release class resources
    size  = x.size;          // (3) Re-initialize class resources
    array = new int[x.size];
    std::copy(x.array, x.array + size, array); // (4) deep copy
    return *this;
}
```

- Second option (less intuitive):

```
Array& operator=(Array x) { // pass by-value
    swap(*this, x);         // now we need a swap function for A
    return *this;           // x is destroyed at the end
}                           // --> see next slide
```

`swap` method:

```
friend void swap(A& x, A& y) {
    using std::swap;
    swap(x.size,  y.size);
    swap(x.array, y.array);
}
```

- **why using `std::swap`?** if `swap(x, y)` finds a better match, it will use that instead of `std::swap`
- **why `friend`?** it allows the function to be used from outside the structure/class scope

---

stackoverflow.com/questions/3279543
stackoverflow.com/questions/5695548

Stream Operator `operator<<`

The **stream operation** `operator<<` can be overloaded to perform input and output for user-defined types

```cpp
#include <iostream>

struct Point {
    int x, y;

    friend std::ostream& operator<<(std::ostream& stream,
                                    const Point&  point) {
        stream << "(" << point.x << "," << point.y << ")";
        return stream;
    }
    // operator<< is a member of std::ostream -> need friend
};  // implementation and definition can be splitted (not suggested for operator<<)
Point point{1, 2};
std::cout << point;  // print "(1, 2)"
```

Operators Precedence

Operators preserve **precedence** and **short-circuit** properties

```
struct MyInt {
    int x;

    int operator^(int exp) { // exponential
        int ret = 1;
        for (int i = 0; i < exp; i++)
            ret *= x;
        return ret;
    }
};
MyInt x{3};
int   y = x^2;
cout << y;  // 9
int   z = x^2 + 2;
cout << z;  // 81 !!!
```

Binary Operators Note

Binary operators should be implemented as `friend` methods

```
struct A {}; struct C {};

struct B : A {
    bool operator==(const A& x) { return true; }
};
struct D : C {
    friend bool operator==(const C& x, const C& y) { return true; } // inline
};
// bool operator==(const C& x, const C& y) { return true; }  // out-of-line

A a;  B b;  C c; D d;
b == a;     // ok
// a == b; // compile error // "A" does not have == operator
c == d;     // ok, use operator==(const C&, const C&)
d == c;     // ok, use operator==(const C&, const C&)
```

C++ Object Layout ★

Overview

The term **layout** refers to how an object is arranged in memory

C++ defines four types of *layouts*:

- `aggregate`
- `trivial copyable`
- `standard layout`
- `plain-old data (POD)`

Such *layouts* are important to understand how the C++ objects interact with pure C API and for optimization purposes, e.g. pass in registers, `memcpy`, and serialization

Aggregate

An **aggregate** is an `array`, `struct`, or `class` which supports *aggregate initialization* (form of list-initialization) through curly braces syntax {}

- No *user-provided* constructors
- No `private` / `protected` *non-* `static` data members and *base* class
- No `virtual` functions

* No base classes, until C++17

* No *brace-or-equal-initializers* for non-static data members, until C++14

R Apply recursively to *base* classes *non-* `static` data members

No restrictions:

- *Non-* `static` uninitialized (until C++14) data and function members
- `static` data and function members

`stackoverflow.com/questions/4178175`

```
struct Aggregate {
    int x;       // ok, public member
    int y[3];    // ok, arrays are also fine
    int z { 3 }; // only C++14

    Aggregate() = default;                    // ok, defaulted constructor
    Aggregate& operator=(const& Aggregate);  // ok, function
private:                                      // copy-assignment
    void f() {}                               // ok, private function
};

struct NotAggregate1 {
    NotAggregate1();   // !! user-provided constructor
    virtual void f();  // !! virtual function
};
class NotAggregate2 : NotAggregate1 { // !! the base class is not an aggregate
    int           x;   // !! x is private
    NotAggregate1 y;   // !! y is not an aggregate (recursive property)
};
```

```
struct Aggregate1 {
    int x;
    struct Aggregate2 {
        int a;
        int b[3];
    } y;
};

int        array1[3] = {1, 2, 3};
int        array2[3]   {1, 2, 3};
Aggregate1 agg1      = {1, {2, {3, 4, 5}}};
Aggregate1 agg2        {1, {2, {3, 4, 5}}};
Aggregate1 agg3      = {1, 2, 3, 4, 5};
```

Trivial Class

A `Trivial Class` is a `class` `trivial copyable` (supports `memcpy`)

Trivial copyable:

- No *user-provided* copy/move/default *constructors*, *destructor*, and copy/move *assignment* operators
- No `virtual` functions
- R Apply recursively to *base* classes and *non-* `static` data members

No restrictions:

- *User-declared* constructors different from copy/move/default
- Functions or `static` ,*non-* `static` data members initialization
- `protected` / `private` members

```
struct NonTrivial {
    NonTrivial();     // !! user-provided constructor
    virtual void f(); // !! virtual function
};

struct Trivial1 {
    Trivial1() = default;    // ok, defaulted constructor
    Trivial1(int) {}         // ok, user-default constructor
    static int x;            // ok, static member
    void f();                // ok, function
private:
    int z { 3 }              // ok, private and initialized
};
struct Trivial2 : Trivial1 { // ok, base class is trivial
    int Trivial1[3];         // ok, array of trivials is trivial
};
```

Standard-Layout

A `standard-layout class` is a class with the same memory layout of the equivalent C `struct` or `union` (useful for communicating with other languages)

- No `virtual` functions
- Only one control access (`public` / `protected` / `private`) for all *non-* `static` data members
- No base classes with *non-* `static` data members
- No base classes of the same type as the first *non-* `static` data member

R Apply recursively to *base* classes and *non-* `static` data members

```
struct StandardLayout1 {
    StandardLayout1(); // ok, user-provided contructor
    void f();          // ok, non-virtual function
};

class StandardLayout2 : StandardLayout1 {
    int x, y;          // ok, both are private
    StandardLayout1 y; // ok, 'y' is not the first data member
};

struct StandardLayout4 : StandardLayout1, StandardLayout2 {
    // ok, can use multiple inheritance as long as only
    // one class in the hierarchy has non-static data members
};
```

Plain Old Data (POD)

Plain Old Data (POD): Trivial copyable (**T**) + Standard-Layout (**S**)

(**T**) No *user-provided* copy/move/default *constructors*, *destructor*, and copy/move *assignment* operators

(**S**) Only one control access (`public` / `protected` / `private`) for all *non-* `static` data members

(**S**) No base classes with *non-* `static` data members

(**S**) No base classes of the same type as the first *non-* `static` data member

(**T, S**) No `virtual` functions

R Apply recursively to *base* classes and *non-* `static` data members

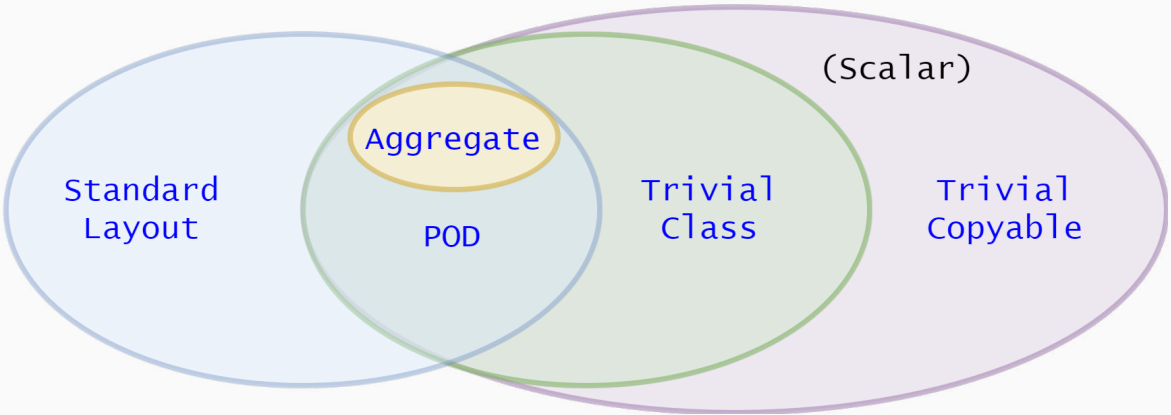
C++ `std` Utilities

C++11 provides three utilities to check if a type is POD, Trivial Copyable, Standard-Layout

- `std::is_pod` checks for POD, deprecated in C++20
- `std::is_trivially_copyable` checks for trivial copyable
- `std::is_standard_layout` checks for standard-layout

```cpp
#include <type_traits>

struct A {
    int x;
private:
    int y;
};

cout << std::is_trivially_copyable_v<A>; // true
cout << std::is_standard_layout_v<A>;    // false
cout << std::is_pod_v<A>;                // false
```

Object Layout Hierarchy



Modern C++ Programming

9. Templates and Meta-programming I

Function Templates and Compile-Time Utilities

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

Template Books

C++ Templates: The Complete Guide (2nd)
D. Vandevoorde, N. M. Josuttis, D. Gregor, 2017

Function Template

Template Overview

Template

A **template** is a mechanism for generic programming to provide a *"schema"* (or *placeholders*) to represent the structure of an entity

In C++, *templates* are a compile-time functionality to represent:

- A family of **functions**
- A family of **classes**
- A family of **variables** C++14

The problem: We want to define a function to handle different types

```
int add(int a, int b) {
    return a + b;
}

float add(float a, float b) { // overloading
    return a + b;
}

char   add(char a, char b)     { ... } // overloading
ClassX add(ClassX a, ClassX b) { ... } // overloading
```

- Redundant code!!
- How many functions we have to write!?
- If the user introduces a new type we have to write another function!!

Function Template

A **function template** is a function schema that operates with *generic* types (independent of any particular type) or concrete values

A function template works with multiple types without repeating the entire code for each of them

```
template<typename T> // or template<class T>
T add(T a, T b) {
    return a + b;
}

int   c1 = add(3, 4);       // c1 = 7
float c2 = add(3.0f, 4.0f); // c2 = 7.0f
```

Templates: Benefits and Drawbacks

Benefits

- **Generic Programming**: Less code and reusable. Reduce *redundancy*, better *maintainability* and *flexibility*
- **Performance**. Computation can be done/optimized at compile-time $\rightarrow$ *faster*

Drawbacks

- **Readability**. "With respect to C++, the syntax and idioms of templates are *esoteric* compared to conventional C++ programming, and templates can be very difficult to understand" [wikipedia] $\rightarrow$ hard to read, cryptic error messages
- **Compile Time/Binary Size**. Templates are implicitly instantiated for every distinct parameters

Template Instantiation

The **template instantiation** is the substitution of template parameters with concrete values or types

The compiler *automatically* generates a **function implementation** for each template instantiation

```
template<typename T>
T add(T a, T b) {
    return a + b;
}
add(3, 4);       // generates: int   add(int, int)
add(3.0f, 4.0f); // generates: float add(float, float)
add(2, 6);       // already generated
// other instances are not generated
// e.g. char add(char,char)
```

Implicit and Explicit Template Instantiation

Implicit Template Instantiation

Implicit template instantiation occurs when the compiler generates code depending on the *deduced argument types* or the *explicit template arguments* and only when the definition is needed

Explicit Template Instantiation

Explicit template instantiation occurs when the compiler generates code depending only on the *explicit template arguments* specified in the declaration. Useful when dealing with multiple translation units to reduce the binary size

Implicit and Explicit Template Instantiation

```
template<typename T>
void f(T a) {}

void g() {
    f(3);                   // generates: void f(int) → implicit
    f<short>(3.0);          // generates: void f(short) → implicit
}

template void f<int>(int); // generates: void f(int) → explicit
```

Template Parameters

Template Parameters are the names following the `template` keyword

```
template<typename T>
void f() {}

f<int>();
```

`typename T` is the **template parameter**

`int` is the **template argument**

A **template parameter** can be a *generic type*, i.e. `typename`, as well as a *non-type template parameters* (NTTP), e.g. `int`, `enum`, etc.

The **template argument** of a *generic type* is a built-in or user-declared type, while a *concrete value* for a *non-type template parameter*

`int` parameter

```
template<int A, int B>
int add_int() {
    return A + B; // sum is computed at compile-time
}                 // e.g. add_int<3, 4>();
```

`enum` parameter

```
enum class Enum { Left, Right };

template<Enum Z>
int add_enum(int a, int b) {
    return (Z == Enum::Left) ? a + b : a;
}        // e.g. add_enum<Enum::Left>(3, 4);
```

- **Ceiling division**

```
template<int DIV, typename T>
T ceil_div(T value) {
    return (value + DIV - 1) / DIV;
}
// e.g. ceil_div<5>(11); // returns 3
```

- **Rounded division**

```
template<int DIV, typename T>
T round_div(T value) {
    return (value + DIV / 2) / DIV;
}
// e.g. round_div<5>(11); // returns 2 (2.2)
```

Since `DIV` is known at compile-time, the compiler can heavily optimize the division (almost for every number, not just for power of two)

C++11 Template parameters can have default values

```
template<int A = 3, int B = 4>
void print1() { cout << A << ", " << B; }

template<int A = 3, int B>                  // still possible, but little sense
void print2() { cout << A << ", " << B; }

print1<2, 5>();  // print 2, 5
print1<2>();     // print 2, 4 (B:   default)
print1<>();      // print 3, 4 (A,B: default)
print1();        // print 3, 4 (A,B: default)

print2<2, 5>();  // print 2, 5
// print2<2>();  compile error
// print2<>();   compile error
// print2();     compile error
```

Template parameters may have no name

```
void f() {}

template<typename = void>
void g() {}

int main() {
    g();  // generated
}
```

`f()` is always generated in the final code

`g()` is generated in the final code only if it is called

C++11 **Unlike function parameters, template parameters can be initialized by previous values**

```
template<int A, int B = A + 3>
void f() {
    cout << B;
}

template<typename T, int S = sizeof(T)>
void g(T) {
    cout << S;
}

f<3>();  // B is 6
g(3);    // S is 4
```

Function Template Overloading

Template Functions can be *overloaded*

```
template<typename T>
T add(T a, T b) {
    return a + b;
}  // e.g add(3, 4);

template<typename T>
T add(T a, T b, T c) { // different number of parameters
    return a + b + c;
}  // e.g add(3, 4, 5);
```

Also, templates themselves can be *overloaded*

```
template<int C, typename T>
T add(T a, T b) {     // it is not in conflict with
    return a + b + C; // T add(T a, T b)
}                     // "C" is part of the signature
```

Template Specialization

Template specialization refers to the concrete implementation for a specific combination of template parameters

The problem:

```
template<typename T>
bool compare(T a, T b) {
    return a < b;
}
```

The direct comparison between two floating-point values is dangerous due to rounding errors

Solution: Template specialization

```
template<>
bool compare<float>(float a, float b) {
    return ...   // a better floating point implementation
}
```

Full Specialization: *Function* templates can be specialized only if **ALL** template arguments are specialized

Template Variable

Template Variable

C++14 allows variables with templates

A template variable can be considered a special case of a *class template* (see next lecture)

```
template<typename T>
constexpr T pi{ 3.1415926535897932385 }; // variable template

template<typename T>
T circular_area(T r) {
    return pi<T> * r * r; // pi<T> is a variable template instantiation
}

circular_area(3.3f); // float
circular_area(3.3);  // double
// circular_area(3); // compile error, narrowing conversion with "pi"
```

Template Parameter Types

Template Parameter Types

Template parameters can be:

- *integral type*
- `enum`, `enum class`
- *floating-point type* C++20
- `auto` placeholder C++17
- *class literals* and *concepts* C++20
- *generic type* `typename`

and rarely:

- *function*
- *reference/pointer* to global `static` function or object
- *pointer to member type*
- `nullptr_t` C++14

Generic Type Notes

Pass multiple values and floating-point types

```
template<float V>  // only in C++20
void print_float() {}

template<typename T>
void print() {
    cout << T::x << ", " << T::y;
}

struct Multi {
    static const     int   x = 1;
    static constexpr float y = 2.0f;
};

print<Multi>(); // print "1, 2"
```

`auto` Placeholder

C++17 introduces automatic deduction of *non-type* template parameters with the `auto` keyword

```
template<int X, int Y>
void f() {}

template<typename T1, T1 X, typename T2, T2 Y>
void g1() {} // before C++17

template<auto X, auto Y>
void g2() {}

f<2u, 2u>();              // X: int, Y: int
g1<int, 2, char, 'a'>(); // X: int, Y: char
g2<2, 'a'>();             // X: int, Y: char
```

Class Template Parameter Type

C++20 A *non-type template parameter* of a **class literal type**:

- A *class literal* is a class that can be assigned to `constexpr` variable
- All *base* classes and *non-static data members* are `public` and `non-mutable`
- All *base* classes and *non-static data members* have the same properties

```
#include <array>
struct A {
    int x;
    constexpr A(int x1) : x{x1} {}
};
template<A a>
void f() { std::cout << a.x; }

template<std::array array>
void g() { std::cout << array[2]; }

f<A{5}>();               // print '5'
g<std::array{1,2,3}>(); // print '3'
```

Array and pointer

```
template<int* ptr>    // pointer
void g() {
    cout << ptr[0];
}

template<int (&array)[3]>  // reference
void f() {
    cout << array[0];
}

int array[] = {2, 3, 4}; // global

int main() {
    f<array>(); // print 2
    g<array>(); // print 2
}
```

Class member

```
struct A {
    int x    = 5;
    int y[3] = {4, 2, 3};
};

template<int A::*x>       // pointer to
void h1() {}              // member type

template<int (A::*y)[3]> // pointer to
void h2() {}              // member type


int main() {
    h1<&A::x>();
    h2<&A::y>();
}
```

Function

```
template<int (*F)(int, int)>  // <-- signature of "f"
int apply1(int a, int b) {
    return F(a, b);
}

int f(int a, int b) { return a + b; }

int g(int a, int b) { return a * b; }

template<decltype(f) F>       // alternative syntax
int apply2(int a, int b) {
    return F(a, b);
}

int main() {
    apply1<f>(2, 3); // return 5
    apply2<g>(2, 3); // return 6
}
```

Compile-Time Utilities

static_assert

C++11 `static_assert` **is used to test an assertion at compile-time**, e.g. `sizeof`, literals, templates, `constexpr`

If the *static assertion* fails, the program does not compile

```
static_assert(2 + 2 == 4, "test1"); // ok, it compiles
static_assert(2 + 2 == 5, "test2"); // compile error, print "test2"
```

C++17: assertions without messages

```
template<typename T, typename R>
void f() { static_assert(sizeof(T) == sizeof(R)); }

f<int, unsigned>(); // ok, it compiles
// f<int, char>();  // compile error
```

C++26: assertions with text formatting

```
static_assert(sizeof(T) != 4, std::format("test1 with sizeof(T)={}", sizeof(T)));
```

using keyword (C++11)

The using keyword introduces an *alias-declaration* or *alias-template*

- `using` is an enhanced version of `typedef` with a more readable syntax
- `using` can be combined with templates, as opposite to `typedef`
- `using` is useful to simplify complex template expression
- `using` allows introducing new names for partial and full specializations

```
typedef int distance_t;  // equal to:

using distance_t = int;
```

```
typedef void (*function)(int, float);  // equal to:

using function = void (*)(int, float);
```

Full/Partial specialization alias:

```
template<typename T, int Size>
struct Vector {};                    // see next lecture for further details
                                     // on class template
template<int Size>
using Bitset = Vector<bool, Size>; // partial specialization alias

using IntV4 = Vector<int, 4>;        // full specialization alias
```

Accessing a type within a structure:

```
struct A {
    using type = int;
};
using Alias = A::type;
```

C++11 decltype keyword captures the type of *entity* or an *expression*

- `decltype` never executes, it is always evaluated at compile-type

```
int       x = 3;
int&      y = x;
const int z = 4;
int       array[2];
void      f(int, float);

decltype(x)          d1; // int
decltype(2 + 3.0)    d2; // double
decltype(y)          d3; // int&
decltype(z)          d4; // const int
decltype(array)      d5; // int[2]
decltype(f(1, 2.0f)) d6; // void

using function = decltype(f);
```

```
bool f(int) { return true; }

struct A {
    int x;
};
int x = 3;
const A a{4};

decltype(x)     d1;     // int
decltype((x))   d2 = x; // int&

decltype(f)     d3;     // bool (int)
decltype((f))   d4 = f; // bool (&)(int)

decltype(a.x)   d5;     // int
decltype((a.x)) d6 = x; // const int
```

www.ibm.com/support/knowledgecenter

C++11

```
template<typename T, typename R>
decltype(T{} + R{}) add(T x, R y) {
    return x + y;
}

unsigned v1 = add(1, 2u);
double   v2 = add(1.5, 2u);
```

C++14

```
template<typename T, typename R>
auto add(T x, R y) {
    return x + y;
}
```

Type Traits

Introspection

Introspection is the ability to inspect a type and query its properties

Reflection

Reflection is the ability of a computer program to examine, introspect, and modify its own structure and behavior

C++ provides compile-time reflection and introspection capabilities through type traits

Type traits (C++11)

Type traits define a compile-time interface to *query* or *modify* the properties of types

The problem:

```
template<typename T>
T integral_div(T a, T b) {
    return a / b;
}

integral_div(7, 2);     // returns 3 (int)
integral_div(71, 21);   // returns 3 (long int)
integral_div(7.0, 3.0); // !!! a floating-point value is not an integral type
```

Two alternatives: (1) Specialize (2) Type Traits + `static_assert`

If we want to **prevent floating-point/other objects division at compile-time**, a first solution consists in specialize for all integral types

```
template<typename T>
T integral_div(T a, T b); // declaration (error for other types)

template<>
char integral_div<char>(char a, char b) {   // specialization
    return a / b;
}
template<>
int  integral_div<int>(int a, int b) {      // specialization
    return a / b;
}
...unsigned char
...short
...
```

The best solution is to use **type traits**

```cpp
#include <type_traits>      // <-- std type traits library
template<typename T>
T integral_div(T a, T b) {
    static_assert(std::is_integral<T>::value,
                  "integral_div accepts only integral types");
    return a / b;
}
```

`std::is_integral<T>` is a `struct` with a `static constexpr` boolean field `value`

`value` is `true` if T is `bool`, `char`, `short`, `int`, `long`, `long long`, `false` otherwise

---

C++17 provides utilities to improve the readability of type traits

```cpp
std::is_integral_v<T>;  // std::is_integral<T>::value
```

- `is_integral` checks for an integral type (`bool` , `char` , `unsigned char` , `short` , `int` , `long` , etc.)
- `is_floating_point` checks for a floating-point type (`float` , `double`)
- `is_arithmetic` checks for a integral or floating-point type
- `is_signed` checks for a signed type (`float` , `int` , etc.)
- `is_unsigned` checks for an unsigned type (`unsigned` , `bool` , etc.)
- `is_enum` checks for an enumerator type (`enum` , `enum class`)
- `is_void` checks for (`void`)
- `is_pointer` checks for a pointer (`T*`)
- `is_null_pointer` checks for a (`nullptr`) C++14

Entity type queries:

- `is_reference` checks for a reference (`T&`)
- `is_array` checks for an array (`T (&)[N]`)
- `is_function` checks for a function type

Class queries:

- `is_class` checks for a class type (`struct` , `class`)
- `is_abstract` checks for a class with at least one pure virtual function
- `is_polymorphic` checks for a class with at least one virtual function

Type property queries:

- `is_const` checks if a type is `const`

Type relation:

- `is_same<T, R>` checks if `T` and `R` are the same type
- `is_base_of<T, R>` checks if `T` is base of `R`
- `is_convertible<T, R>` checks if `T` can be converted to `R`

Full list: `en.cppreference.com/w/cpp/header/type_traits`

Example - const Deduction

```cpp
#include <type_traits>
template<typename T>
void f(T x) {  cout << std::is_const_v<T>;  }

template<typename T>
void g(T& x) {  cout << std::is_const_v<T>;  }

template<typename T>
void h(T& x) {
   cout << std::is_const_v<T>;
   x = nullptr; // ok, it compiles for T: (const int)*
}
const int a = 3;
f(a);  // print false, "const" drop in pass by-value
g(a);  // print true
const int* b = new int;
h(b);  // print false!! T: (const int)*
```

Example - Type Relation

```cpp
#include <type_traits>
template<typename T, typename R>
T add(T a, R b) {
    static_assert(std::is_same_v<T, R>, "T and R must have the same type");
    return a + b;
}
add(1, 2);      // ok
// add(1, 2.0); // compile error, "T and R must have the same type"
```

```cpp
#include <type_traits>
struct A {};
struct B : A {};

std::is_base_of_v<A, B>;           // true
std::is_convertible_v<int, float>; // true
```

Type Manipulation

Type traits allow also to manipulate types by using the `type` field

Example: produce `unsigned` from `int`

```
#include <type_traits>

using U = typename std::make_unsigned<int>::type; // see next lecture to understand
                                                  // why 'typename' is needed here
U y = 5;  // unsigned
```

C++14 provides utilities to improve the readability of type traits

```
std::make_unsigned_t<T>;  // instead of 'typename std::make_unsigned<T>::type'
```

Signed and Unsigned types:

- `make_signed` makes a signed type
- `make_unsigned` makes an unsigned type

Pointers and References:

- `remove_pointer` remove pointer (`T*` → `T`)
- `remove_reference` remove reference (`T&` → `T`)
- `add_pointer` add pointer (`T` → `T*`)
- `add_lvalue_reference` add reference (`T` → `T&`)

`const` **specifiers:**

- `remove_const` remove `const` (`const T` $\rightarrow$ `T`)
- `add_const` add `const`

Other type transformation:

- `common_type<T, R>` returns the common type between `T` and `R`
- `conditional<pred, T, R>` returns `T` if `pred` is `true` , `R` otherwise
- `decay<T>` returns the same type as a function parameter passed by-value

Type Manipulation Example

```cpp
#include <type_traits>
template<typename T>
void f(T ptr) {
    using R = std::remove_pointer_t<T>;
    R x = ptr[0]; // char
}

template<typename T>
void g(T x) {
    using R = std::add_const_t<T>;
    R y = 3;
//  y   = 4;     // compile error
}

char a[] = "abc";
f(a);  // T: char*
g(3);  // T: int
```

std::common_type Example

```cpp
#include <type_traits>

template<typename T, typename R>
std::common_type_t<R, T> // <-- return type
add(T a, R b) {
    return a + b;
}

// we can also use decltype to derive the result type
using result_t = decltype(add(3, 4.0f));
result_t x = add(3, 4.0f);
```

`std::conditional` Example

```cpp
#include <type_traits>

template<typename T, typename R>
auto f(T a, R b) {
    constexpr bool pred = sizeof(T) > sizeof(R);
    using S = std::conditional_t<pred, T, R>;
    return static_cast<S>(a) + static_cast<S>(b);
}

f(   2,  'a'); // return 'int'
f(   2, 2ull); // return 'unsigned long long'
f(2.0f, 2ull); // return 'unsigned long long'
```

Modern C++ Programming

10. Templates and Meta-programming II

Class Templates , Sfinae, and Concepts

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

7 C++20 Concepts

Class Template

Class Template

Similarly to function templates, **class templates** are used to build a family of classes

```
template<typename T>
struct A { // class template (typename template)
    T x = 0;
};
template<int N1>
struct B { // class template (numeric template)
    int N = N1;
};

A<int>    a1; // a1.x is int    x = 0
A<float>  a2; // a2.x is float  x = 0.0f
B<1>      b1; // b1.N is 1
B<2>      b2; // b2.N is 2
```

The *main difference* with template functions is that classes can be **partially** specialized

Note: Every class specialization (both partial and full) is a completely **new class**, and it does not share anything with the generic class

```
template<typename T, typename R>
struct A {};              // generic class template

template<typename T>
struct A<T, int> {};      // partial specialization

template<>
struct A<float, int> {}; // full specialization
```

```
template<typename T, typename R>
struct A {              // GENERIC class template
    T x;
};

template<typename T>
struct A<T, int> {      // PARTIAL specialization
    T y;
};

A<float, float> a1;
a1.x;    // ok, generic template
// a1.y; // compile error

A<float, int> a2;
a2.y;    // ok, partial specialization
// a2.x; // compile error
```

Example 1: Implement a Simple Type Trait

```
template<typename T, typename R> // GENERIC template declaration
struct is_same {
    static constexpr bool value = false;
};

template<typename T>
struct is_same<T, T> {           // PARTIAL template specialization
    static constexpr bool value = true;
};

cout << is_same<  int,  char>::value; // print false, generic template
cout << is_same<float, float>::value; // print true, partial template
```

Example 2: Check if a Pointer is const

```
#include <type_traits>

// std::true_type and std::false_type contain a field "value"
//    set to true or false respectively

template<typename T>
struct is_const_pointer : std::false_type {}; // GENERIC template declaration

template<typename R>                          // PARTIAL specialization
struct is_const_pointer<const R*> : std::true_type {};

cout << is_const_pointer<int*>::value;       // print false, generic template
cout << is_const_pointer<const int*>::value; // print true, partial template
cout << is_const_pointer<int* const>::value; // print false, generic template
```

Example 3: Compare Class Templates

```cpp
#include <type_traits>

template<typename T>
struct A {};

template<typename T, typename R>
struct Compare : std::false_type {};            // GENERIC template declaration

template<typename T, typename R>
struct Compare<A<T>, A<R>> : std::true_type {}; // PARTIAL specialization

cout << Compare<int, float>::value;             // false, generic template
cout << Compare<A<int>, A<int>>::value;         // true, partial template
cout << Compare<A<int>, A<float>>::value;       // true, partial template
```

Class Template Constructor

Class template arguments don't need to be repeated if they are the default ones

```
template<typename T>
struct A {
    A(const A& x); // A(const A<T>& x);

    A f();         // A<T> f();
};
```

Constructor Template Automatic Deduction (CTAD)

Constructor Template Automatic Deduction (CTAD)

C++17 introduces *automatic* deduction of class template arguments in constructor calls

```
template<typename T, typename R>
struct A {
    A(T x, R y) {}
};

A<int, float> a1(3, 4.0f); // < C++17
A             a2(3, 4.0f); // C++17

// A<int> a{3, 5}; compile error, "partial" specialization
```

CTAD - User-Defined Deduction Guides

Template deduction guide is a mechanism to instruct the compiler how to map *constructor parameter types* into *class template parameters*

```
template<typename T>
struct MyString {
    MyString(T) {}
};

// constructor            class instantiation
MyString(char const*) -> MyString<std::string>; // deduction guide

MyString s{"abc"}; // construct 'MyString<std::string>'
```

CTAD - User-Defined Deduction Guides - Aggregate Example

```
template<typename T>
struct A {
    T x, y;
};

template<typename T>
A(T, T) -> A<T>; // deduction guide
                 // not required in C++20+ for aggregates

A a{1, 3};       // construct 'A<int, int>'
```

CTAD - User-Defined Deduction Guides - Independent Argument Example

```cpp
template<int I>
struct A {
    template<typename T>
    A(T) {}
};

template<typename T>
A(T) -> A<sizeof(T)>; // deduction guide

A a{1};               // construct 'A<4>', 4 == sizeof(int)
```

CTAD - User-Defined Deduction Guides - Universal Reference Example

```cpp
#include <type_traits> // std::remove_reference_t

template<typename T>
struct A {
    template<typename R>
    A(R&&) {}
};

template<typename R>
A(R&&) -> A<std::remove_reference_t<R>>; // deduction guide

int x;
A a{x}; // construct 'A<int>' instead of 'A<int&>'
```

CTAD - User-Defined Deduction Guides - Iterator Example

```cpp
#include <type_traits> // std::remove_reference_t
#include <vector>      // std::vector

template<typename T>
struct Container {
    template<typename Iter>
    Container(Iter beg, Iter end) {}
};

template<typename Iter>
Container(Iter b, Iter e) ->       // deduction guide
    Container<typename std::iterator_traits<Iter>::value_type>;

std::vector v{1, 2, 3};
Container   c{v.begin(), v.end()}; // construct 'Container<int>'
```

CTAD - User-Defined Deduction Guides - Alias Template

Alias template deduction requires C++20

```
template<typename T>
struct A {
   A(T) {}
};

template<typename T>
A(T) -> A<int>;          // deduction guide

template<typename T>
using B = A<T>;          // alias template

B c{3.0};                // alias template deduction
                         // construct 'A<int>'
```

CTAD User-Defined Deduction Guides - Limitation

Template deduction guide doesn't work within the class scope

```
template<typename T>
struct MyString {
    MyString(T) {}
    MyString f() { return MyString("abc"); } // create 'MyString<const char*>'
};                                           // not 'MyString<std::string>'
MyString(const char*) -> MyString<std::string>; // deduction guide

MyString<const char*> s{"abc"}; // construct 'MyString<const char*>'
```

The problem can be avoided by using a factory

```
template<typename T>
auto make_my_string(const T& x) { return MyString(x); }
```

Class Template - Advanced Concepts

Given a class template and a template member function

```
template<typename T, typename R>
struct A {
    template<typename X, typename Y>
    void f();
};
```

There are two ways to specialize the class/function:

- **Generic class + generic function**
- **Full class specialization + generic/full specialization function**

```
template<typename T, typename R>
template<typename X, typename Y>
void A<T, R>::f() {}
// ok, A<T, R> and f<X, Y> are not specialized

template<>
template<typename X, typename Y>
void A<int, int>::f() {}
//  ok, A<int, int> is full specialized
//  ok, f<X, Y> is not specialized

template<>
template<>
void A<int, int>::f<int, int>() {}
//  ok, A<int, int> and f<int, int> are full specialized
```

```
template<typename T>
template<typename X, typename Y>
void A<T, int>::f() {}
// error A<T, int> is partially specialized
//     (A<T, int> class must be defined before)

template<typename T, typename R>
template<typename X>
void A<T, R>::f<int, X>() {}
// error function members cannot be partially specialized

template<typename T, typename R>
template<>
void A<T, R>::f<int, int>() {}
// error function members of a non-specialized class cannot be specialized
//     (requires a binding to a specific template instantiation at compile-time)
```

Structure templates can have different data members for each specialization.
The compiler needs to know in advance if a symbol within a structure is a type or a static member when the structure template *depends on* another template parameter

The keyword `typename` placed before a *structure template* solves this ambiguous

```
template<typename T>
struct A {
    using type = int;
};

template<typename R>
void g() {
    using X = typename A<R>::type; // "type" is a typename or
}                                  // a data member depending on R
```

The `using` keyword can be used to simply the expression to get the structure type

```
template<typename T>
struct A {
    using type = int;
};

template<typename T>
using AType = typename A<T>::type;

template<typename R>
void g() {
    using X = AType<R>;
}
```

Template Dependent Names - `template` Keyword

The `template` keyword tells the compiler that what follows is a *template name* (*function* or *class*)

note: some recent compilers don't strictly require this keyword in simple cases

```
template<typename T>
struct A {
    template<typename R>
    void g() {}
};

template<typename T>   // A<T> is a dependent name (from T)
void f(A<T> a) {
//  a.g<int>(); // compile error g<int> is a dependent name (from int)
                // interpreted as: "(a.g < int) > ()"
    a.template g<int>(); // ok
}
```

Class Template Hierarchy and `using`

Member of class templates can be used *internally* in derived class templates by specifying the particular type of the base class with the keyword `using`

```
template<typename T>
struct A {
    T    x;
    void f() {}
};

template<typename T>
struct B : A<T> {
    using A<T>::x;  // needed (otherwise it could be another specialization)
    using A<T>::f;  // needed

    void g() {
        x;   // without 'using': this->x
        f();
    }
};
```

`virtual` Function and Template

Virtual functions cannot have template arguments

- **Templates** are a compile-time feature
- **Virtual functions** are a run-time feature

Full story:

The reason for the language disallowing the particular construct is that there are potentially infinite different types that could be instantiating your template member function, and that in turn means that the compiler would have to generate code to dynamically dispatch those many types, which is infeasible

`stackoverflow.com/a/79682130`

`friend` Keyword

```
template<typename T>              struct A {};
template<typename T, typename R> struct B {};
template<typename T>              void f() {}
//-------------------------------------------------------------------------------
class C {
    friend void f<int>();                 // match only f<int>

    template<typename T> friend void f(); // match all templates

    friend struct A<int>;                 // match only A<int>

    template<typename> friend struct A;   // match all A templates

//  template<typename T> friend struct B<int, T>;
//      partial specialization cannot be declared as a friend
};
```

Template Template Arguments

Template template parameters match *templates* instead of concrete types

```
template<typename T> struct A {};

template< template<typename> class R >
struct B {
    R<int>   x;
    R<float> y;
};
template< template<typename> class R, typename S >
void f(R<S> x) {}  // works with every class with exactly one template parameter

B<A> y;
f( A<int>() );
```

`class` and `typename` keyword are interchangeably in C++17

Template Meta-Programming

Template Meta-Programming

"Metaprogramming is the writing of computer programs with the ability to ***treat programs as their data****. It means that a program could be designed to read, generate, analyze or transform other programs, and even modify itself while running"*

"Template meta-programming refers to uses of the C++ template system to ***perform computation at compile-time*** *within the code. Templates meta-programming include compile-time constants, data structures, and complete functions"*

Template Meta-Programming

- **Template Meta-Programming is fast** (runtime)
  Template Metaprogramming is computed at compile-time (nothing is computed at run-time)
- **Template Meta-Programming is Turing Complete**
  Template Metaprogramming is capable of expressing all tasks that standard programming language can accomplish
- **Template Meta-Programming requires longer compile time**
  Template recursion heavily slows down the compile time, and requires much more memory than compiling standard code
- **Template Meta-Programming is complex**
  Everything is expressed recursively. Hard to read, hard to write, and also very hard to debug

Example 1: Factorial

```
template<int N>
struct Factorial {      // GENERIC template: Recursive step
    static constexpr int value = N * Factorial<N - 1>::value;
};

template<>
struct Factorial<0> {   // FULL SPECIALIZATION: Base case
    static constexpr int value = 1;
};

constexpr int x = Factorial<5>::value; // 120
// int y = Factorial<-1>::value;       // Infinite recursion :)
```

Example 1: `Factorial` (Notes)

The previous example can be easily written as a `constexpr` in C++14

```
template<typename T>
constexpr int factorial(T value) {
    T tmp = 1;
    for (int i = 2; i <= value; i++)
        tmp *= i;
    return tmp;
};
```

Advantages:

- Easy to read and write (easy to debug)
- Faster compile time (no recursion)
- Works with different types (`typename T`)
- Works at run-time *and* compile-time

Example 2: Log2

```
template<int N>
struct Log2 {    // GENERIC template: Recursive step
    static_assert(N > 0, "N must be greater than zero");

    static constexpr int value = 1 + Log2<N / 2>::value;
};

template<>
struct Log2<1> { // FULL SPECIALIZATION: Base case
    static constexpr int value = 0;
};

constexpr int x = Log2<20>::value; // 4
```

Example 3: Log

```
template<int A, int B>
struct Max { // utility
    static constexpr int value = A > B ? A : B;
};

template<int N, int BASE>
struct Log {      // GENERIC template: Recursive step
    static_assert(N > 0,    "N must be greater than zero");
    static_assert(BASE > 0, "BASE must be greater than zero");
                            // Max is used to avoid Log<0, BASE>
    static constexpr int TMP   = Max<1, N / BASE>::value;
    static constexpr int value = 1 + Log<TMP, BASE>::value;
};
template<int BASE>
struct Log<1, BASE> {   // PARTIAL SPECIALIZATION: Base case
    static constexpr int value = 0;
};

constexpr int x = Log<20, 2>::value; // 4
```

Example 4: `Unroll` (Compile-time/Run-time Mix) ★

```cpp
template<int NUM_UNROLL, int STEP = 0>
struct Unroll {            // GENERIC template: Recursive step
    template<typename Op>
    static void run(Op op) {
        op(STEP);
        Unroll<NUM_UNROLL, STEP + 1>::run(op);
    }
};

template<int NUM_UNROLL>
struct Unroll<NUM_UNROLL, NUM_UNROLL> { // PARTIAL SPECIALIZATION: Base case
    template<typename Op>
    static void run(Op) {}
};

auto lambda = [](int step) { cout << step << ", "; };
Unroll<5>::run(lambda);  // print "0, 1, 2, 3, 4"
```

SFINAE: Substitution Failure Is Not An Error

SFINAE

Substitution Failure Is Not An Error (SFINAE) applies during overload resolution of function templates. When substituting the deduced type for the template parameter fails, the specialization is discarded from the overload set *instead* of causing a compile error

The Problem

```
template<typename T>
T ceil_div(T value, T div);

template<>
unsigned ceil_div<unsigned>(unsigned value, unsigned div) {
    return (value + div - 1) / div;
}

template<>
int ceil_div<int>(int value, int div) { // handle negative values
    return (value > 0) ^ (div > 0) ?
           (value / div) : (value + div - 1) / div;
}
```

What about `long long int`, `long long unsigned`, `short`, `unsigned short`, etc.?

`std::enable_if` Type Trait

The common way to adopt `SFINAE` is using the `std::enable_if/std::enable_if_t` type traits

`std::enable_if` allows a function template or a class template specialization to include or exclude itself from a set of matching functions/classes

```
template<bool Condition, typename T = void>
struct enable_if {
//  "type" is not defined if "Condition == false"
};
template<typename T>
struct enable_if<true, T> {
    using type = T;
};
```

helper alias: `std::enable_if_t<T>` instead of `typename std::enable_if<T>::type`

```cpp
#include <type_traits> // std::is_signed_v, std::enable_if_t

template<typename T>
std::enable_if_t<std::is_signed_v<T>>
f(T) {
    cout << "signed";
}

template<typename T>
std::enable_if_t<!std::is_signed_v<T>>
f(T) {
    cout << "unsigned";
}

f(1);  // print "signed"
f(1u); // print "unsigned"
```

```cpp
#include <type_traits>

template<typename T>
void f(std::enable_if_t<std::is_signed_v<T>, T>) {
    cout << "signed";
}

template<typename T>
void f(std::enable_if_t<!std::is_signed_v<T>, T>) {
    cout << "unsigned";
}

f(1);  // print "signed"
f(1u); // print "unsigned"
```

```cpp
#include <type_traits>

template<typename T>
void f(T,
       std::enable_if_t<std::is_signed_v<T>, int> = 0) {
    cout << "signed";
}

template<typename T>
void f(T,
       std::enable_if_t<!std::is_signed_v<T>, int> = 0) {
    cout << "unsigned";
}

f(1);  // print "signed"
f(1u); // print "unsigned"
```

```cpp
#include <type_traits>

template<typename T,
         std::enable_if_t<std::is_signed_v<T>, int> = 0>
void f(T) {}

template<typename T,
         std::enable_if_t<!std::is_signed_v<T>, int> = 0>
void f(T) {}

f(4);
f(4u);
```

```cpp
#include <type_traits>
template<typename T, typename R>          // (1)
decltype(T{} + R{}) add(T a, R b) {       // T{} + R{} is not possible with 'A'
    return a + b;
}

template<typename T, typename R>          // (2)
std::enable_if_t<std::is_class_v<T>, T> // 'int' is not a class
add(T a, R b) {
    return a;
}

struct A {};

add(1, 2u);    // call (1)
add(A{}, A{}); // call (2)
// if 'A' supports operator+, then we have a conflict
```

Function SFINAE Example - Array vs. Pointer

```cpp
#include <type_traits>

template<typename T, int Size>
void f(T (&array)[Size]) {} // (1)

//template<typename T, int Size>
//void f(T* array) {}       // (2)

template<typename T>
std::enable_if_t<std::is_pointer_v<T>>
f(T ptr) {}                 // (3)

int array[3];
f(array); // It is not possible to call (1) if (2) is present
          // The reason is that 'array' decays to a pointer
          // Now with (3), the code calls (1)
```

Function SFINAE Notes

The wrong way to achieve SFINAE

```
template<typename T, typename = std::enable_if_t<std::is_signed_v<T>>
void f(T) {}

// template<typename T, typename = std::enable_if_t<!std::is_signed_v<T>>
// void f(T) {}
// compile error redefinition of the second template parameter
```

Using `std::enable_if_t` for the *return type* prevents `auto` deduction

```
// template<typename T>
// std::enable_if_t<std::is_signed_v<T>, auto> f(T) {}
// compile error auto is not allowed here
```

Class SFINAE

```cpp
#include <type_traits>

template<typename T, typename Enable = void>
struct A;

template<typename T>
struct A<T, std::enable_if_t<std::is_signed_v<T>>> {
};

template<typename T>
struct A<T, std::enable_if_t<!std::is_signed_v<T>>> {
};

A<int>      a1;
A<unsigned> a2;
```

SFINAE can be also used to check if a structure has a specific data member or type

Let consider the following structures:

```
struct A {
    static int x;
    int        y;
    using type = int;
};

struct B {};
```

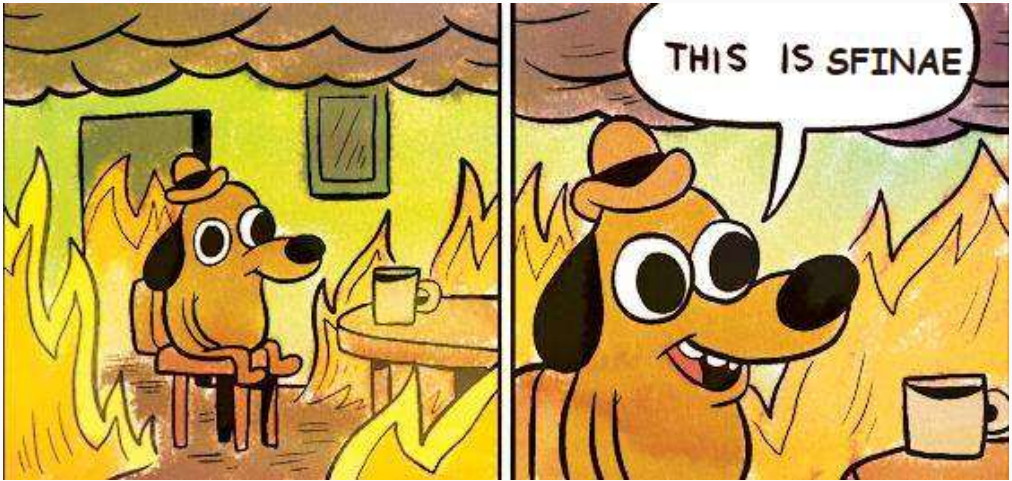
```cpp
#include <type_traits>
template<typename T, typename = void>
struct has_x : std::false_type {};

template<typename T>
struct has_x<T, decltype((void) T::x)> : std::true_type {};

template<typename T, typename = void>
struct has_y : std::false_type {};

template<typename T>
struct has_y<T, decltype((void) std::declval<T>().y)> : std::true_type {};

has_x< A >::value; // returns true
has_x< B >::value; // returns false
has_y< A >::value; // returns true
has_y< B >::value; // returns false
```

```cpp
template<typename...>
using void_t = void;  // included in C++17 <utility>

template<typename T, typename = void>
struct has_type : std::false_type {};

template<typename T>
struct has_type<T,
                std::void_t<typename T::type> > : std::true_type {};

has_type< A >::value; // returns true
has_type< B >::value; // returns false
```

Support Trait for Stream Operator ★

```cpp
template<typename T>
using EnableP = decltype( std::declval<std::ostream&>() <<
                          std::declval<T>() );

template<typename T, typename = void>
struct is_stream_supported : std::false_type {};

template<typename T>
struct is_stream_supported<T, EnableP<T>> : std::true_type {};

struct A {};

is_stream_supported<int>::value; // returns true
is_stream_supported<A>::value;   // returns false
```

SFINAE



https://twitter.com/IAmErikN/status/1252316405724336128

Variadic Templates

Variadic template (C++11)

A **variadic template** captures a *parameter pack* of arguments, which hold an arbitrary number of values or types

```
template<typename... TArgs> // Variadic typename -> parameter pack: ... TArgs
void f(TArgs... args) {}    // pack expansion    -> pattern:        TArgs
```

A **parameter pack** is introduced by an identifier `TArgs` prefixed by an *ellipsis* `... TArgs`. Once captured, a *parameter pack* can later be used in a *pattern* expanded by an *ellipsis* `...`

A **pack expansion** is equivalent to a comma-separated list of instances of the *pattern*

A **pattern** is a *set of tokens* containing the identifiers of one or more *parameter packs*. When a *pattern* contains more than one *parameter pack*, all packs must have the same length

```
template<typename... TArgs>
void f(TArgs... args) {        // Typename expansion
    int values[] = {args...}; // Arguments expansion
}
f(1, 2, 3);
```

The pack `TArgs` expands in a *template-argument-list*, i.e. list of template arguments
The pack `args` expands in an *initializer-list*, i.e. list of values

The number of variadic arguments can be retrieved with the `sizeof...` operator

```
sizeof...(args) // e.g. 3
```

Note: variadic arguments must be the last one in the declaration

Example 1

```cpp
// BASE CASE
template<typename T, typename R>
auto add(T a, R b) {
    return a + b;
}

// RECURSIVE CASE
template<typename T, typename... TArgs>  // Variadic typename
auto add(T a, TArgs... args) {           // Typename expansion
    return a + add(args...);             // Arguments expansion
}

add(2, 3.0);       // 5
add(2, 3.0, 4);    // 9
add(2, 3.0, 4, 5); // 14
// add(2);         // compile error the base case accepts only two arguments
```

Example 2 - Function Unpack

```cpp
template<typename T, typename... TArgs>
auto add(T a, TArgs... args); // see previous slides

struct A {
    int v;
    int f() { return v; }
};

template<typename... TArgs>
int f(TArgs... args) {
    return add(args.f()...);  // equivalent to: 'A{1}.f(), A{2}.f(), A{3}.f()'
}

f(A{1}, A{2}, A{3}); // return 6
```

Example 3 - Function Application

```
template<typename T, typename... TArgs>
auto add(T a, TArgs... args); // see previous slides

template<typename T>
T square(T value) { return value * value; }

//------------------------------------------------------------

template<typename... TArgs>
auto add_square(TArgs... args) {
    return add(square(args)...); // square() is applied to each
}                                // variadic argument

add_square(2, 2, 3.0f); // returns 17.0f
```

Example 4 - Type Expansion

```
template<typename... TArgs>
int g(TArgs... args) {}

template<typename... TArgs>
int f(TArgs... args) {
   g<std::make_unsigned_t<TArgs>...>(args...);
}

f(1, 2, 3);
```

Function Initializer List Types

```
template<typename... TArgs>
void f(TArgs... args) {}            // pass by-value

template<typename... TArgs>
void g(const TArgs&... args) {}     // pass by-const reference

template<typename... TArgs>
void h(TArgs*... args) {}           // pass by-pointer

template<int... Sizes>
void l(int (&...arrays)[Sizes]) {} // pass a list of array references

int a[] = {1, 2};
int b[] = {1, 2, 3};
f(1, 2.0);
h(a, b);
l(a, b); // same as g()
```

Other Notes

Parameter pack can be also used to create a **homogeneous variadic function parameters**

```
template<int... IntSeq>
void f(IntSeq... seq) {} // accepts only integers
```

Variadic templates can be also applied to lambdas with *generic parameters* (C++14) and *concepts* (C++20)

```
auto lambda = [](auto... args) {};

void f(std::floating_point auto... args) {}
```

Advanced Usages ★

Besides *initializer-lists*, *template-argument-list*, parameter pack can be used in: *capture list*, *constructor initializer-list*, `using` *declaration*

```
template<typename... BaseClasses>
struct A : BaseClasses... {          // : BaseClass_1, BaseClass_2, ...
    A(int v) : BaseClasses...{v} {} // BaseClass_1{v},  BaseClass_2{v}, ...

    using BaseClasses::f;
//  equivalent to:
//    using BaseClass_1::f;
//    using BaseClass_2::f;
//    ...
};

void f(auto... args) {
    auto lambda = [arg&...](){}; // capture by-reference
}
```

C++17 **Folding expressions** perform a *fold* of a template parameter pack over any binary operator in C++ (`+` , `*` , `,` , `+=` , `&&` , `<=` etc.)

Unary/Binary folding

```cpp
template<typename... Args>
auto add_unary(Args... args) {  // Unary folding
    return (... + args);        // unfold: 1 + 2.0f + 3ull
}

template<typename... Args>
auto add_binary(Args... args) { // Binary folding
    return (1 + ... + args);    // unfold: 1 + 1 + 2.0f + 3ull
}

add_unary(1, 2.0f, 3ll);  // returns 6.0f (float)
add_binary(1, 2.0f, 3ll); // returns 7.0f (float)
```

Example 1 - Extract The Last Argument

```
template<typename... TArgs>
int f(TArgs... args) {
    return (args, ...); // the comma operator discards left values
}                       // same as (..., args)

f(1, 2, 3); // return 3
```

Example 2 - Function Application

Same example of “Variadic Template - Function Application” ... but shorter

```
template<typename T>
T square(T value) { return value * value; }

template<typename... TArgs>
auto add_square(TArgs... args) {
    return (square(args) + ...); // square() is applied to each
}                                // variadic argument

add_square(2, 2, 3.0f); // returns 17.0f
```

Variadic Template and Classes

```
template<int... NArgs>
struct Add;                  // data structure declaration

template<int N1, int N2>
struct Add<N1, N2> {         // BASE case
    static constexpr int value = N1 + N2;
};

template<int N1, int... NArgs>
struct Add<N1, NArgs...> {  // RECURSIVE case
    static constexpr int value = N1 + Add<NArgs...>::value;
};

Add<2, 3, 4>::value; // returns 9
// Add<>;            // compile error no match
// Add<2>::value;    // compile error
                     // call Add<N1, NArgs...>, then Add<>
```

Variadic Class Template ★

Variadic Template can be used to build recursive data structures

```
template<typename... TArgs>
struct Tuple;       // data structure declaration

template<typename T>
struct Tuple<T> {  // base case
    T value;       // specialization with one parameter
};

template<typename T, typename... TArgs>
struct Tuple<T, TArgs...> {  // recursive case
    T               value;   // specialization with more
    Tuple<TArgs...> tail;    // than one parameter
};

Tuple<int, float, char> t1 { 2, 2.0, 'a' };
t1.value;           // 2
t1.tail.value;      // 2.0
t1.tail.tail.value; // 'a'
```

Get function arity at compile-time:

```
template <typename T>
struct GetArity;

// generic function pointer
template<typename R, typename... Args>
struct GetArity<R(*)(Args...)> {
    static constexpr int value = sizeof...(Args);
};
// generic function reference
template<typename R, typename... Args>
struct GetArity<R(&)(Args...)> {
    static constexpr int value = sizeof...(Args);
};
// generic function object
template<typename R, typename... Args>
struct GetArity<R(Args...)> {
    static constexpr int value = sizeof...(Args);
};
```

```
void f(int, char, double) {}

int main() {
    // function object
    GetArity<decltype(f)>::value;

    auto& g = f;
    // function reference
    GetArity<decltype(g)>::value;

    // function reference
    GetArity<decltype((f))>::value;

    auto* h = f;
    // function pointer
    GetArity<decltype(h)>::value;
}
```

stackoverflow.com/a/27867127

Get `operator()` (and `lambda`) arity at compile-time:

```
template <typename T>
struct GetArity;

template<typename R, typename C, typename... Args>
struct GetArity<R(C::*)(Args...)> {         // class member
    static constexpr int value = sizeof...(Args);
};
template<typename R, typename C, typename... Args>
struct GetArity<R(C::*)(Args...) const> {  // "const" class member
    static constexpr int value = sizeof...(Args);
};

struct A {
    void operator()(char, char) {}
    void operator()(char, char) const {}
};

GetArity<A>::value;       // call GetArity<R(C::*)(Args...)>
GetArity<const A>::value; // call GetArity<R(C::*)(Args...) const>
```

C++20 Concepts

C++20 Concepts

C++20 introduces **concepts** as an extension for *templates* to enforce *constraints*, which specifies the *requirements* on template arguments

Concepts allows performing compile-time validation of template arguments

Advantages compared to SFINAE (`std::enable_if`):

- Concepts are easier to read and write
- Clear compile-time messages for debugging
- Faster compile time

Keyword:

`concept` Constrain

`requires` Constrain list/Requirements, *clause* and *expression*

- `The concept behind C++ concepts`
- `Constraints and concepts`
- `What are C++20 concepts and constraints? How to use them?`

The Problem

Goal: define a function to sum only arithmetic types

```
template<typename T>
T add(T valueA, T valueB) {
    return valueA + valueB;
}
struct A {};

add(3, 4);         // ok
// add(A{}, A{}); // not supported
```

SFINAE solution (ugly, verbose):

```
template<typename T>
std::enable_if_t<T, std::is_arithmetic_v<T>>
add(T valueA, T valueB) {
    return valueA + valueB;
}
```

`concept` Keyword

```
[template arguments]
concept [name] = [compile-time boolean expression];
```

Example: arithmetic type concept

```
template<typename T>
concept Arithmetic = std::is_arithmetic_v<T>;
```

- *Template argument constrain*

```
template<Arithmetic T>
T add(T valueA, T valueB) {
    return valueA + valueB;
}
```

- `auto` *deduction constrain* (*constrained* `auto`)

```
auto add(Arithmetic auto valueA, Arithmetic auto valueB) {
    return valueA + valueB;
}
```

`requires` Clause

```
requires [compile-time boolean expression or Concept]
```

it acts like SFINAE

- After *template parameter list*

```
template<typename T>
requires Arithmetic<T>
T add(T valueA, T valueB) {
    return valueA + valueB;
}
```

- After *function declaration*

```
template<typename T>
T add(T valueA, T valueB) requires (sizeof(T) == 4) {
    return valueA + valueB;
}
```

`requires` Clause and `concept` Notes

Concepts and *requirements* can have *multiple* statements. It must be a *primary expression*, e.g. `constexpr` value (not a `constexpr` function) or a sequence of *primary expressions* joined with the operator `&&` or `||`

```
template<typename T>
concept Arithmetic2 = std::is_arithmetic_v<T> && sizeof(T) >= 4;
```

Concepts and *requirements* can be used together

```
template<Arithmetic T>
requires (sizeof(T) >= 4)
T add(T valueA, T valueB) {
```

A `requires` **expression** is a *compile-time* expression of type `bool` that defines the **constraints** on template arguments

```
requires [(arguments)] {
    [SFINAE contrain];    // or
    requires [predicate];
} -> bool
```

```
template<typename T>
concept MyConcept = requires (T a, T b) { // First case: SFINAE constrains
    a + b;            // Req. 1 - support add operator
    a[0];             // Req. 2 - support subscript operator
    a.x;              // Req. 3 - has "x" data member
    a.f();            // Req. 4 - has "f" function member
    typename T::type; // Req. 5 - has "type" field
};
```

Concept library

```cpp
#include <concept>

template<typename T>
concept MyConcept2 = requires (T a, T b) {
    {*a + 1} -> std::convertible_to<float>; // Req. 6 - can be deferred and the sum
                                           // with an integer is convertible
                                           // to float
    {a * a}  -> std::same_as<int>;         // Req. 7 - "a * a" must be valid and
                                           // the result type is "int"
};
```

`requires` Expression + Clause

`requires` **expression** can be combined with `requires` **clause** (see `requires` definition, second case) to compute a boolean value starting from SFINAE expressions

```
template<typename T>
concept Arithmetic = requires {       // expression -> bool (zero args)
    T::value;                         // clause     -> direct SFINAE
    requires std::is_arithmetic_v<T>; // clause     -> SFINAE from boolean
};
```

```
template<typename T>
concept MyConcept = requires (T value) {  // expression -> bool  (one arg)
    requires sizeof(value) >= 4;          // clause     -> SFINAE from boolean
    requires std::is_floating_point_v<T>; // clause     -> SFINAE from boolean
};
```

`requires` Clause + Expression

`requires` **clause** can be combined with `requires` **expression** to apply SFINAE (functions, structures) starting from a compile-time *boolean expressions*

```
template<typename T>
void f(T a) requires requires { T::value; }
//          clause -> SFINAE followed by
//                    expression -> bool (zero args)
{}
```

```
template<typename T>
T increment(T a) requires requires (T x) { x + 1; }
//                clause -> SFINAE followed by
//                          expression -> bool (one arg)
{
   return a + 1;
}
```

requires and constexpr

Some examples:

- ```
  constexpr bool has_member_x = requires(T v){ v.x; };
  ```
- ```
  if constexpr (MyConcept<T>)
  ```
- ```
  static_assert(requires(T v){ ++v; }, "no increment");
  ```
- ```
  template<typename Iter>
  constexpr bool is_iterator() {
      return requires(Iter it) { *it++; };
  }
  ```

Nested `requires`

Nested `requires` example:

```
requires(Iter v) { // expression -> bool (one arg)
    Iter it;
    requires requires(typename Iter::value_type v) {
//  clause -> SFINAE followed by
//            expression -> bool (one arg)
        v   = *it; // read
        *it = v;   // write
    };
}
```

Modern C++ Programming

11. Translation Units I

Linkage and One Definition Rule

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Basic Concepts

Translation Unit

Header File and Source File

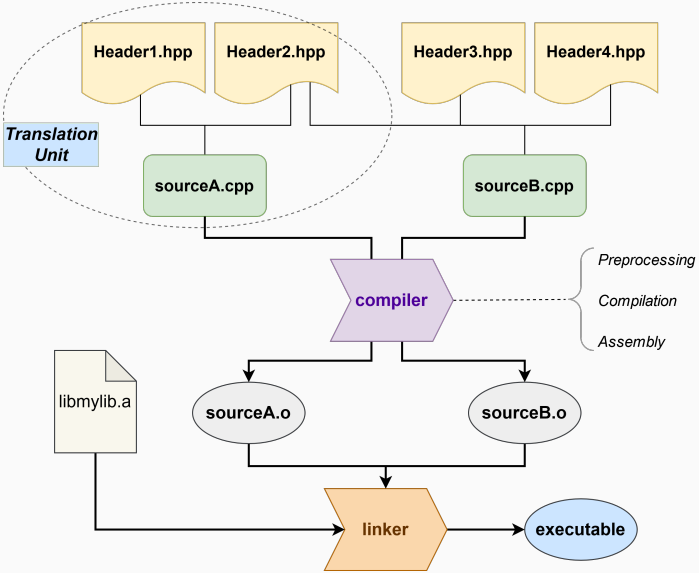
Header files allow defining interfaces (`.h`, `.hpp`, `.hxx`), while keeping the implementation in separated **source files** (`.c`, `.cpp`, `.cxx`).

Translation Unit

A **translation unit** (or *compilation unit*) is the basic unit of compilation in C++. It consists of the content of a *single* source file, plus the content of *any* header file directly or indirectly included by it

A single translation unit can be compiled into an object file, library, or executable program

Compile Process



Local and Global Scope

Scope

The **scope** of a variable/function/object is the region of the code within the entity can be accessed

Local Scope / Block Scope

Entities that are declared inside a function or a block are called local variables.
Their memory address is not valid outside their scope

Global Scope / File Scope / Namespace Scope

Entities that are defined outside all functions.
They hold a single memory location throughout the life-time of the program

Local and Global Scope

```
int var1;      // global scope

int f() {
    int var2; // local scope
}

struct A {
    int var3; // depends on where the instance of 'A' is used
};
```

Linkage

Linkage

Linkage refers to the *visibility* of symbols to the linker

No Linkage

No linkage refers to symbols in the local scope of declaration and not visible to the linker

Internal Linkage

Internal linkage refers to symbols visible only in scope of a *single* translation unit. The same symbol name has a different memory address in distinct translation units

External Linkage

External linkage refers to entities that exist (visible/accessible) *outside* a single translation unit. They are accessible and have the same *identical memory address* through the whole program, which is the combination of all translation units

Storage Class and Duration

Storage Duration

The **storage duration** (or *duration class*) determines the *duration* of a variable, namely when it is created and destroyed

Storage Duration	Allocation	Deallocation
Automatic	Code block start	Code block end
Static	Program start	Program end
Dynamic	Memory allocation	Memory deallocation
Thread	Thread start	Thread end

en.cppreference.com/w/cpp/language/storage_duration

- **Automatic storage duration**. Local variables temporary allocated on registers or stack (depending on compiler, architecture, etc.).
  If not explicitly initialized, their value is undefined
- **Static storage duration**. The storage of an object is allocated when the program begins and deallocated when the program ends.
  If not explicitly initialized, it is zero-initialized
- **Dynamic storage duration**. The object is allocated and deallocated by using dynamic memory allocation functions (`new/delete`).
  If not explicitly initialized, its memory content is undefined
- **Thread storage duration** C++11. The object is allocated when the thread begins and deallocated when the thread ends. Each thread has its own instance of the object

Storage Duration Examples

```
int v1; // static duration

void f() {
    int  v2;                    // automatic duration
    auto v3    = 3;             // automatic duration
    auto array = new int[10]; // dynamic duration (allocation)
} // array, v2, v3 variables deallocation (from stack)
  // the memory associated to "array" is not deallocated

int main() {
    f();
}
// main end: v1 is deallocated
```

Storage Class

Storage Class Specifier

The **storage class** for a variable declaration is a **type specifier** that, *together with the scope*, governs its *storage duration* and *linkage*

Storage Class	Notes	Scope	Storage Duration	Linkage
`auto`	local `var` decl.	Local	*automatic*	*No linkage*
no storage class	global `var` decl.	Global	*static*	*External*
`static`		Local	*static*	*Function Dependent*
`static`		Global	*static*	*Internal*
`extern`		Global	*static*	*External*
`thread_local`	C++11	any	*thread local*	*any*

Storage Class Examples

```
int                     v1;     // no storage class
static              int v2 = 2; // static storage class
extern              int v3;     // external storage class
thread_local        int v4;     // thread local storage class
thread_local static int v5;     // thread local and static storage classes

int main() {
    int              v6;        // auto storage class
    auto             v7 = 3;    // auto storage class
    static int       v8;        // static storage class
    thread_local int v9;        // thread local and auto storage classes
    auto array = new int[10];   // auto storage class ("array" variable)
}
```

Local `static` Variables

`static` *local variables* are allocated when the program begins, *initialized* when the function is called the first time, and deallocated when the program ends

```
int f() {
    static int val = 1;
    val++;
    return val;
}

int main() {
    cout << f(); // print 2 ("val" is initialized)
    cout << f(); // print 3
    cout << f(); // print 4
}
```

`static` and `extern` Keywords

`static` / *anonymous namespace-included global variables* or *functions* are visible only within the file → *internal linkage*

- **Non-** `static` global variables or functions with the same name in different translation units produce *name collision* (or name conflict)

`extern` keyword is used to declare the existence of *global variables* or *functions* in another translation unit → *external linkage*

- the variable or function must be defined in one and only one translation unit
- it is redundant for functions
- it is necessary for variables to prevent the compiler to associate a memory location in the current translation unit

If the same identifier within a translation unit appears with both *internal* and *external* linkage, the behavior is undefined

Internal/External Linkage Examples

```
int         var1 = 3;  // external linkage
                       // (in conflict with variables in other
                       //  translation units with the same name)
static int var2 = 4;   // internal linkage (visible only in the
                       //                   current translation unit)
extern int var3;       // external linkage
                       // (implemented in another translation unit)
void        f1() {}    // external linkage (could conflict)

static void f2() {}    // internal linkage

namespace {            // anonymous namespace
void        f3() {}    // internal linkage
}
extern void f4();      // external linkage
                       // (implemented in another translation unit)
```

Linkage of `const` and `constexpr` Variables

Linkage of `const` and `constexpr` Variables

`const` variables have *internal linkage* at global scope

`constexpr` variables imply `const`, which implies *internal linkage*

note: the same variable has different memory addresses on different translation units (code bloat)

```
const     int var1 = 3;         // internal linkage
constexpr int var2 = 2;         // internal linkage

static const     int var3 = 3; // internal linkage (redundant)
static constexpr int var4 = 2; // internal linkage (redundant)

int main() {}
```

In C++, the order in which global variables are initialized at runtime is not defined. This introduces a subtle problem called *static initialization order fiasco*

source.cpp

```cpp
int f() { return 3; } // run-time function

int x = f();          // run-time evalutation
```

main.cpp

```cpp
extern int x;
int        y = x; // run-time initialized

int main() {
    cout << y;    // print "3" or "0" depending on the linking order
}
```

source.cpp

```cpp
constexpr int f() { return 3; } // compile-time/run-time function

constinit int x = f();          // compile-time initialized (C++20)
```

main.cpp

```cpp
constinit extern int x;     // compile-time initialized (C++20)
int                  y = x; // run-time     initialized

int main() {
    cout << y; // print "3"!!
}
```

Linkage Summary

No Linkage: *Local* variables, functions, classes

- `static` local variable address depends on the linkage of its function

Internal Linkage:

(not accessible by other translation units, no conflicts, different memory addresses)

- **Global Variables**:
  - `static`
  - *non-inline, non-template, non-specialized, non-extern* `const` / `constexpr`
- **Functions**: `static`
- Anonymous `namespace` content, even structures/classes

External Linkage:

(accessible by other translation units, potential conflicts, same memory address)

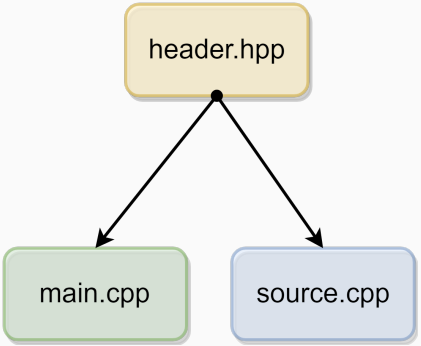
- **Global Variables**:
  - `no specifier`, or `extern`
  - `template/specialized` C++14 (no conflicts for `template`, see ODR)
  - `inline` `const` / `constexpr` C++17 (no conflicts, see ODR)
- **Functions**:
  - `no specifier` (no conflicts with `inline`, see ODR), or `extern`
  - `template/specialized` (no conflicts for `template`, see ODR)

  Note: `inline`, `constexpr` (which implies `inline` for functions) functions are not accessible by other translation units even with *external linkage*
- **Enumerators**, **Classes** and their *static, non-static* members

Dealing with Multiple Translation Units

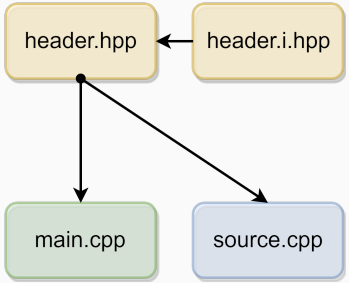
Code Structure 1

- one header, two source files → two translation units
- *the header is included in both translation units*



Code Structure 2

- two headers, two source files → two translation units
- one header for declarations (`.hpp`), and the other one for implementations (`.i.hpp`)
- *the header and the header implementation are included in both translation units*



* separate header declaration and implementation is not mandatory, but it could help to better organize the code

Class in Multiple Translation Units

header.hpp:

```
class A {
public:
    void        f();
    static void g();
private:
    int        x;
    static int y;
};
```

main.cpp:

```
#include "header.hpp"
#include <iostream>

int main() {
    A a;
    std::cout << A.x;  // print 1
    std::cout << A::y; // print 2
}
```

source.cpp:

```
#include "header.hpp"

void A::f() {}
void A::g() {}

int A::x = 1;
int A::y = 2;
```

Class in Multiple Translation Units

header.hpp:

```
struct A {
    static int y;      // zero-init
//  static int y = 3; // compile error
//            must be initialized out-of-class

    const int z = 3; // only in C++11
//  const int z;      // compile error
//                       must be initialized

    static const int w1;      // zero-init
    static const int w2 = 4; // inline-init
};
```

source.cpp:

```
#include "header.hpp"

int        A::y  = 2;
const int A::w1 = 3;
```

One Definition Rule (ODR)

One Definition Rule (ODR)

(1) In any **(single) translation unit**, a `template`, `type`, `function`, or `object`, *cannot* have more than one definition

- *Compiler error* otherwise
- Any number of declarations are allowed

(2) In the **entire program**, an `object` or `non-inline function` *cannot* have more than one definition

- *Multiple definitions linking error* otherwise
- `Entities` with *internal linkage* in different translation units are allowed, even if their names and types are the same

(3) A `template`, `type`, or `inline functions/variables`, can be defined in more than one translation unit. For a given entity, each definition must be the same

- *Undefined behavior* otherwise
- Common case: same header included in multiple translation units

ODR - Point (1), (2)

header.hpp:

```cpp
void f(); // DECLARATION
```

main.cpp:

```cpp
#include "header.hpp"
#include <iostream>
int         a = 1; // external linkage
// int      a = 7; // compiler error, Point (1)

extern int b;

static int c = 2; // internal linkage

int main() {
    std::cout << a; // print 1
    std::cout << b; // print 5
    std::cout << c; // print 2
    f();
}
```

source.cpp:

```cpp
#include "header.hpp"
#include <iostream>
// linking error, multiple definitions
// int      a = 2;   // Point (2)

int         b = 5;   // ok
// internal linkage
static int c = 4;   // ok

void f() {          // DEFINITION
//  std::cout << a; // 'a' is not visible
    std::cout << b; // print 5
    std::cout << c; // print 4
}
```

Global Variable Issues - ODR Point (2)

header.hpp:

```cpp
#include <iostream>
struct A {
    A()  { std::cout << "A()";  }
    ~A() { std::cout << "~A()"; }
};
// A            obj;         // linking error multiple definitions, Point (2)
const A         const_obj{}; // "const/constexpr" implies internal linkage
constexpr float PI = 3.14f;
```

source1.cpp:

```cpp
#include "header.hpp"

void f() { std::cout << &PI; }
// address: 0x1234ABCD

// print "A()"  the first time
// print "~A()" the first time
```

source2.cpp:

```cpp
#include "header.hpp"

void f() { std::cout << &PI; }
 // print address: 0x3820FDAC !!

// print "A()"  the second time!!
// print "~A()" the second time!!
```

Common Class Error - ODR Point (2)

header.hpp:

```
struct A {
    void f() {};  // inline DEFINITION
    void g();     // DECLARATION
    void h();     // DECLARATION
};
void A::g() {}    // DEFINITION
```

main.cpp:

```
#include "header.hpp"
// linking error
// multiple definitions of A::g()

int main() {}
```

source.cpp:

```
#include "header.hpp"
// linking error
// multiple definitions of A::g()

void A::h() {}  // DEFINITION, ok
```

ODR - Point (3)

ODR Point (3): A `template`, `type`, or `inline functions/variables`, can be defined in more than one translation unit

- The linker removes all definitions of an `inline` / `template` entity except one
- All definitions must be identical to avoid undefined behavior due to arbitrary linking order
- `inline` / `template` entities have a *unique memory address* across all translation units
- `inline` / `template` entities have the *same linkage* as the corresponding variables/functions without the specifier

`inline`

`inline` specifier allows a function or a variable (in C++17) to be identically defined (not only declared) in multiple translation units

- `inline` is one of the most misunderstood features of C++
- `inline` is a hint for the linker. Without it, the linker can emit "`multiple definitions`" error
- `inline` entities cannot be *exported*, namely, used by other translation units even if they have *external linkage* (related warning: `-Wundefined-inline`)
- `inline` doesn't mean that the compiler is forced to perform function *inlining*. It just increases the optimization heuristic threshold

```
      void f() {}
inline void g() {}
```

`f()` :

- Cannot be defined in a header included in multiple source files
- The linker issues a *"multiple definitions"* error

`g()` :

- Can be defined in a header and included in multiple source files

constexpr and inline

`constexpr` functions are implicitly `inline`

`constexpr` variables are not implicitly `inline`. C++17 added `inline` variables

```
void                 f1() {}  // external linkage
                              // potential multiple definitions error

constexpr void       f2() {}  // external linkage, implicitly inline
                              // multiple definitions allowed

constexpr int        x = 3;   // internal linkage
                              // different files allows distinct definitions
                              // -> different addresses, code bloat

inline constexpr int y = 3;   // external linkage unique memory address
                              // -> potential undefined behavior
int main() {}
```

One Definition Rule - Point (3) 1/2

header.hpp:

```
inline void f() {} // the function is marked 'inline' (no linking error)
inline int  v = 3; // the variable is marked 'inline' (no linking error) (C++17)

template<typename T>
void g(T x) {}     // the function is a template (no linking error)

using var_t = int; // types can be defined multiple times (no linking error)
```

main.cpp:

```
#include "header.hpp"

int main() {
   f();
   g(3); // g<int> generated
}
```

source.cpp:

```
#include "header.hpp"

void h() {
    f();
    g(5); // g<int> generated
}
```

Alternative organization:

header.hpp:

```
inline void f();    // DECLARATION
inline int  v;      // DECLARATION

template<typename T>
void g(T x);        // DECLARATION

using var_t = int; // type
#include "header.i.hpp"
```

header.i.hpp:

```
void f() {}     // DEFINITION
int  v = 3;     // DEFINITION

template<typename T>
void g(T x) {} // DEFINITION
```

main.cpp:

```
#include "header.hpp"

int main() {
    f();
    g(3); // g<int> generated
}
```

source.cpp:

```
#include "header.hpp"

void h() {
    f();
    g(5); // g<int> generated
}
```

ODR - Function Template

Function Template - Case 1

header.hpp:

```
template<typename T>
void f(T x) {}; // DECLARATION and DEFINITION
```

main.cpp:

```
#include "header.hpp"

int main() {
    f(3);    // call f<int>()
    f(3.3f); // call f<float>()
    f('a');  // call f<char>()
}
```

source.cpp:

```
#include "header.hpp"

void h() {
    f(3);    // call f<int>()
    f(3.3f); // call f<float>()
    f('a');  // call f<char>()
}
```

`f<int>()`, `f<float>()`, `f<char>()` are generated two times (in both translation units)

Function Template - Case 2

header.hpp:

```
template<typename T>
void f(T x); // DECLARATION
```

main.cpp:

```
#include "header.hpp"

int main() {
   f(3);    // call f<int>()
   f(3.3f); // call f<float>()
// f('a');  // linking error
} // the specialization does not exist
```

source.cpp:

```
#include "header.hpp"

template<typename T>
void f(T x) {} // DEFINITION

// template SPECIALIZATION
template void f<int>(int);
template void f<float>(float);
// any explicit instance is also
// fine, e.g. f<int>(3)
```

Function Template and Specialization

header.hpp:

```
template<typename T>
void f() {} // DECLARATION and DEFINITION
```

main.cpp:

```
#include "header.hpp"

int main() {
   f<char>(); // use the generic function
   f<int>();  // use the specialization
}
```

source.cpp:

```
#include "header.hpp"

template<>
void f<int>() {} // SPECIALIZATION
                 // DEFINITION
```

Function Template - `extern` Keyword

C++11

header.hpp:

```
template<typename T>
void f() {} // DECLARATION and DEFINITION
```

main.cpp:

```
#include "header.hpp"

extern template void f<int>();
// f<int>() is not generated by the
// compiler in this translation unit

int main() {
    f<int>();
}
```

source.cpp:

```
#include "header.hpp"

void g() {
    f<int>();
}
// or 'template void f<int>(int);'
```

ODR Function Template Common Error

header.hpp:

```
template<typename T>
void f();              // DECLARATION

// template<>          // linking error
// void f<int>() {}    // multiple definitions -> included twice
                       // full specializations are like standard functions
                       // it can be solved by adding "inline"
```

main.cpp:

```
#include "header.hpp"

int main() {}
```

source.cpp:

```
#include "header.hpp"

// some code
```

ODR - Class Template

Class Template - Case 1

header.hpp:

```cpp
template<typename T>
struct A {
    T    x = 3;  // "inline" DEFINITION
    void f() {}; // "inline" DEFINITION
};
```

main.cpp:

```cpp
#include "header.hpp"

int main() {
   A<int>   a1; // ok
   A<float> a2; // ok
   A<char>  a3; // ok
}
```

source.cpp:

```cpp
#include "header.hpp"

int g() {
   A<int>   a1; // ok
   A<float> a2; // ok
   A<char>  a3; // ok
}
```

Class Template - Case 2

header.hpp:

```
template<typename T>
struct A {
    T    x;
    void f(); // DECLARATION
};
#include "header.i.hpp"
```

header.i.hpp:

```
template<typename T>
T A<T>::x = 3;     // DEFINITION

template<typename T>
void A<T>::f() {} // DEFINITION
```

main.cpp:

```
#include "header.hpp"

int main() {
   A<int>   a1; // ok
   A<float> a2; // ok
   A<char>  a3; // ok
}
```

source.cpp:

```
#include "header.hpp"

int g() {
   A<int>   a1; // ok
   A<float> a2; // ok
   A<char>  a3; // ok
}
```

Class Template - Case 3

header.hpp:

```cpp
template<typename T>
struct A {
    T    x;
    void f(); // DECLARATION
};
```

main.cpp:

```cpp
#include "header.hpp"

int main() {
    A<int>  a1; // ok
//  A<char> a2; // linking error
}               // 'f()' is undefined
                // while 'x' has an undefined
                // value for A<char>
```

source.cpp:

```cpp
#include "header.hpp"

template<typename T>
int A<T>::x = 3;  // initialization

template<typename T>
void A<T>::f() {} // DEFINITION

// generate template specialization
template class A<int>;
```

Class Template - `extern` Keyword

C++11

header.hpp:

```
template<typename T>
struct A {
    T    x;
    void f() {}
};
```

source.cpp:

```
#include "header.hpp"

extern template class A<int>;
// A<int> is not generated by the
// compiler in this translation unit
int main() {
    A<int> a;
}
```

source.cpp:

```
#include "header.hpp"

// template specialization
template class A<int>;

// or any instantiation of A<int>
```

ODR Undefined Behavior and Summary

Undefined Behavior - `inline` Function

main.cpp:

```cpp
#include <iostream>
inline int f() { return 3; }

void g();

int main() {
    std::cout << f(); // print 3
    std::cout << g(); // print 3!!
}                     // not 5
```

source.cpp:

```cpp
// same signature and inline
inline int f() { return 5; }

int g() { return f(); }
```

The linker can *arbitrary* choose one of the two definitions of `f()`. With `-O3`, the compiler could *inline* `f()` in `g()`, so now `g()` return `5`

This issue is easy to detect in trivial examples but hard to find in large codebase

Solution: `static` or `anonymous namespace`

Undefined Behavior - Member Function

header.hpp:

```cpp
#include <iostream>

struct A {
    int f() { return 3; }
};

int g();
```

main.cpp:

```cpp
#include "header.hpp"

int main() {
    A a;
    std::cout << a.f();// print 3
    std::cout << g();  // print 3!!
}
```

source.cpp:

```cpp
struct A {
    int f() { return 5; }
};

int g() {
    A<int> a;
    return a.f();
}
```

Undefined Behavior - Function Template

header.hpp:

```cpp
template<typename T>
int f() {
    return 3;
}

int g();
```

main.cpp:

```cpp
#include "header.hpp"

int main() {
    std::cout << f<int>(); // print 3
    std::cout << g();      // print 3!!
}
```

source.cpp:

```cpp
template<typename T>
int f() {
    return 5;
}

int g() {
    return f<int>();
}
```

Undefined Behavior

Other ODR violations are even harder (if not impossible) to find, see `Diagnosing Hidden ODR Violations in Visual C++`

Some tools for partially detecting ODR violations:

- `-detect-odr-violations` flag for `gold/llvm linker`
- `-Wodr -flto` flag for GCC
- `Clang address sanitizer` + `ASAN_OPTIONS=detect_odr_violation=2` `(link)`

Another solution could be included all files in a single translation unit

ODR - Declarations and Definitions Summary

- **Header:** *declaration* of
  - functions, structures, classes, types, alias
  - `template` functions, structs, classes
  - `extern` variables, functions
- **Header (implementation):** *definition* of
  - `inline` variables/functions
  - `template` variables/functions/classes
  - global *static, non-static* `const/constexpr` variables and `constexpr` functions
- **Source file:** *definition* of
  - functions, including `template` full specializations
  - classes
  - `extern` and `static` global variables/functions

Modern C++ Programming

12. Translation Units II

Include, Module, and Namespace

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

`#include` Issues

The **include guard** avoids the problem of multiple inclusions of a header file in a translation unit

header.hpp:

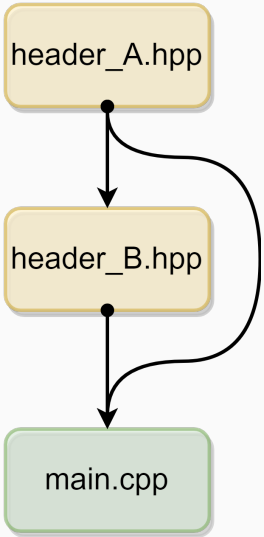
```
#ifndef HEADER_HPP  // include guard
#define HEADER_HPP

... many lines of code ...

#endif // HEADER_HPP
```

`#pragma once` preprocessor directive is an alternative to the **include guard** to force current file to be included only once in a translation unit

- **#pragma once** is less portable but less verbose and compile faster than the **include guard**

The **include guard**/**#pragma once** should be used in every header file

Common case:



Include Guard

header_A.hpp:

```
#pragma once   // prevent "multiple definitions" linking error

struct A {
};
```

header_B.hpp:

```
#include "header_A.hpp"  // included here

struct B {
    A a;
};
```

main.cpp:

```
#include "header_A.hpp"  // .. and included here
#include "header_B.hpp"
int main() {
    A a;  // ok, here we need "header_A.hpp"
    B b;  // ok, here we need "header_B.hpp"
}
```

Forward Declaration

Forward declaration is a declaration of an identifier for which a complete definition has not yet given. "*forward*" means that an entity is declared before it is defined

```
void f();  // function forward declaration

class A;   // class    forward declaration

int main() {
    f();   // ok, f() is defined in the translation unit
//  A a;   // compiler error no definition (incomplete type)
           // e.g. the compiler is not able to deduce the size of A
    A* a;  // ok
}

void f() {}  // definition of f()
class A  {}; // definition of A()
```

Forward Declaration vs. `#include`

Advantages:

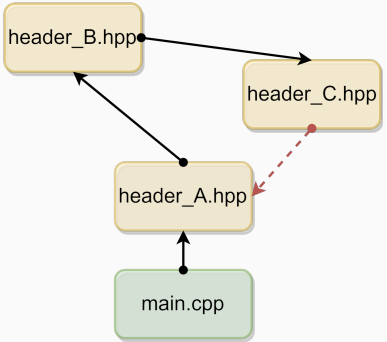
- Forward declarations can save compile time as `#include` forces the compiler to open more files and process more input
- Forward declarations can save on unnecessary recompilation. `#include` can force your code to be recompiled more often, due to unrelated changes in the header

Disadvantages:

- Forward declarations can hide a dependency, allowing user code to skip necessary recompilation when headers change
- A forward declaration may be broken by subsequent changes to the library
- Forward declaring multiple symbols from a header can be more verbose than simply `#including` the header

google.github.io/styleguide/cppguide.html#Forward_Declarations

A **circular dependency** is a relation between two or more modules which either directly or indirectly depend on each other to function properly



Circular dependencies can be solved by using forward declaration, or better, by rethinking the project organization

header_A.hpp:

```
#pragma once  // first include
#include "header_B.hpp"
class A {
    B* b;
};
```

header_B.hpp:

```
#pragma once  // second include
#include "header_C.hpp"
class B {
    C* c;
};
```

header_C.hpp:

```
#pragma once  // third include
#include "header_A.hpp"
class C {  // compile error "header_A.hpp": already included by "main.cpp"
    A* a;  // the compiler does not know the meaning of "A"
};
```

Circular Dependencies (fix) 3/3

header_A.hpp:

```
#pragma once
class B;   // forward declaration
           // note: does not include "header_B.hpp"
class A {
    B* b;
};
```

header_B.hpp:

```
#pragma once
class C;   // forward declaration
class B {
   C* c;
};
```

header_C.hpp:

```
#pragma once
class A;   // forward declaration
class C {
    A* a;
};
```

Common Linking Errors

Very common *linking* errors:

- `undefined reference`

  Solutions:

  - Check if the right headers and sources are included
  - Break circular dependencies (could be hard to find)

- `multiple definitions`

  Solutions:

  - `inline` function, variable definition or `extern` declaration
  - Add `include guard`/ `#pragma once` to header files
  - Place template definition in header file and full specialization in source files

C++20 Modules

The `#include` problem: *The duplication of work* - the same header files are possibly parsed/compiled multiple times and most of the compiled output is later-on thrown away again by the linker

C++20 introduces **modules** as a robust replacement for plain `#include`

Module (C++20)

A **module** is a set of source code files that are compiled independently of the translation units that import them

Modules allow defining clearer interfaces with a fine-grained control on what to *import* and *export* (similar to Java, Python, Rust, etc.)

- A Practical Introduction to C++20's Modules
- Modules the beginner's guide
- Understanding C++ Modules
- Overview of modules in C++

Less error-prone than `#include`:

- No effect on the compilation of the translation unit that *imports* the module
- Macros, preprocessor directives, and *non-exported* names declared in a module are not visible outside the module
- Declarations in the *importing* translation unit do not participate in overload resolution or name lookup in the *imported* module

Other benefits:

- **(Much) Faster compile time**. After a module is compiled once, the results are stored in a binary file that describes all the exported types, functions, and templates
- **Smaller binary size**. Allow to incorporate only the imported code and not the whole `#include`

Terminology

A **module** consists of one or more **module units**

A **module unit** is a *translation unit* that contains a `module` declaration

```
module my.module.example;
```

A **module name** is a concatenation of *identifiers* joined by dots (the dot carries no meaning) `my.module.example`

A **module unit purview** is the content of the translation unit

A **module purview** is the set of **purviews** of a given *module name*

Visibility and Reachability

Visibility of **names** instructs the linker if a symbol can be used by another translation unit. *Visible* also means *a candidate for name lookup*

Reachable of **declarations** means that the semantic properties of an entity are available

- Each *visible* declaration is also *reachable*
- Not all *reachable* declarations are also *visible*

Reachability Example

Common example: the members of a class are reachable (i.e. can be used) or the class size is known, but not the class type itself

```
auto g() {
    struct A {
        void f() {}
    };
    return A{};
}
//-------------------------------------------------------------------------------

auto x = g();           // ok
// A y = g();           // compile error, "A" is unknown at this point
x.f();                  // ok
sizeof(x);              // ok
using T = decltype(x); // ok
```

Module Unit Types

- A **module interface unit** is a *module unit* that exports a symbol and/or *module name* or *module partition name*
- A **primary module interface unit** is a *module interface unit* that exports the *module name*. There must be one and only one *primary module interface unit* in a module
- A **module implementation unit** is a *module unit* that does not export a *module name* or *module partition name*

A **module interface unit** should contain only declarations if one or more *module implementation units* are present. A **module implementation unit** implements/defines the declarations of *module interface units*

Keywords

`module` specifies that the file is a *named module*

```
module my.module; // first code line
```

`import` makes a module and its symbols visible in the current file

```
import my.module; // after module declaration and #include
```

`export` makes symbols visible to the files that **`import`** the current module

- **`export module`** `<`**`module_name`**`>` makes visible all the exported symbols of a module. It must appear once per module in the *primary module interface unit*
- **`export namespace`** `<`**`namespace`**`>` makes visible all symbols in a namespace
- **`export`** `<`**`entity`**`>` makes visible a specific function, class, or variable
- **`export`** `{<`**`code`**`>}` makes visible all symbols in a block

`import` Example

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World";
}
```

Preprocessing size `-E`: ∼1MB

```cpp
import <iostream>

int main() {
    std::cout << "Hello World";
}
```

Preprocessing size: 236B (x500)

Compile time: 2x (up to 10x) less

```
g++-12 -std=c++20 -fmodules-ts main.cpp -x c++-system-header iostream
```

`export` Example - Single Primary Module Interface Unit

my_module.cpp

```
export module my.example;       // make visible all module symbols

export int f1() { return 3; } // export function

export namespace my_ns {       // export namespace and its content
int f2() { return 5; }
}

export {                       // export code block
int f3() { return 2; }
int f4() { return 8; }
}

void internal() {}             // NOT exported. It can be used only internally
```

`export` Example - Two Module Interface Units

`my_module1.cpp` *Primary Module Interface Unit*

```
export module my.example; // This is the only file that exports all module symbols

export int f1() { return 3; } // export function
```

`my_module2.cpp` *Module Interface Unit*

```
module my.example; // Module declaration but symbols are not exported

export namespace my_ns {      // export namespace
int f2() { return 5; }
}

export {                      // export code block7
int f3() { return 2; }
int f4() { return 8; }
}
```

`export` Example - Module Interface and Implementation Units

`my_module1.cpp` *Primary Module Interface Unit*

```
export module my.example; // This is the only file that exports all module symbols

export int f1();          // export function

export {                  // export code block
int f3();
int f4();
}
```

`my_module2.cpp` *Module Implementation Unit*

```
module my.example; // Module declaration but symbols are not exported

int f1() { return 3; }
int f3() { return 2; }
int f4() { return 8; }
```

Keyword Notes

`import`

- A **module implementation unit** can `import` another module, but cannot `export` any names. Symbols of the *module interface unit* are imported implicitly
- All `import` must appear before any declarations in that module unit and after `module;` a `export module` (if present)

`export`

- Symbols with *internal linkage* or *no linkage* cannot be exported, i.e. anonymous namespaces and `static` entities
- The `export` keyword is used in **module interface units** only
- The semantic properties associated to **exported** symbols become *reachable*

`export import` Declaration

Imported modules can be directly **re-exported**

```
export module main_module; // Top-level primary module interface unit

export import sub_module;  // import and re-export "sub_module"
```

```
export module sub_module; // Primary module interface unit

export void f() {}
```

```
import main_module;

int main() {
    f(); // ok, f() is visible
}
```

Global Module Fragment

A **global module fragment** (*unnamed module*) can be used to *include header files* in a *module interface* when importing them is not possible or preprocessing directives are needed

```
module;                 // start Global Module Fragment

#define ENABLE_FAST_MATH
#include "my_math.h"

export modul my.module; // end Global Module Fragment
```

Macro definitions or other preprocessing directives are not visible outside the file itself

Private Module Fragment

A **private module fragment** allows a module to be represented as a single translation unit without making all the contents of the module reachable to importers

→ A modification of the *private module fragment* does not cause recompilation

If a module unit contains a *private module fragment*, it will be the only module unit of its module

```
export module my.example;
export int f();

module :private; // start private module fragment

int f() {        // definition not reachable from importers of f()
    return 42;
}
```

Header Module Unit

Legacy headers can be directly imported with `import` instead of `#include`

All declarations are implicitly *exported* and attached to the **global module (fragment)**

- Macros from the header are available for the *importer*, but macros defined in the *importer* have no effect on the *imported header*
- Importing compiled declarations is faster than `#include`

C++23 will introduce modules for the standard library

A *module* can be organized in *isolated* **module partitions**

Syntax:

```
export module module_name : partition_name;
```

- *Declarations* in any of the **partitions** are visible within the entire module
- Like common modules, a *module partition* consists in one **module partition interface unit** and zero or more **module partition implementation units**
- *Module partitions* are not *visible* outside the module
- *Module partitions* do not *implicitly import* the module interface
- All names `exported` by *partition interface* files must be `imported` and `re-exported` by the *primary module interface file*

main_module.ixx

```
export module main_module;

export import :partition1; // re-export f() to importers of "main_module"
export import :partition2; // re-export g() to importers of "main_module"

export void h() { internal(); } // internal() can be directly used
```

partition1.ixx

```
export module module_name:partition1;

export void f() {}
```

partition2.ixx

```
export module module_name:partition2;

export void g() {}
void internal() {} // not exported
```

Namespace

Overview

The problem: Named entities, such as variables, functions, and compound types declared outside any block has *global scope*, meaning that its name is valid anywhere in the code

Namespaces *allow grouping named entities that otherwise would have global scope into narrower scopes, giving them* ***namespace scope*** *(where* `std` *stands for "standard")*

Namespaces provide a method for preventing name conflicts in large projects. Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes

Namespace Functions vs. Class + static Methods

Namespace functions:

- Namespace can be extended anywhere (without control)
- Namespace specifier can be avoided with the keyword `using`

Class + `static` methods:

- Can interact only with static data members
- `struct/class` cannot be extended outside their declarations

→ `static` methods should define operations strictly related to an object state (*statefull*)

→ otherwise `namespace` should be preferred (*stateless*)

Namespace Example 1

```cpp
#include <iostream>
namespace ns1 {
void f() {
    std::cout << "ns1" << std::endl;
}
} // namespace ns1

namespace ns2 {
void f() {
    std::cout << "ns2" << std::endl;
}
} // namespace ns2

int main () {
    ns1::f(); // print "ns1"
    ns2::f(); // print "ns2"
//  f();      // compile error f() is not visible
}
```

Namespace Example 2

```cpp
#include <iostream>

namespace ns1 {
void f() { std::cout << "ns1::f()" << endl; }
} // namespace ns1

namespace ns1 { // the same namespace can be declared multiple times
void g() { std::cout << "ns1::g()" << endl; }
} // namespace ns1

int main () {
    ns1::f(); // print "ns1::f()"
    ns1::g(); // print "ns1::g()"
}
```

‘using namespace’ Declaration

```
#include <iostream>
void f() { std::cout << "global" << endl; }

namespace ns1 {
void f() { std::cout << "ns1::f()" << endl; }
void g() { std::cout << "ns1::g()" << endl; }
} // namespace ns1

int main () {
    f();       // ok, print "global"
    using namespace ns1; // expand "ns1" in this scope (from this line)
    g();       // ok, print "ns1::g()", only one choice
//  f();       // compile error ambiguous function name
    ::f();     // ok, print "global"
    ns1::f(); // ok, print "ns1::f()"
}
```

Nested Namespaces

```
#include <iostream>
namespace ns1 {
void f() { std::cout << "ns1::f()" << endl; }

namespace ns2 {
void f() { std::cout << "ns1::ns2::f()" << endl; }
} // namespace ns2

} // namespace ns1
```

C++17 allows *nested namespace* definitions with less verbose syntax:

```
namespace ns1::ns2 {
    void h()
}
```

Namespace Alias

Namespace alias allows declaring an alternate name for an existing namespace

```
namespace very_very_long_namespace {
    void g() {}
}

int main() {
    namespace ns = very_very_long_namespace; // namespace alias
    ns::g();                                 // available only in this scope
}
```

Anonymous Namespace

A namespace with no identifier is called **unnamed/anonymous namespace**

Entities within an anonymous namespace have *internal linkage* and, therefore, are used for declaring unique identifiers, visible only in the same source file

Anonymous namespaces vs. static: Anonymous namespaces allow *type declarations* and *class definition*, and they are *less verbose*

main.cpp

```cpp
#include <iostream>
namespace { // anonymous
void f() { std::cout << "main"; }
} // namespace      internal linkage

int main() {
    f();   // print "main"
}
```

source.cpp

```cpp
#include <iostream>
namespace { // anonymous
void f() { std::cout << "source"; }
} // namespace      internal linkage

int g() {
    f();   // print "source"
}
```

`inline` Namespace

`inline` namespace is a concept similar to library versioning. It is a mechanism that makes a nested namespace look and act as if all its declarations were in the surrounding namespace

```
namespace ns1 {

inline namespace V99 { void f(int) {} }  // most recent version

namespace V98 { void f(int) {} }

} // namespace ns1
using namespace ns1;

V98::f(1);  // call V98
V99::f(1);  // call V99
f(1);       // call default version (V99)
```

Attributes for Namespace

C++17 allows defining attribute on namespaces

```
namespace [[deprecated]] ns1 {

void f() {}

} // namespace ns1

ns1::f(); // compiler warning
```

Compiling Multiple Translation Units

Fundamental Compiler Flags

Include flag: `g++ -I include/ main.cpp -o main.x`

- `-I` : Specify the **`include` path** for the project headers
- `-isystem` : Specify the **`include` path** for system (external) headers (warnings are not emitted)

They can be used multiple times

Important: *include* and *library* compiler flags, as well as multiple values in an environment variable, are evaluated in order from left to right. The first match suppress the other ones

Compile to a file object: `g++ -c source.cpp -o source.o`

Compile Methods

Method 1

Compile all files together (naive):

```
g++ main.cpp source.cpp -o main.out
```

Method 2

Compile each *translation unit* in a file object:

```
g++ -c source.cpp -o source.o
g++ -c main.cpp -o main.o
```

Multiple objects can be compiled in parallel

Link all file objects:

```
g++ main.o source.o -o main.out
```

A **library** is a package of code that is meant to be reused by many programs

A **static library** is a set of object files (just the concatenation) that are directly linked into the final executable. If a program is compiled with a static library, all the functionality of the static library becomes part of final executable

- A static library cannot be modified without re-link the final executable
- Increase the size of the final executable
+ The linker can optimize the final executable (*link time optimization*)

Given the static library `my_lib`, the corresponding file is:

- Linux: `libmy_lib.a`
- Windows: `my_lib.lib`

A **dynamic library**, also called a **shared library**, consists of routines that are loaded into the application at run-time. If a program is compiled with a dynamic library, the library does not become part of final executable. It remains as a separate unit

+ A dynamic library can be modified without re-link
− Dynamic library functions are called outside the executable
− Neither the linker nor the compiler can optimize the code between shared libraries and the final executable
- The environment variables must be set to the right shared library path, otherwise the application crashes at the beginning

Given the shared library `my_lib`, the corresponding file is:

- Linux: `libmy_lib.so`
- Windows: `my_lib.dll` + `my_lib.lib`

Deal with Libraries

Specify the **library path** (path where search for static/dynamic libraries) to the compiler: `g++ -L<library_path> main.cpp -o main`

`-L` can be used multiple times (`/LIBPATH` on Windows)

Specify the **library name** (e.g. `liblibrary.a`) to the compiler:
`g++ -llibrary main.cpp -o main`

The full path on Windows instead

Deal with Libraries

Linux/Unix environmental variables:

- `LIBRARY_PATH` Specify the directories where search for *static* libraries `.a` at *compile-time*
- `LD_LIBRARY_PATH` Specify the directories where search for *dynamic/shared* libraries `.so` at *run-time*

Windows environmental variables:

- `LIBPATH` Specify the directories where search for *static* libraries `.lib` at *compile-time*
- `PATH` Specify the directories where search for *dynamic/shared* libraries `.dll` at *run-time*

Build Static/Dynamic Libraries

Static Library Creation

- Create object files for each translation unit (`.cpp`)
- Create the static library by using the **`archiver`** (**`ar`**) Linux utility

```
g++ source1.c -c source1.o
g++ source2.c -c source2.o
ar rvs libmystaticlib.a source1.o source2.o
```

Dynamic Library Creation

- Create object files for each translation unit (`.cpp`). Since library cannot store code at fixed addresses, the compiler must generate *position independent code*
- Create the dynamic library

```
g++ source1.c -c source1.o -fPIC
g++ source2.c -c source2.o -fPIC
g++ source1.o source2.o -shared -o libmydynamiclib.so
```

Demangling

Name mangling is a technique used to solve various problems caused by the need to resolve unique names

Transforming C++ ABI (Application binary interface) identifiers into the original source identifiers is called **demangling**

Example (linking error):

```
_ZNSt13basic_filebufIcSt11char_traitsIcEED1Ev
```

After demangling:

```
std::basic_filebuf<char, std::char_traits<char> >::~basic_filebuf()
```

How to demangle: `c++filt`

Online Demangler: `https://demangler.com`

Find Dynamic Library Dependencies

The `ldd` utility shows the shared objects (shared libraries) required by a program or other shared objects

```
$ ldd /bin/ls
      linux-vdso.so.1 (0x00007ffcc3563000)
      libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f87e5459000)
      libcap.so.2 => /lib64/libcap.so.2 (0x00007f87e5254000)
      libc.so.6 => /lib64/libc.so.6 (0x00007f87e4e92000)
      libpcre.so.1 => /lib64/libpcre.so.1 (0x00007f87e4c22000)
      libdl.so.2 => /lib64/libdl.so.2 (0x00007f87e4a1e000)
      /lib64/ld-linux-x86-64.so.2 (0x00005574bf12e000)
      libattr.so.1 => /lib64/libattr.so.1 (0x00007f87e4817000)
      libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f87e45fa000)
```

The nm utility provides information on the symbols being used in an object file or executable file

```
$ nm -D -C something.so
    w __gmon_start__
    D __libc_start_main
    D free
    D malloc
    D printf

# -C:  Decode low-level symbol names
# -D:  accepts a dynamic library
```

readelf displays information about ELF format object files

```
$ readelf --symbols something.so | c++filt
    ... OBJECT  LOCAL  DEFAULT   17 __frame_dummy_init_array_
    ... FILE    LOCAL  DEFAULT  ABS prog.cpp
    ... OBJECT  LOCAL  DEFAULT   14 CC1
    ... OBJECT  LOCAL  DEFAULT   14 CC2
    ... FUNC    LOCAL  DEFAULT   12 g()

# --symbols: display symbol table
```

objdump displays information about object files

```
$ objdump -t -C something.so | c++filt
... df *ABS*   ...  prog.cpp
...  O .rodata ...  CC1
...  O .rodata ...  CC2
...  F .text   ...  g()
...  O .rodata ...  (anonymous namespace)::CC3
...  O .rodata ...  (anonymous namespace)::CC4
...  F .text   ...  (anonymous namespace)::h()
...  F .text   ...  (anonymous namespace)::B::j1()
...  F .text   ...  (anonymous namespace)::B::j2()

# --t: display symbols
# -C:  Decode low-level symbol names
```

References and Additional Material

- 20 ABI (Application Binary Interface) breaking changes every C++ developer should know
- Policies/Binary Compatibility Issues With C++
- 10 differences between static and dynamic libraries every C++ developer should know

Modern C++ Programming

13. Code Conventions

Federico Busato

2024-03-29

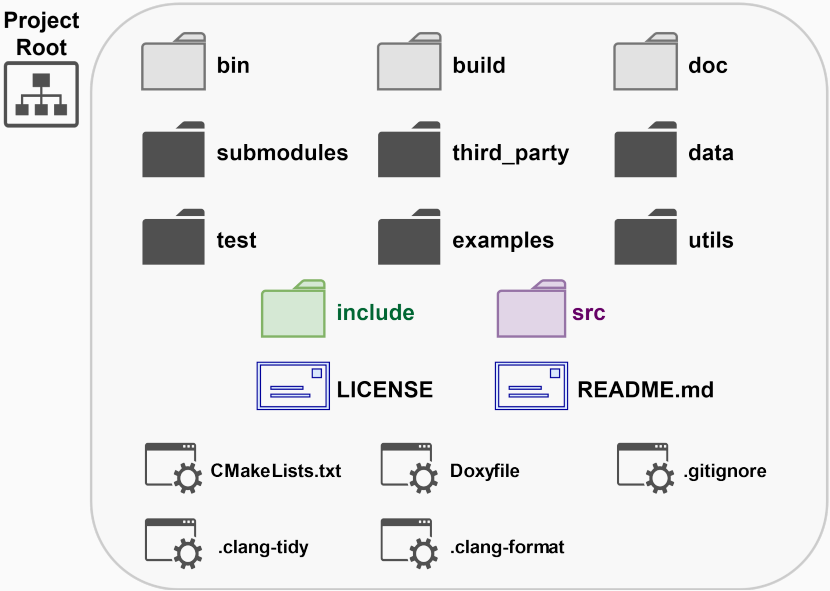
Table of Contents

Table of Contents

Table of Contents

C++ Project Organization

“Common” Project Organization



Fundamental directories

`include` Project *public* header files

`src` Project source files and *private* headers

`test` (or `tests`) Source files for testing the project

Empty directories

`bin` Output executables

`build` All intermediate files

`doc` (or `docs`) Project documentation

Optional directories

`submodules`	Project submodules
`third_party`	(less often `deps`/`external`/`extern`) dependencies or external libraries
`data`	(or `extras`) Files used by the executables or for testing
`examples`	Source files for showing project features
`utils`	(or `tools`, or `script`) Scripts and utilities related to the project
`cmake`	CMake submodules (.cmake)

Project Files

`LICENSE`	Describes how this project can be used and distributed
`README.md`	General information about the project in `Markdown` format *
`CMakeLists.txt`	Describes how to compile the project
`Doxyfile`	Configuration file used by `doxygen` to generate the documentation (see next lecture)
others	`.gitignore`, `.clang-format`, `.clang-tidy`, etc.

* `Markdown` is a language with a syntax corresponding to a subset of HTML tags
`github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet`

Readme and License

README.md

- README template:
  - `Embedded Artistry README Template`
  - `Your Project is Great, So Let's Make Your README Great Too`

LICENSE

- Choose an open source license:
  `choosealicense.com`
- License guidelines:
  `Why your academic code needs a software license`

File extensions

Common C++ file extensions:

- **header** `.h` `.hh` `.hpp` `.hxx`
- **header implementation** `.i.h` `.i.hpp` `-inl.h` `.inl.hpp`
  1. separate implementation from interface for inline functions and templates
  2. keep implementation "inline" in the header file
- **source/implementation** `.c` `.cc` `.cpp` `.cxx`

Common conventions:

- `.h` `.c` `.cc` GOOGLE
- `.hh` `.cc`
- `.hpp` `.cpp`
- `.hxx` `.cxx`

Common Rules

The file should have the same name of the class/namespace that they implement

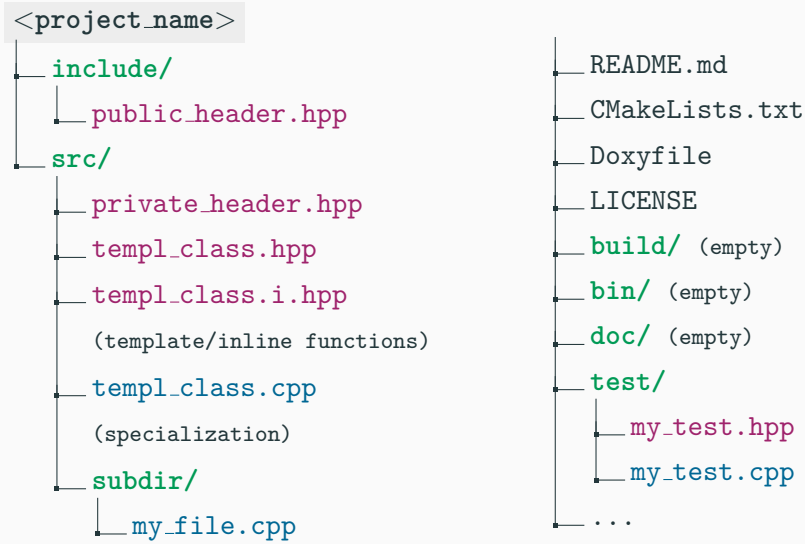
- `class MyClass`
  `my_class.hpp` (`MyClass.hpp`)
  `my_class.i.hpp` (`MyClass.i.hpp`)
  `my_class.cpp` (`MyClass.cpp`)

- `namespace my_np`
  `my_np.hpp` (`MyNP.hpp`)
  `my_np.i.hpp` (`MyNP.i.hpp`)
  `my_np.cpp` (`MyNP.cpp`)

“Common” Project Organization Notes

- Public **header(s)** in `include/`
- **source files**, private **headers**, **header implementations** in `src/` directory
- The **main** file (if present) can be placed in `src/` and called `main.cpp`
- **Code tests**, *unit* and *functional* (see `C++ Ecosystem I` slides), can be placed in `test/`, or **unit tests** can appear in the same directory of the component under test with the same filename and include `.test` suffix, e.g. `my_file.test.cpp`

"Common" Project Organization Example



"Common" Project Organization - Improvements

The "common" project organization can be improved by adding the *name of the project* as subdirectory of `include/` and `src/`

This is particularly useful when the project is used as *submodule* (part of a larger project) or imported as an *external library*

The includes now look like:

```
#include <my_project/public_header.hpp>
```

```
<project_name>
├── include/
│   └── <project_name>/
│       └── public_header.hpp
└── src/
    └── <project_name>/
        └── private_file.hpp
```

- *Header* and *source files* (or *module interface* and *implementation files*) are next to each other (no `include/` and `src/` split)
- *Headers* are included with `<>` and contain the project directory prefix, for example, `<hello/hello.hpp>` (no need of `""` syntax)
- *Header* and *source file* extensions are `.hpp` / `.cpp` (`.mpp` for module interfaces). No special characters other than `_` and `-` in file names with `.` only used for extensions
- A source file that implements a *module's unit tests* should be placed next to that *module's files* and be called with the module's name plus the `.test` second-level extension
- A project's functional/integration tests should go into the **`tests/`** subdirectory

<project_name> (v1)

- <project_name>/
  - public_header.hpp
  - private_header.hpp
  - my_file.cpp
  - my_file.mpp
  - my_file.test.cpp
- tests/
  - my_functional_test.cpp
- build/
- doc/
- ...

<project_name> (v2)

- <project_name>/
  - public_header.hpp
  - private/
    - private_header.hpp
    - my_internal_file.cpp
    - my_internal_file.test.cpp
- tests/
  - my_functional_test.cpp
- build/
- doc/
- ...

References

- Kick-start your C++! A template for modern C++ projects
- The Pitchfork Layout
- Canonical Project Structure

Coding Styles and Conventions

"One thing people should remember is there is what you can do in a language and what you should do"

Bjarne Stroustrup

Most important rule:

BE CONSISTENT!!

“The best code explains itself”

GOOGLE

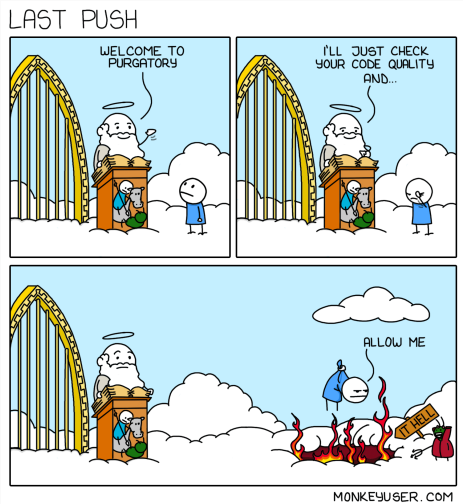
"80% of the lifetime cost of a piece of software goes to maintenance"

Unreal Engine

Code Quality

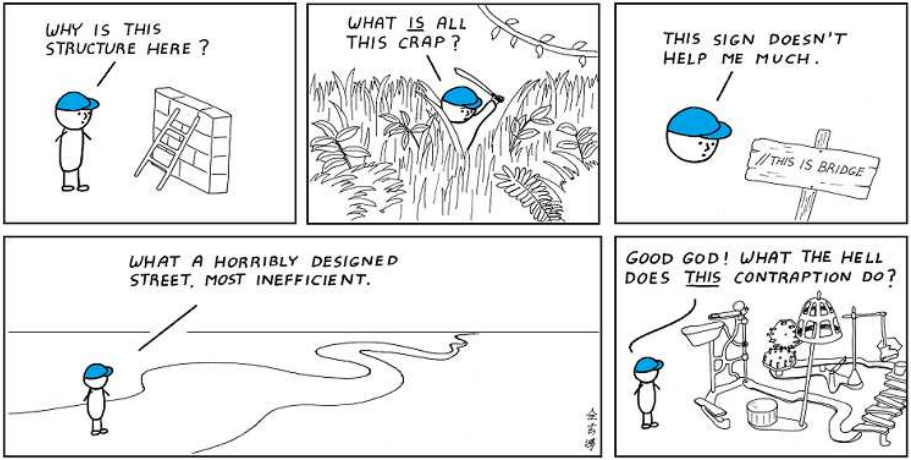
“The worst thing that can happen to a code base is size”

— Steve Yegge



Bad Code

How *my* code looks like for other people?



abstrusegoose.com/432

Coding styles are common guidelines to improve the *readability*, *maintainability*, prevent *common errors*, and make the code more *uniform*

- ***LLVM Coding Standards***. `llvm.org/docs/CodingStandards.html`
- ***Google C++ Style Guide***. `google.github.io/styleguide/cppguide.html`
- ***Webkit Coding Style***. `webkit.org/code-style-guidelines`
- ***Mozilla Coding Style***. `firefox-source-docs.mozilla.org`
- ***Chromium Coding Style***. `chromium.googlesource.com`, `c++-dos-and-donts.md`

- ***Unreal Engine - Coding Standard***
  `docs.unrealengine.com/en-us/Programming`
- ***μOS++***
  `micro-os-plus.github.io/develop/coding-style`
  `micro-os-plus.github.io/develop/naming-conventions`

More educational-oriented guidelines

- ***C++ Guidelines***
  `isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`

Secure Coding

- ***High Integrity C++ Coding Standard***. `www.perforce.com/resources`
- ***CERT C++ Secure Coding***. `wiki.sei.cmu.edu`

Critical system coding standards

- ***Misra - Coding Standard***. `www.misra.org.uk`
- ***Autosar - Coding Standard***. `www.misra.org.uk`
- ***Joint Strike Fighter Air Vehicle***.
  `www.perforce.com/blog/qac/jsf-coding-standard-cpp`

Legend

※ → **Important!**
Highlight potential code issues such as bugs, inefficiency, and can compromise readability. Should not be ignored

∗ → **Useful**
It is not fundamental, but it emphasizes good practices and can help to prevent bugs. Should be followed if possible

▪ → **Minor / Obvious**
Style choice or not very common issue

#include

※ **Every include must be self-contained**

- include every header you need directly
- do not rely on recursive `#include`
- the project must compile with any include order

LLVM, Google, Unreal, μOS++, Core

∗ **Include as less as possible, especially in header files**

- do not include unneeded headers
- minimize dependencies
- minimize code in headers (e.g. use forward declarations)

LLVM, Google, Chromium, Unreal, Hic, μOS++

Order of #include LLVM, WebKit, Core

(1) Main module/interface header, if exists (it is only one)

- space

(2) Local project includes (in lexicographic order)

- space

(3) System includes (in lexicographic order)

Note: **(2)** and **(3)** can be swapped Google

System includes are self-contained, local includes might not

Project includes LLVM, Google, WebKit, Hic, Core

* Use `""` syntax
* Should be absolute paths from the project include root
  e.g. `#include "directory1/header.hpp"`

System includes LLVM, Google, WebKit, Hic

* Use `<>` syntax
  e.g. `#include <iostream>`

※ **Always use an include guard**

- `macro include guard` vs. `#pragma once`
  - Use `macro include guard` if portability is a very strong requirement — LLVM, Google, Chromium, Core
  - `#pragma once` otherwise — WebKit, Unreal

- `#include` preprocessor should be placed immediately after the header comment and include guard — LLVM

Forward declarations vs. #includes

- *Prefer forward declaration*: reduce compile time, less dependency — Chromium
- *Prefer* `#include` : *safer* — Google

* **Use C++ headers instead of C headers:**
  `<cassert>` instead of `<assert.h>`
  `<cmath>` instead of `<math.h>`, etc.

- **Report at least one function used for each include**
  `<iostream>  // std::cout, std::cin`

```
#include "my_class.hpp"            // MyClass
                                   [ blank line ]
#include "my_dir/my_headerA.hpp"   // npA::ClassA, npB::f2()
#include "my_dir/my_headerB.hpp"   // np::g()
                                   [ blank line ]
#include <cmath>                   // std::fabs()
#include <iostream>                // std::cout
#include <vector>                  // std::vector
```

Macro and Preprocessing

※ **Avoid defining macros**, especially in headers — Google

- Do not use macro for enumerators, constants, and functions

— WebKit, Google

※ **Use a prefix for all macros** related to the project `MYPROJECT_MACRO`

— Google, Unreal

※ `#undef` **macros wherever possible** — Google

- Even in the source files if *unity build* is used (merging multiple source files to improve compile time)

※ **Always use curly brackets for multi-line macro**

```
#define MACRO       \
{                   \
    line1;          \
    line2;          \
}
```

※ **Always put macros after** `#include` statements HIC

- Put macros outside namespaces as they don't have a scope

- Close `#endif` with the respective condition of the first `#if`

```
#if defined(MACRO)
    ...
#endif // defined(MACRO)
```

- The hash mark that starts a preprocessor directive should always be at the beginning of the line GOOGLE

```
#if defined(MACRO)
#   define MACRO2
#endif
```

- Place the `\` rightmost for multi-line macro

```
#define MACRO2                  \
    macro_def...
```

- Prefer `#if defined(MACRO)` instead of `#ifdef MACRO`

  Improve readability, help grep-like utils, and it is uniform with multiple conditions

```
#if defined(MACRO1) && defined(MACRO2)
```

namespace

※ **Avoid `using namespace`-directives at global scope**
LLVM, Google, WebKit, Unreal, Hic, μOS++

∗ **Limit `using namespace`-directives at local scope** and prefer explicit namespace specification
Google, WebKit, Unreal

※ **Always place code in a namespace** to avoid *global namespace pollution*
Google, WebKit

* **Avoid *anonymous* namespaces in headers** GOOGLE, CERT

- `anonymous namespace` vs. `static`
  - Prefer `anonymous namespaces` instead of `static` variables/functions
    GOOGLE, CORE
  - Use `anonymous namespaces` only for inline class declaration, `static` otherwise
    LLVM, STATIC

* **Anonymous namespaces and source files:**
  Items local to a source file (e.g. `.cpp`) file should be wrapped in an anonymous namespace. While some such items are already file-scope by default in C++, not all are; also, shared objects on Linux builds export all symbols, so anonymous namespaces (which restrict these symbols to the compilation unit) improve function call cost and reduce the size of entry point tables CHROMIUM, CORE, HIC

- The content of namespaces is not indented — LLVM, Google, WebKit

```
namespace ns {

void f() {}

}
```

- Close namespace declarations — LLVM, Google

```
} // namespace <namespace_identifier>
} // namespace (for anonymous namespaces)
```

Variables and Arithmetic Types

※ **Place a variable in the *narrowest scope* possible, and *always initialize* variables in the declaration**

GOOGLE, ISOCPP, MOZILLA, HIC, *mu*OS, CERT

∗ **Avoid static (non-const) global variables** LLVM, GOOGLE, CORE, HIC

- Use assignment syntax `=` when performing "simple" initialization CHROMIUM

※ **Use fixed-width integer type** (e.g. `int64_t`, `int8_t`, etc.)
Exception: `int` Google, `int/unsigned` Unreal

* `size_t` vs. `int64_t`
- Use `size_t` for object and allocation sizes, object counts, array and pointer offsets, vector indices, and so on. (integer overflow behavior for signed types is undefined) Chromium
- Use `int64_t` instead of `size_t` for object counts and loop indices Google

▪ Use brace initialization to convert *constant* arithmetic types (narrowing) e.g. `int64_t{MyConstant}` Google

* Use `true`, `false` for boolean variables instead numeric values `0, 1` WebKit

- ※ **Do not shift $\ll$ signed operands** — Hic, Core, μOS
- ※ **Do not directly compare floating point** `==`, $<$, etc. — Hic
- ※ **Use signed types for arithmetic** — Core

Style:

- Use floating-point literals to highlight floating-point data types, e.g. `30.0f`
  WebKit (opposite)
- Avoid redundant type, e.g. `unsigned int`, `signed int` — WebKit

Functions

* **Limit overloaded functions**. Prefer default arguments GOOGLE, CORE

* **Split up large functions** into logical sub-functions for improving readability and compile time UNREAL, GOOGLE, CORE

- Use `inline` only for small functions (e.g. < 10 lines) GOOGLE, HIC

※ **Never return pointers for new objects**. Use `std::unique_ptr` instead CHROMIUM, CORE

```
int*                 f() { return new int[10]; } // wrong!!
std::unique_ptr<int> f() { return new int[10]; } // correct
```

※ **Prefer pass `by-reference` instead `by-value`** except for raw arrays and built-in types — WebKit

∗ **Pass function arguments by `const` *pointer* or *reference*** if those arguments are not intended to be modified by the function — Unreal

∗ **Do not pass `by-const-value` for built-in types**, especially in the declaration (same signature of `by-value`)

∗ **Prefer returning values** rather than output parameters — Google

∗ **Do not declare functions with an excessive number of parameters**. Use a wrapper structure instead — Hic, Core

- Prefer `enum` to `bool` on function parameters
- All parameters should be aligned if they do not fit in a single line (especially in the declaration) GOOGLE

```
void f(int        a,
       const int* b);
```

- Parameter names should be the same for declaration and definition CLANG-TIDY
- Do not use `inline` when declaring a function (only in the definition) LLVM
- Do not separate declaration and definition for template and inline functions GOOGLE

Structs and Classes

* **Use a `struct` only for passive objects that carry data; everything else is a `class`** Google

※ **Objects are fully initialized by constructor call** Google, WebKit, Core

* **Prefer in-class initializers to member initializers** Core

* **Initialize member variables in the order of member declaration** Core, Hic

- Use delegating constructors to represent common actions for all constructors of a class Core

* **Do not define implicit conversions**. Use the `explicit` keyword for conversion operators and constructors GOOGLE, CORE

* **Prefer `= default` constructors** over user-defined / implicit default constructors MOZILLA, CHROMIUM, CORE, HIC

* **Use `= delete` for mark deleted functions** CORE, HIC

- Mark *destructor* and *move constructor* `noexcept` CORE

- Use braced initializer lists for aggregate types `A{1, 2}` LLVM, Google
- Do not use braced initializer lists `{}` for constructors (at least for containers, e.g. `std::vector`). It can be confused with `std::initializer_list` LLVM
- Prefer braced initializer lists `{}` for constructors to clearly distinguish from function calls and avoid implicit narrowing conversion

※ **Avoid virtual method calls in constructors** Google, Core, Cert

※ **Default arguments are allowed only on *non-virtual* functions**
Google, Core, Hic

∗ **A class with a *virtual function* should have a *virtual or protected destructor***
(e.g. interfaces and abstract classes) Core

- Does not use `virtual` with `final/override` (implicit)

see A hole in Clang's -Wsuggest-override

* **_Multiple inheritance_ and _virtual inheritance_ are discouraged**
  GOOGLE, CHROMIUM
* **Prefer _composition_ to _inheritance_** GOOGLE
* **A polymorphic class should suppress copying** CORE

- **Declare class data members in special way***. Examples:
  - Trailing underscore (e.g. `member_var_`) — Google, μOS, Chromium
  - Leading underscore (e.g. `_member_var`) — .NET
  - Public members (e.g. `m_member_var`) — WebKit

  Personal Comment: Prefer `_member_var` as I read left-to-right and is less invasive

- Class inheritance declarations order: `public`, `protected`, `private` — Google, μOS
- First data members, then function members
- If possible, **avoid** `this->` keyword

* It helps to keep track of class variables and local function variables

* The first character is helpful in filtering through the list of available variables

```
struct A {        // passive data structure
    int   x;
    float y;
};

class B {
public:
    B();
    void public_function();

protected:
    int  _a;                    // in general, it is not public in derived classes
    void _protected_function(); // "protected_function()" is not wrong
                                // it may be public in derived classes
private:
    int   _x;
    float _y;

    void _private_function();
};
```

- In the constructor, each member should be indented on a separate line, e.g.

WebKit, Mozilla

```
A::A(int x1, int y1, int z1) :
    x{x1},
    y{y1},
    z{z1} {
```

Control Flow

※ **Avoid redundant control flow** (see next slide)

- Do not use `else` after a `return` / `break`

  LLVM, Mozilla, Chromium, WebKit

- Avoid `return true/return false` pattern
- Merge multiple conditional statements

∗ **Prefer `switch` to multiple `if`-statement** Core

∗ **Avoid `goto`** μOS, Core

▪ Avoid `do-while` loop Core

▪ Do not use default labels in fully covered switches over enumerations LLVM

```
if (condition) {    // wrong!!
    < code1 >
    return;
}
else // <-- redundant
    < code2 >
//--------------------------
if (condition) {    // Corret
   < code1 >
   return;
}
< code2 >
```

```
if (condition)    // wrong!!
    return true;
else
    return false;
//------------------------
return condition; // Corret
```

- Use *early exits* (`continue` , `break` , `return`) to simplify the code LLVM

```
for (<condition1>) {    // wrong!!
    if (<condition2>)
        ...
}
//-----------------------------
for (<condition1>) {    // Correct
    if (!<condition2>)
        continue;
     ...
}
```

- Turn predicate loops into predicate functions LLVM

```
bool var = ...;
for (<loop_condition1>) { // should be an external
   if (<condition2>) {    // function
       var = ...
       break;
   }
```

※ **Tests for `null/non-null`, and `zero/non-zero` should all be done with equality comparisons**

Core, WebKit
(opposite) Mozilla

```
if (!ptr)   // wrong!!
    return;
if (!count) // wrong!!
    return;
```

```
if (ptr == nullptr) // correct
    return;
if (count == 0)     // correct
    return;
```

※ **Prefer `(ptr == nullptr)` and `x > 0` over `(nullptr == ptr)` and `0 < x`**

Chromium

- Do not compare to `true/false`, e.g. `if (x == true)`

※ **Do not mix `signed` and `unsigned` types** HIC

∗ **Prefer signed integer for loop indices** (better 64-bit) CORE

- Prefer `empty()` method over `size()` to check if a container has no items
  MOZILLA
- Ensure that all statements are reachable HIC

∗ **Avoid *RTTI* (`dynamic_cast`) or *exceptions*** if possible
LLVM, GOOGLE, MOZILLA

※ **The `if` and `else` keywords belong on separate lines**

```
if (c1) <statement1>; else <statement2> // wrong!!
```

GOOGLE, WEBKIT

∗ Multi-lines statements and complex conditions require curly braces GOOGLE

```
if (c1 && ... &&
    c2 && ...) {  // correct
    <statement>
}
```

▪ Curly braces are not required for single-line statements (but allowed)
(`for, while, if`) GOOGLE, WEBKIT

```
if (c1) {  // not mandatory
    <statement>
}
```

Modern C++ Features

Use modern C++ features wherever possible

* `static_cast` `reinterpret_cast` instead of *old style cast* `(type)`
  GOOGLE, μOS, HIC

* **Do not define implicit conversions**. Use the `explicit` keyword for conversion operators and constructors
  GOOGLE, μOS

- Use `constexpr` instead of *macro* — Google, WebKit
- Use `using` instead `typedef`
- Prefer `enum class` instead of plain `enum` — Unreal, μOS
- `static_assert` compile-time assertion — Unreal, Hic
- `lambda` expression — Unreal
- `move` semantic — Unreal
- `nullptr` instead of `0` or `NULL`

LLVM, Google, Unreal, WebKit, Mozilla, Hic, μOS

※ **Use *range-based for loops* whenever possible**

LLVM, WebKit, Unreal, Core

※ **Use `auto`** to avoid type names that are noisy, obvious, or unimportant

```
auto array = new int[10];
auto var   = static_cast<int>(var);
```

lambdas, iterators, template expressions

LLVM, Google
Unreal (only)

* **Use `[[deprecated]]` / `[[noreturn]]` / `[[nodiscard]]`** to indicate deprecated functions / that do not return / result should not be discarded

▪ Avoid `throw()` expression. Use `noexcept` instead

Hic

※ **Always use `override/final` function member keyword**

WebKit, Mozilla, Unreal, Chromium, Hic

* **Use braced *direct-list-initialization* or *copy-initialization*** for setting default data member value. Avoid initialization in constructors if possible — Unreal

```
struct A {
    int x = 3;   // copy-initialization
    int x { 3 }; // direct-list-initialization (best option)
};
```

* Use `= default` constructors
* Use `= delete` to mark deleted functions
- Prefer *uniform initialization* when it cannot be confused with `std::initializer_list` — Chromium

Maintainability

※ **Avoid complicated template programming** GOOGLE

∗ **Write self-documenting code**

e.g. `(x + y - 1) / y` → `ceil_div(x, y)` UNREAL

∗ **Use symbolic names** instead of literal values in code HIC

```
double area1 = 3.14 * radius * radius; // wrong!!
```

```
constexpr auto Pi    = 3.14; // correct
double         area2 = Pi * radius * radius;
```

※ **Do not use `reinterpret_cast` or `union` for type punning** Core, Hic

※ **Enforce const-correctness** Unreal

- but **don't `const` all the things**
  - Pass by-`const` value: almost useless (copy), ABI break
  - `const` return: useless (copy)
  - `const` data member: disable assignment and copy constructor
  - `const` local variables: verbose, rarely effective

※ **Do not overload operators with special semantics `&&`, `^`** Hic

※ **Use `assert` to document preconditions and assumptions** LLVM

* **Address compiler warnings**. Compiler warning messages mean something is wrong UNREAL

* **Ensure ISO C++ compliant code** and avoid non-standard extension, deprecated features, or asm declarations, e.g. `register`, `__attribute__` HIC

* **Prefer** `sizeof(variable/value)` instead of `sizeof(type)` GOOGLE

* **Prefer core-language features** over library facilities, e.g. `uint8_t` vs. `std::byte`

Naming

"Beyond basic mathematical aptitude, the difference between good programmers and great programmers is verbal ability"

Marissa Mayer

- **Naming is hard**. *Most of the time, code is shared with other developers*. It is worth spending a few seconds to find the right name
- Think about the purpose to choose names
- Adopt names commonly used in real contexts (outside the code)
- Don't use the same name for different things. Use a specific name everywhere
- Prefer single English word to implementation-focused, e.g. `UpdateConfigFile()` → `save()`
- Use natural word pair, e.g. `create()/destroy()`, `open()/close()`, `begin()/end()`, `source()/destination()`

- Don't overdecorate, e.g. `Base/Impl`, `Factory/Singleton`
- Don't list the content, e.g. `NameAndAddress` → `ContactInfo`
- Don't repeat class/enum names, e.g. `Employee::EmployeeName`
- Avoid temporal attributes, e.g. `PreLoad()`, `PostLoad()`
- Use adjectives to enrich a name, e.g. `Name` → `FullName`, `Salary` → `AnnualSalary`
- Abbreviations are generally bad, longer names are better in most cases (don't be lazy)

※ **Use whole words**, except in the rare case where an abbreviation would be more canonical and easier to understand, e.g. `tmp` WebKit

* **Avoid short and very long names**. Remember that the average word length in English is 4.8

* The length of a variable should be **proportional to the size of the scope** that contains it. For example, `i` is fine within a loop scope

※ **Do not use reserved names** CERT

- double underscore followed by any character `__var`
- single underscore followed by uppercase `_VAR`

- Use common loop variable names
  - `i, j, k, l` used in order
  - `it` for iterators

Functions Naming

* **Should be descriptive verb** (as they represent actions) — WebKit

* **Should describe their action or effect instead of how they are implemented**, e.g. `partial_sort()` → `top_n()`

* **Functions that return boolean values should start with boolean verbs**, like `is, has, ~~should, does~~` — μOS
  `empty()` → `is_empty()`

- Use `set` prefix for modifier methods `set_value()` — WebKit

- Do not use `get` for observer methods (`const`) without parameters, e.g. `get_size()` → `size()` — WebKit

Style Conventions

Capital Case Uppercase first word letter (sometimes called *Pascal style* or uppercase Camel style) (less readable, shorter names)

```
CapitalStyle
```

Camel-Back style Uppercase first word letter except the first one (less readable, shorter names)

```
camelBack
```

Snake style Lower case words separated by single underscore (good readability, longer names)

```
snake_style
```

Macro style Upper case words separated by single underscore (sometimes called *Screaming style*) (best readability, longer names)

```
MACRO_STYLE
```

Variable Variable names should be nouns

- Capital style e.g. `MyVar` — LLVM, Unreal
- Snake style e.g. `my_var` — Google, Std, μOS

Constant

- Capital style + `k` prefix, e.g. `kConstantVar` — Google, Mozilla
- Macro style e.g. `CONSTANT_VAR` — WebKit, OpenStack

Enum

- Capital style + `k`
  e.g. `enum MyEnum { kEnumVar1, kEnumVar2 }` — Google
- Capital style
  e.g. `enum MyEnum { EnumVar1, EnumVar2 }` — LLVM, WebKit

Namespace
- Snake style, e.g. `my_namespace` — Google, LLVM, Std
- Capital style, e.g. `MyNamespace` — WebKit

Typename Should be nouns
- Capital style (including `classes`, `structs`, `enums`, `typedefs`, etc.)
  e.g. `HelloWorldClass` — LLVM, Google, WebKit
- Snake style — μOS (class), Std

Macro Macro style, e.g. `MY_MACRO` — Google, Std, LLVM

File
- Snake style (`my_file`) — Google
- Capital style (`MyFile`), could lead Windows/Linux conflicts — LLVM

Function Styles

- Camel-back style, e.g. `myFunc()` — LLVM
- Capital style for standard functions
  e.g. `MyFunc()` — Google, Mozilla, Unreal
- Snake style for cheap functions, e.g. `my_func()` — Google, Std

Personal Comment: **Macro style** needs to be used only for macros to avoid subtle bugs. I adopt **snake style** for almost everything as it has the best readability. On the other hand, I don't want to confuse `typenames` and variables, so I use **camel style** for the former ones. Finally, I also use **camel style** for compile-time constants because they are very relevant in my work and I need to quickly identify them

Enforcing Naming Styles

Naming style conventions can be also enforced by using tools like **clang-tidy: readability-identifier-naming**

`.clang-tidy` configuration file

```
Checks:                          'readability-identifier-naming'
HeaderFileExtensions:            ['', 'h','hh','hpp','hxx']
ImplementationFileExtensions: ['c','cc','cpp','cxx']
CheckOptions:
    readability-identifier-naming.ClassCase:           'lower_case'
    readability-identifier-naming.MacroDefinitionCase: 'UPPER_CASE'
```

```
class MyClass {}; // before
#define my_macro
```

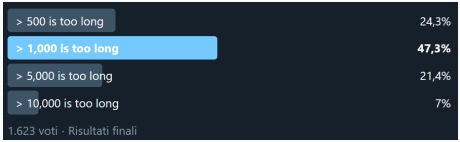
```
class my_class {}; // after
#define MY_MACRO
```

Readability and Formatting

- **Write all code in English**, comments included
- **Limit line length (width)** to be at most **80 characters** long (or 100, or 120) → help code view on a terminal LLVM, Google, Mozilla, μOS

  Personal Comment: I was tempted several times to use a line length > 80 to reduce the number of lines, and therefore improve the readability. Many of my colleagues use split-screens or even the notebook during travels. A line length of **80 columns** is a good compromise for everyone

- Do not write excessive long file



- `Is the 80 character limit still relevant in times of widescreen monitors?`

※ **Use always the same indentation style**

- tab → 2 spaces — Google, Mozilla, Hic, μOS
- tab → 4 spaces — LLVM, Webkit, Hic, μOS
- (actual) tab = 4 spaces — Unreal

Personal Comment: I worked on projects with both two and four-space tabs. I observed less bugs due to indentation and better readability with **four-space tabs**. 'Actual tabs' breaks the line length convention and can introduce tabs in the middle of the code, producing a very different formatting from the original one

※ **Separate commands, operators, etc., by a space** LLVM, Google, WebKit

```
if(a*b<10&&c)          // wrong!!
if (a * c < 10 && c)   // correct
```

* **Prefer consecutive alignment**

```
int           var1     = ...
long long int longvar2 = ...
```

- Minimize the number of empty rows
- Do not use more than one empty line

GOOGLE

What is your threshold for a long source file?

* **Use always the same style for braces**
  - Same line, aka `Kernigham & Ritchie` — WebKit (func. only), Mozilla
  - Its own line, aka `Allman` — Unreal, WebKit (function) Mozilla (class)

```
int main() {
    code
}
```

```
int main()
{
    code
}
```

Personal Comment: C++ is a very verbose language. **Same line** convention helps to keep the code more compact, improving the readability

- Declaration of pointer/reference variables or arguments may be placed with the asterisk/ampersand *adjacent* to either the *type* or to the *variable name* for all symbols in the same way — Google
  - `char* c;` — WebKit, Mozilla, Chromium, Unreal
  - `char *c;`
  - `char * c;`

- The same concept applies to `const`
  - `const int*` *West notation*
  - `int const*` *East notation*

Personal Comment: I prefer **West notation** to prevent unintentional cv-qualify (const/volatile) of a reference or pointer types `char &const p`, see `DCL52-CPP. Never qualify a reference type with const or volatile`

Reduce Code Verbosity

- Use the **short name version** of built-in types, e.g.
  `unsigned` instead of `unsigned int`
  `long long` instead of `long long int`

- **Don't** `const` **all the things**. Avoid Pass by-`const`, `const` return, `const` data member, `const` local variables

- **Use same line braces** for functions and structures

- **Minimize the number of empty rows**

Other Issues

* **Use the same line ending** (e.g. `'\n'`) for all files — Mozilla, Chromium
* **Do not use UTF characters* for portability**, prefer ASCII
* If UTF is needed, **prefer `UTF-8` encoding for portability** — Chromium
- Declare each identifier on a separate line in a separate declaration — Hic, Misra
- Never put trailing white space or tabs at the end of a line — Google, Mozilla
- Only one space between statement and comment — WebKit
- Close files with a blank line — Mozilla, Unreal

* `Trojan Source attack for introducing invisible vulnerabilities`

Code Documentation

* **Any file start with a license** LLVM, Unreal

* **Each file should include**
  - `@author` name, surname, affiliation, email
  - `@date` e.g. year and month
  - `@file` the purpose of the file

  in both header and source files

- Document each entity (functions, classes, namespaces, definitions, etc.) and only in the declarations, e.g. header files

- The first sentence (beginning with `@brief`) is used as an abstract
- Document the input/output parameters `@param[in]`, `@param[out]`, `@param[in,out]`, return value `@return`, and template parameters `@tparam`
- Document ranges, impossible values, status/return values meaning — Unreal
- Use always the same style of comment
- Use anchors for indicating special issues: `TODO`, `FIXME`, `BUG`, etc. — WebKit, Chromium

- Be aware of the comment style, e.g.
  - Multiple lines

```
/**
 * comment1
 * comment2
 */
```

  - Single line

```
/// comment
```

- Prefer `//` comment instead of `/* */` → allow string-search tools like `grep` to identify valid code lines HIC, μOS

- μOS++ Doxygen style guide link
- Teaching the art of great documentation, by Google

Modern C++ Programming

14. Debugging and Testing

Federico Busato

2024-03-29

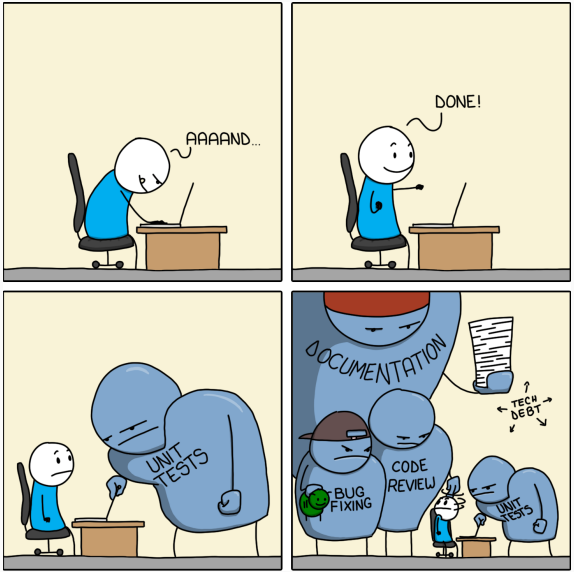
Table of Contents

Table of Contents

Table of Contents

Table of Contents

Feature Complete



Debugging Overview

Is this a bug?

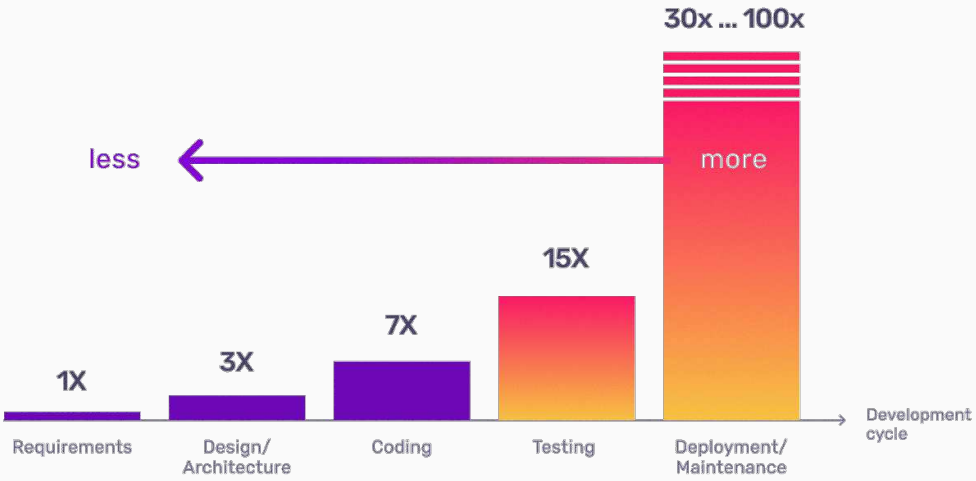
```
for (int i = 0; i <= (2^32) - 1; i++) {
```

> *"Software developers spend 35-50 percent of their time validating and debugging software. The cost of debugging, testing, and verification is estimated to account for 50-75 percent of the total budget of software development projects"*

from: John Regehr (on Twitter)

Errors, Defects, and Failures

- An **error** is a human mistake. *Errors* lead to *software defects*
- A **defects** is an unexpected behavior of the software (correctness, performance, etc.). *Defects* potentially lead to *software failures*
- A **failure** is an observable incorrect behavior


30x ... 100x
less
more
15X
7X
3X
1X
Requirements
Design/
Architecture
Coding
Testing
Deployment/
Maintenance
Development
cycle

Some examples:

- **The Millennium Bug** (2000): $100 billion
- **The Morris Worm** (1988): $10 million (single student)
- **Ariane 5** (1996): $370 million
- **Knight's unintended trades** (2012): $440 million
- **Bitcoin exchange error** (2011): $1.5 million
- **Pentium FDIV Bug** (1994): $475 million
- **Boeing 737 MAX** (2019): $3.9 million

see also:

`11 of the most costly software errors in history`
`Historical Software Accidents and Errors`
`List of software bugs`

Types of Software Defects

Ordered by fix complexity, (time to fix):

(1) Typos, Syntax, Formatting (seconds)

(2) Compilation Warnings/Errors (seconds, minutes)

(3) Logic, Arithmetic, Runtime Errors (minutes, hours, days)

(4) Resource Errors (minutes, hours, days)

(5) Accuracy Errors (hours, days)

(6) Performance Errors (days)

(7) Design Errors (weeks, months)

Causes of Bugs

- *C++ is very error prone language*, see `60 terrible tips for a C++ developer`
- *Human behavior*, e.g. copying & pasting code is very common practice and can introduce subtle bugs $\rightarrow$ check the code carefully, deep understanding of its behavior

Program Errors

A **program error** is a set of conditions that produce an *incorrect result* or *unexpected behavior*, including performance regression, memory consumption, early termination, etc.

We can distinguish between two kind of errors:

Recoverable *Conditions that are not under the control of the program*. They indicates *"exceptional"* run-time conditions. e.g. file not found, bad allocation, wrong user input, etc.

Unrecoverable *It is a synonym of a bug*. It indicates a problem in the program logic. The program must terminate and modified. e.g. out-of-bound, division by zero, etc.

A *recoverable* should be considered *unrecoverable* if it is extremely rare and difficult to handle, e.g. bad allocation due to out-of-memory error

Dealing with Software Defects

Software defects can be identifies by:

Dynamic Analysis A *mitigation* strategy that acts on the runtime state of a program.

Techniques: Print, run-time debugging, sanitizers, fuzzing, unit test support, performance regression tests

Limitations: Infeasible to cover all program states

Static Analysis A *proactive* strategy that examines the source code for (potential) errors.

Techniques: Warnings, static analysis tool, compile-time checks

Limitations: Turing's undecidability theorem, exponential code paths

Assertions

Unrecoverable Errors and Assertions

Unrecoverable errors cannot be handled. They should be prevented by using *assertion* for ensuring *pre-conditions* and *post-conditions*

An **assertion** is a statement to detect a violated assumption. An assertion represents an *invariant* in the code

It can happen both at *run-time* (`assert`) and *compile-time* (`static_assert`). Run-time assertion failures should never be exposed in the normal program execution (e.g. release/public)

Assertion

```cpp
#include <cassert>     // <-- needed for "assert"
#include <cmath>       // std::is_finite
#include <type_traits> // std::is_arithmetic_v

template<typename T>
T sqrt(T value) {
    static_assert(std::is_arithmetic_v<T>,       // precondition
                  "T must be an arithmetic type");
    assert(std::is_finite(value) && value >= 0); // precondition
    int ret = ...                                // sqrt computation
    assert(std::is_finite(value) && ret >= 0 &&  // postcondition
           (ret == 0 || ret == 1 || ret < value));
    return ret;
}
```

Assertions may slow down the execution. They can be disable by define the `NDEBUG` macro

```cpp
#define NDEBUG // or with the flag "-DNDEBUG"
```

Execution Debugging

Execution Debugging (gdb)

How to compile and run for debugging:

```
g++ -O0 -g [-g3] <program.cpp> -o program
gdb [--args] ./program <args...>
```

`-O0` Disable any code optimization for helping the debugger. It is implicit for most compilers

`-g` Enable debugging
- stores the *symbol table information* in the executable (mapping between assembly and source code lines)
- for some compilers, it may disable certain optimizations
- slow down the compilation phase and the execution

`-g3` Produces enhanced debugging information, e.g. macro definitions. Available for most compilers. Suggested instead of `-g`

gdb - Breakpoints

Command	Abbr.	Description
`breakpoint` <*file*>:<*line*>	`b`	insert a breakpoint in a specific line
`breakpoint` <*function_name*>	`b`	insert a breakpoint in a specific function
`breakpoint` <*ref*> `if` <*condition*>	`b`	insert a breakpoint with a conditional statement
`delete`	`d`	delete all breakpoints or watchpoints
`delete` <*breakpoint_number*>	`d`	delete a specific breakpoint
`clear` [*function_name/line_number*]		delete a specific breakpoint
`enable/disable` <*breakpoint_number*>		enable/disable a specific breakpoint
`info breakpoints`	**info** `b`	list all active breakpoints

gdb - Watchpoints / Catchpoints

Command	Abbr.	Description
`watch` <*expression*>		stop execution when the value of `expression` changes (variable, comparison, etc.)
`rwatch` <*variable/location*>		stop execution when `variable/location` is read
`delete` <*watchpoint_number*>	`d`	delete a specific watchpoint
`info watchpoints`		list all active watchpoints
`catch throw`		stop execution when an *exception* is thrown

gdb - Control Flow

Command	Abbr.	Description
`run` [args]	`r`	run the program
`continue`	`c`	continue the execution
`finish`	`f`	continue until the end of the current function
`step`	`s`	execute next line of code (follow function calls)
`next`	`n`	execute next line of code
`until` <*program_point*>		continue until reach line number, function name, address, etc.
`CTRL+C`		stop the execution (not quit)
`quit`	`q`	exit
`help` [<*command*>]	`h`	show help about `command`

gdb - Stack and Info

Command	Abbr.	Description
`list`	`l`	print code
`list` <*function or #start,#end*>	`l`	print function/range code
`up`	`u`	move up in the call stack
`down`	`d`	move down in the call stack
`backtrace [full]`	`bt`	prints stack backtrace (call stack) [local vars]
`info args`		print current function arguments
`info locals`		print local variables
`info variables`		print all variables
`info` <*breakpoints/watchpoints/registers*>		show information about program breakpoints/watchpoints/registers

gdb - Print

Command	Abbr.	Description
`print <variable>`	`p`	print variable
`print/h <variable>`	`p/h`	print variable in hex
`print/nb <variable>`	`p/nb`	print variable in binary (`n` bytes)
`print/w <address>`	`p/w`	print address in binary
`p /s <char array/address>`		print `char` array
`p *array_var@n`		print `n` array elements
`p (int[4])<address>`		print four elements of type `int`
`p *(char**)&<std::string>`		print `std::string`

gdb - Disassemble

Command	Description
`disasseble` <*function_name*>	disassemble a specified function
`disasseble` <`0xStart,0xEnd addr`>	disassemble function range
`nexti` <*variable*>	execute next line of code (follow function calls)
`stepi` <*variable*>	execute next line of code
`x`/`nfu` <*address*>	examine address `n` number of elements, `f` format (**d**: int, **f**: float, etc.), `u` data size (**b**: byte, **w**: word, etc.)

The debugger automatically stops when:

- breakpoint (by using the debugger)
- assertion fail
- segmentation fault
- trigger software breakpoint (e.g. SIGTRAP on Linux)
  `github.com/scottt/debugbreak`

Full story: `www.yolinux.com/TUTORIALS/GDB-Commands.html` (it also contains a script to *de-referencing* STL Containers)

`gdb reference card V5 link`

Memory Debugging

"70% of all the vulnerabilities in Microsoft products are memory safety issues"

***Matt Miller**, Microsoft Security Engineer*

"Chrome: 70% of all security bugs are memory safety issues"

Chromium Security Report

"you can expect at least 65% of your security vulnerabilities to be caused by memory unsafety"

What science can tell us about C and C++'s security

```
Microsoft:  70% of all security bugs are memory safety issues
Chrome:  70% of all security bugs are memory safety issues
What science can tell us about C and C++'s security
```

"Memory Unsafety in Apple's OS represents 66.3%- 88.2% of all the vulnerabilities"

"Out of bounds (OOB) reads/writes comprise ∼70% of all the vulnerabilities in Android"

***Jeff Vander**, Google, Android Media Team*

"Memory corruption issues are the root-cause of 68% of listed CVEs"

***Ben Hawkes**, Google, Project Zero*

Memory Unsafety in Apple's Operating Systems
Google Security Blog: Queue the Hardening Enhancements
Google Project Zero

Terms like *buffer overflow*, *race condition*, *page fault*, *null pointer*, *stack exhaustion*, *heap exhaustion/corruption*, *use-after-free*, or *double free* – all describe ***memory safety vulnerabilities***

Mitigation:

- Run-time check
- Static analysis
- Avoid unsafe language constructs

valgrind is a tool suite to automatically detect many memory management and threading bugs

How to install the last version:

```
$ wget ftp://sourceware.org/pub/valgrind/valgrind-3.21.tar.bz2
$ tar xf valgrind-3.21.tar.bz2
$ cd valgrind-3.21
$ ./configure --enable-lto
$ make -j 12
$ sudo make install
$ sudo apt install libc6-dbg #if needed
```

some linux distributions provide the package through `apt install valgrid`, but it could be an old version

Basic usage:

- compile with `-g`
- ```
  $ valgrind ./program <args...>
  ```

Output example 1:

```
==60127== Invalid read of size 4              !!out-of-bound access
==60127==    at 0x100000D9E: f(int) (main.cpp:86)
==60127==    by 0x100000C22: main (main.cpp:40)
==60127==  Address 0x10042c148 is 0 bytes after a block of size 40 alloc'd
==60127==    at 0x1000161EF: malloc (vg_replace_malloc.c:236)
==60127==    by 0x100000C88: f(int) (main.cpp:75)
==60127==    by 0x100000C22: main (main.cpp:40)
```
```

Output example 2:

```
!!memory leak
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (main.cpp:5)
==19182==    by 0x80483AB: main (main.cpp:11)

==60127== HEAP SUMMARY:
==60127==     in use at exit: 4,184 bytes in 2 blocks
==60127==   total heap usage: 3 allocs, 1 frees, 4,224 bytes allocated
==60127==
==60127== LEAK SUMMARY:
==60127==    definitely lost: 128 bytes in 1 blocks    !!memory leak
==60127==    indirectly lost: 0 bytes in 0 blocks
==60127==      possibly lost: 0 bytes in 0 blocks
==60127==    still reachable: 4,184 bytes in 2 blocks  !!not deallocated
==60127==         suppressed: 0 bytes in 0 blocks
```

Memory leaks are divided into four categories:

- *Definitely lost*
- *Indirectly lost*
- *Still reachable*
- *Possibly lost*

When a program terminates, it releases all heap memory allocations. Despite this, leaving memory leaks is considered a *bad practice* and *makes the program unsafe* with respect to multiple internal iterations of a functionality. If a program has memory leaks for a single iteration, is it safe for multiple iterations?

A **robust program** prevents any memory leak even when abnormal conditions occur

Definitely lost indicates blocks that are *not deleted at the end of the program* (return from the `main()` function). The common case is local variables pointing to newly allocated heap memory

```
void f() {
    int* y = new int[3];  // 12 bytes definitely lost
}

int main() {
    int* x = new int[10]; // 40 bytes definitely lost
    f();
}
```

Indirectly lost indicates blocks pointed by other heap variables that are not deleted. The common case is global variables pointing to newly allocated heap memory

```
struct A {
    int* array;
};

int main() {
    A* x     = new A;       //  8 bytes definitely lost
    x->array = new int[4]; // 16 bytes indirectly lost
}
```

Still reachable indicates blocks that are *not deleted but they are still reachable at the end of the program*

```
int* array;

int main() {
    array = new int[3];
}
// 12 bytes still reachable (global static class could delete it)
```

```
#include <cstdlib>
int main() {
    int* array = new int[3];
    std::abort();                // early abnormal termination
    // 12 bytes still reachable
    ... // maybe it is delete here
}
```

Possibly lost indicates blocks that are still reachable but pointer arithmetic makes the deletion more complex, or even not possible

```
#include <cstdlib>
int main() {
    int* array = new int[3];
    array++;                    // pointer arithmetic
    std::abort();               // early abnormal termination
    // 12 bytes still reachable
    ... // maybe it is delete here but you should be able
        // to revert pointer arithmetic
}
```

Advanced flags:

- `--leak-check=full` print details for each "`definitely lost`" or "`possibly lost`" block, including where it was allocated
- `--show-leak-kinds=all` to combine with `--leak-check=full`. Print all leak kinds
- `--track-fds=yes` list open file descriptors on exit (not closed)
- `--track-origins=yes` tracks the origin of uninitialized values (very slow execution)

```
valgrind --leak-check=full --show-leak-kinds=all
         --track-fds=yes --track-origins=yes ./program <args...>
```

Track stack usage:

```
valgrind --tool=drd --show-stack-usage=yes ./program <args...>
```

Hardening Techniques

References

- Compiler Options Hardening Guide for C and C++ [March, 2024]
- Hardened mode of standard library implementations

Compile-time Stack Usage

- `-Wstack-usage=<byte-size>` Warn if the stack usage of a function might exceed byte-size. The computation done to determine the stack usage is conservative (no VLA)
- `-fstack-usage` Makes the compiler output stack usage information for the program, on a per-function basis
- `-Wvla` Warn if a variable-length array is used in the code
- `-Wvla-larger-than=<byte-size>` Warn for declarations of variable-length arrays whose size is either unbounded, or bounded by an argument that allows the array size to exceed byte-size bytes

Compile-time Stack Protection

- `-Wtrampolines` Check whether the compiler generates trampolines for pointers to nested functions which may interfere with stack virtual memory protection
- `-Wl,-z,noexecstack` Enable data execution prevention by marking stack memory as non-executable

Run-time Stack Usage

- `-fstack-clash-protection` Enables run-time checks for variable-size stack allocation validity
- `-fstack-protector-strong` Enables run-time checks for stack-based buffer overflows using strong heuristic
- `-fstack-protector-all` Enables run-time checks for stack-based buffer overflows for all functions

`_FORTIFY_SOURCE` **define**: the compiler provides buffer overflow checks for the following functions:

`memcpy`, `mempcpy`, `memmove`, `memset`, `strcpy`, `stpcpy`, `strncpy`, `strcat`, `strncat`, `sprintf`, `vsprintf`, `snprintf`, `vsnprintf`, `gets`.

Recent compilers (e.g. GCC 12+, Clang 9+) allow detects buffer overflows with enhanced coverage, e.g. dynamic pointers, with `_FORTIFY_SOURCE=3` *

*GCC's new fortification level: The gains and costs

```cpp
#include <cstring> // std::memset
#include <string>  // std::stoi
int main(int argc, char** argv) {
    int  size = std::stoi(argv[1]);
    char buffer[24];
    std::memset(buffer, 0xFF, size);
}
```

```
$ gcc -O1 -D_FORTIFY_SOURCE program.cpp -o program
$ ./program 12 # OK
$ ./program 32 # Wrong
$ *** buffer overflow detected ***: ./program terminated
```

Standard Library Preconditions

The standard library provides run-time precondition checks for library calls, such as bounds-checks for strings and containers, and null-pointer checks, etc.

`-D_GLIBCXX_ASSERTIONS` for `libstdc++` (GCC)

`-D_LIBCPP_ASSERT`, `_LIBCPP_HARDENING_MODE_EXTENSIVE` for `libc++` (LLVM):

- `-fno-strict-overflow` Prevent code optimization (code elimination) due to signed integer undefined behavior
- `-fwrapv` Signed integer has the same semantic of unsigned integer, with a well-defined wrap-around behavior
- `-fno-strict-aliasing` Strict aliasing means that two objects with the same memory address are not same if they have a different type, undefined behavior otherwise. The flag disables this constraint

- `-fno-delete-null-pointer-checks` NULL pointer dereferencing is undefined behavior and the compiler can assume that it never happens. The flag disable this optimization

- `-ftrivial-auto-var-init[=<hex pattern>]` Ensures that default initialization initializes variables with a fixed 1-byte pattern. Explicit uninitialized variables requires the `[[uninitialized]]` attribute

Control Flow Protections

- `-fcf-protection=full` Enable control flow protection to counter Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks on many x86 architectures

- `-mbranch-protection=standard` Enable branch protection to counter Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks on AArch64

Other Run-time Checks

- `-fPIE -pie` Position-Independent Executable enables the support for address space layout randomization, which makes exploits more difficult.
- `-Wl,-z,relro,-z,now` Prevents modification of the Global Offset Table (locations of functions from dynamically linked libraries) after the program startup
- `-Wl,-z,nodlopen` Restrict `dlopen(3)` calls to shared objects

Sanitizers

Address Sanitizer

Sanitizers are compiler-based instrumentation components to perform *dynamic* analysis

Sanitizer are used during development and testing to discover and diagnose memory misuse bugs and potentially dangerous undefined behavior

Sanitizer are implemented in `Clang` (from 3.1), `gcc` (from 4.8) and `Xcode`

Project using Sanitizers:

- Chromium
- Firefox
- Linux kernel
- Android

Address Sanitizer

`Address Sanitizer` is a memory error detector

- heap/*stack*/*global* out-of-bounds
- memory leaks
- use-after-free, use-after-return, use-after-scope
- double-free, invalid free
- initialization order bugs

* Similar to `valgrind` but faster (50X slowdown)

```
clang++ -O1 -g -fsanitize=address -fno-omit-frame-pointer <program>
```

`-O1` disable inlining

`-g` generate symbol table

- github.com/google/sanitizers/wiki/AddressSanitizer
- gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Leak Sanitizer

`LeakSanitizer` is a run-time *memory leak* detector

- integrated into `AddressSanitizer`, can be used as standalone tool
* almost no performance overhead until the very end of the process

```
g++     -O1 -g -fsanitize=address -fno-omit-frame-pointer <program>
clang++ -O1 -g -fsanitize=leak -fno-omit-frame-pointer <program>
```

- github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer
- gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Memory Sanitizers

`Memory Sanitizer` is a detector of *uninitialized* reads

- stack/heap-allocated memory read before it is written
* Similar to `valgrind` but faster (3X slowdown)

```
clang++ -O1 -g -fsanitize=memory -fno-omit-frame-pointer <program>
```

`-fsanitize-memory-track-origins=2`
track origins of uninitialized values

Note: not compatible with Address Sanitizer

- github.com/google/sanitizers/wiki/MemorySanitizer
- gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Undefined Behavior Sanitizer

`UndefinedBehaviorSanitizer` is an *undefined behavior* detector

- signed integer overflow, floating-point types overflow, enumerated not in range
- out-of-bounds array indexing, misaligned address
- divide by zero
- etc.

* Not included in `valgrind`

```
clang++ -O1 -g -fsanitize=undefined -fno-omit-frame-pointer <program>
```

gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

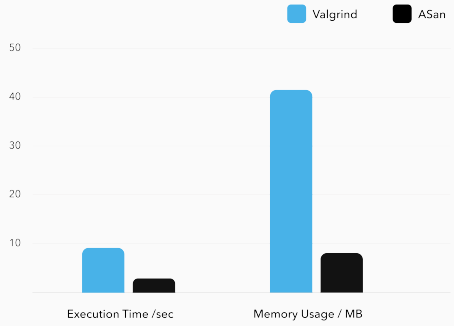
Undefined Behavior Sanitizer

`-fsanitize=<options>`:

`undefined`	All of the checks other than `float-divide-by-zero`, `unsigned-integer-overflow`, `implicit-conversion`, `local-bounds` and the `nullability-*` group of checks
`float-divide-by-zero`	Undefined behavior in C++, but defined by Clang and IEEE-754
`integer`	Checks for undefined or suspicious integer behavior (e.g. unsigned integer overflow)
`implicit-conversion`	Checks for suspicious behavior of implicit conversions
`local-bounds`	Out of bounds array indexing, in cases where the array bound can be statically determined
`nullability`	Checks passing `null` as a function parameter, assigning `null` to an lvalue, and returning `null` from a function

Sanitizers vs. Valgrind

Bug	Valgrind detection	ASan detection
Uninitialized memory read	Yes	No *
Write overflow on heap	Yes	Yes
Write overflow on stack	No	Yes
Read overflow on heap	Yes	Yes
Read underflow on heap	Yes	Yes
Read overflow on stack	No	Yes
Use-after-free	Yes	Yes
Use-after-return	No	Yes
Double-free	Yes	Yes
Memory leak	Yes	Yes
Undefined behavior	No	No **



Debugging Summary

How to Debug Common Errors

Segmentation fault

- gdb, valgrind, sanitizers
- Segmentation fault when just entered in a function → stack overflow

Double free or corruption

- gdb, valgrind, sanitizers

Infinite execution

- gdb + (CTRL + C)

Incorrect results

- valgrind + assertion + gdb + sanitizers

Compiler Warnings

Compiler Warnings - GCC and Clang

Enable specific warnings:

```
g++ -W<warning> <args...>
```

Disable specific warnings:

```
g++ -Wno-<warning> <args...>
```

Common warning flags to minimize accidental mismatches:

`-Wall` Enables many standard warnings (∼50 warnings)

`-Wextra` Enables some extra warning flags that are not enabled by `-Wall` (∼15 warnings)

`-Wpedantic` Issue all the warnings demanded by strict ISO C/C++

Enable ALL warnings, only `clang`: `-Weverything`

Compiler Warnings - MSVC

Enable specific warnings:

```
cl.exe /W<level><warning_id> <args...>
```

Disable specific warnings:

```
cl.exe /We<warning_id> <args...>
```

Common warning flags to minimize accidental mismatches:

`/W1`	Severe warnings
`/W2`	Significant warnings
`/W3`	Production quality warnings
`/W4`	Informational warnings
`/Wall`	All warnings

Static Analysis

Overview

Static analysis is the process of source code examination to find potential issues

Benefits of static code analysis:

- Problem identification before the execution
- Analyze the program outside the execution environment
- The analysis is independent from the run-time tests
- Enforce code quality and compliance by ensuring that the code follows specific rules and standards
- Identify security vulnerabilities

Static Analyzers - Clang and GCC

The **Clang Static Analyzer** (LLVM suite) finds bugs by reasoning about the semantics of code (may produce false positives)

```c
void test() {
    int i, a[10];
    int x = a[i]; // warning: array subscript is undefined
}
```

```
scan-build make
```

The **GCC Static Analyzer** can diagnose various kinds of problems in C/C++ code at compile-time (e.g. double-free, use-after-free, stdio related, etc) by adding the `-fanalyzer` flag

Static Analyzers - cppcheck

The **MSVC Static Analyzer** Enables code analysis and control options (e.g. double-free, use-after-free, stdio related, etc) by adding the `/analyze` flag

cppcheck provides code analysis to detect bugs, undefined behavior and dangerous coding construct. The goal is to detect only real errors in the code (i.e. have very few false positives)

```
cppcheck --enable=warning,performance,style,portability,information,error
         <src_file/directory>
```

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .
cppcheck --enable=<enable_flags> --project=compile_commands.json
```

Popular Static Analyzers - PVS-Studio, SonarLint

PVS-Studio is a high-quality *proprietary* (free for open source projects) static code analyzer supporting C, C++

Customers: IBM, Intel, Adobe, Microsoft, Nvidia, Bosh, IdGames, EpicGames, etc.

SonarSource is a static analyzer which inspects source code for bugs, code smells, and security vulnerabilities for multiple languages (C++, Java, etc.)

SonarLint plugin is available for Visual Code, Visual Studio Code, Eclipse, and IntelliJ IDEA

Other Static Analyzers - FBInfer, DeepCode

FBInfer is a static analysis tool (also available online) to checks for null pointer dereferencing, memory leak, coding conventions, unavailable APIs, etc.

Customers: Amazon AWS, Facebook/Ocolus, Instagram, Whatapp, Mozilla, Spotify, Uber, Sky, etc.

DEEPCODE

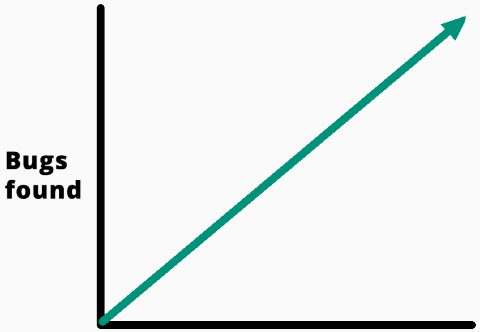
deepCode is an AI-powered code review system, with machine learning systems trained on billions of lines of code from open-source projects

Available for Visual Studio Code, Sublime, IntelliJ IDEA, and Atom

see also: A curated list of static analysis tool

Code Testing

Code Testing



see `Case Study 4: The $440 Million Software Error at Knight Capital`

from: Kat Maddox (on Twitter)

Code Testing

Unit Test A *unit* is the smallest piece of code that can be logically isolated in a system. *Unit test* refers to the verification of a *unit*. It supposes the full knowledge of the code under testing (*white-box* testing)
Goals: meet specifications/requirements, fast development/debugging

Functional Test Output validation instead of the internal structure (*black-box* testing)
Goals: performance, regression (same functionalities of previous version), stability, security (e.g. sanitizers), composability (e.g. integration test)

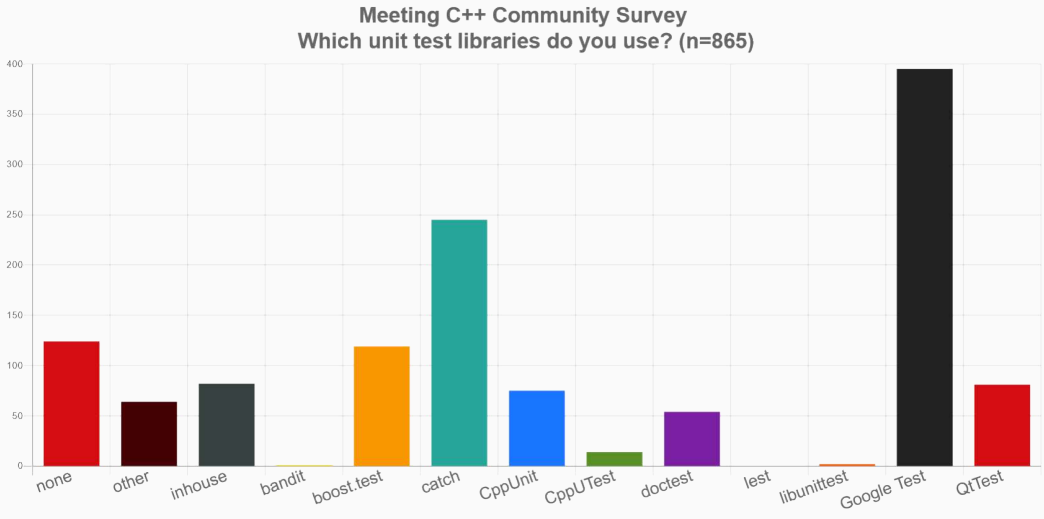
Unit testing involves breaking your program into pieces, and subjecting each piece to a series of tests

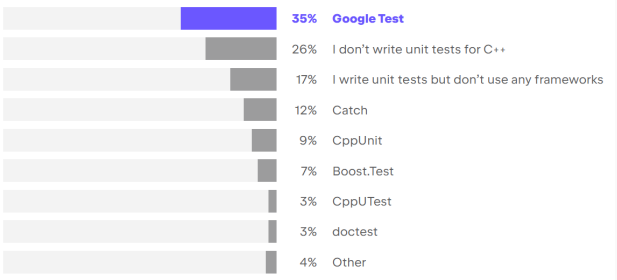
Unit testing should observe the following key features:

- **Isolation**: Each unit test should be *independent* and avoid external interference from other parts of the code
- **Automation**: Non-user interaction, easy to run, and manage
- **Small Scope**: Unit tests focus on small portions of code or specific functionalities, making it easier to identify bugs

Popular C++ Unit testing frameworks:

`catch`, `doctest`, `Google Test`, `CppUnit`, `Boost.Test`


Meeting C++ Community Survey
Which unit test libraries do you use? (n=865)
400
350
300
250
200
150
100
50
0
none
other
inhouse
bandit
boost.test
catch
CppUnit
CppUTest
doctest
lest
libunittest
Google Test
QtTest



The statistic that a quarter of developers aren't writing unit tests freaks me out. I don't feel strongly about how you express those or what framework you use, but we all do need to be writing tests.

Titus Winters
Principal Engineer at Google

Test-Driven Development (TDD)

Unit testing is often associated with the **Test-Driven Development (TDD)** methodology. The practice involves the definition of *automated functional tests* before implementing the functionality

The process consists of the following steps:

1. Write a test for a new functionality
2. Write the minimal code to pass the test
3. Improve/Refactor the code iterating with the test verification
4. Go to 1.

Test-Driven Development (TDD) - Main advantages

- **Software design**. Strong focus on interface definition, expected behavior, specifications, and requirements before working at lower level
- **Maintainability/Debugging Cost** Small, incremental changes allow you to catch bugs as they are introduced. Later refactoring or the introduction of new features still rely on well-defined tests
- **Understandable behavior**. New user can learn how the system works and its properties from the tests
- **Increase confidence**. Developers are more confident that their code will work as intended because it has been extensively tested
- **Faster development**. Incremental changes, high confidence, and automation make it easy to move through different functionalities or enhance existing ones

Catch2 is a multi-paradigm test framework for C++

Catch2 features

- Header only and no external dependencies
- Assertion macro
- Floating point tolerance comparisons

Basic usage:

- Create the test program
- Run the test

```
$ ./test_program [<TestName>]
```

- github.com/catchorg/Catch2
- The Little Things: Testing with Catch2

```cpp
#define CATCH_CONFIG_MAIN  // This tells Catch to provide a main()
#include "catch.hpp"       // only do this in one cpp file

unsigned Factorial(unsigned number) {
    return number <= 1 ? number : Factorial(number - 1) * number;
}

"Test description and tag name"
TEST_CASE( "Factorials are computed", "[Factorial]" ) {
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}

float floatComputation() { ... }

TEST_CASE( "floatCmp computed", "[floatComputation]" ) {
    REQUIRE( floatComputation() == Approx( 2.1 ) );
}
```

Code coverage is a measure used to describe the degree to which the source code of a program is executed when a particular execution/test suite runs

`gcov` and `llvm-profdata/llvm-cov` are tools used in conjunction with compiler instrumentation (`gcc`, `clang`) to interpret and visualize the raw code coverage generated during the execution

`gcovr` and `lcov` are utilities for managing `gcov/llvm-cov` at higher level and generating code coverage results

Step for code coverage:

- Compile with `--coverage` flag (objects + linking)
- Run the program / test
- Visualize the results with `gcovr`, `llvm-cov`, `lcov`

program.cpp:

```cpp
#include <iostream>
#include <string>

int main(int argc, char* argv[]) {
    int value = std::stoi(argv[1]);
    if (value % 3 == 0)
        std::cout << "first\n";
    if (value % 2 == 0)
        std::cout << "second\n";
}
```

```
$ gcc -g --coverage program.cpp -o program
$ ./program 9
first
$ gcovr -r --html --html-details <path>  # generate html
#or
$ lcov --coverage --directory . --output-file coverage.info
$ genhtml coverage.info --output-directory <path> # generate html
```

Code Coverage


```
    1:    4:int main(int argc, char* argv[]) {
    1:    5:    int value = std::stoi(argv[1]);
    1:    6:    if (value % 3 == 0)
    1:    7:        std::cout << "first\n";
    1:    8:    if (value % 2 == 0)
#####:    9:        std::cout << "second\n";
    4:   10:}
```

Current view:	top level - /home/ubuntu/workspace/prove		Hit	Total	Coverage
Test:	coverage.info	Lines:	6	7	85.7 %
Date:	2018-02-09	Functions:	3	3	100.0 %

Filename	Line Coverage		Functions	
program.cpp	85.7 %	6 / 7	100.0 %	3 / 3

Current view:	top level - home/ubuntu/workspace/prove - program.cpp (source / functions)		Hit	Total	Coverage
Test:	coverage.info	Lines:	6	7	85.7 %
Date:	2018-02-09	Functions:	3	3	100.0 %

	Line data	Source code
1		: #include <iostream>
2		: #include <string>
3		:
4	1	: int main(int argc, char* argv[]) {
5	1	: int value = std::stoi(argv[1]); // convert to int
6	1	: if (value % 3 == 0)
7	1	: std::cout << "first";
8	1	: if (value % 2 == 0)
9	0	: std::cout << "second";
10	4	: }

Coverage-Guided Fuzz Testing

A **fuzzer** is a specialized tool that tracks which areas of the code are reached, and generates *mutations* on the corpus of input data in order to *maximize* the code coverage

`LibFuzzer` is the library provided by LLVM and feeds fuzzed inputs to the library via a specific fuzzing entrypoint

The *fuzz target function* accepts an array of bytes and does something interesting with these bytes using the API under test:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* Data,
                                      size_t         Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;
}
```

Code Quality

lint: The term was derived from the name of the undesirable bits of fiber

`clang-tidy` provides an extensible framework for diagnosing and fixing typical *programming errors*, like *style violations*, *interface misuse*, or *bugs* that can be deduced via static analysis

```
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .
$ clang-tidy -p .
```

`clang-tidy` searches the configuration file `.clang-tidy` file located in the closest parent directory of the input file

`clang-tidy` is included in the LLVM suite

Coding Guidelines:

- CERT Secure Coding Guidelines
- C++ Core Guidelines
- High Integrity C++ Coding Standard

Supported Code Conventions:

- Fuchsia
- Google
- LLVM

Bug Related:

- Android related
- Boost library related
- Misc
- Modernize
- Performance
- Readability
- clang-analyzer checks
- bugprone code constructors

`.clang-tidy`

```
Checks: 'android-*,boost-*,bugprone-*,cert-*,cppcoreguidelines-*,
clang-analyzer-*,fuchsia-*,google-*,hicpp-*,llvm-*,misc-*,modernize-*,
performance-*,readability-*'
```

Modern C++ Programming

15. C++ Ecosystem

CMake and Other Tools

Federico Busato

2024-03-29

Table of Contents

4 Other Tools

CMake

CMake Overview

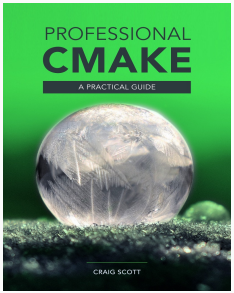
`CMake` is an *open-source*, *cross-platform* family of tools designed to build, test and package software

`CMake` is used to control the software compilation process using simple platform and compiler independent configuration files, and *generate* native `Makefile/Ninja` and workspaces that can be used in the compiler environment of your choice

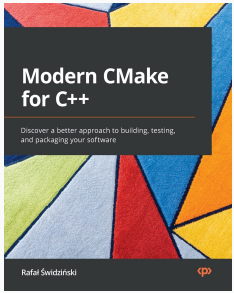
`CMake` features:

- Turing complete language (if/else, loops, functions, etc.)
- Multi-platform (Windows, Linux, etc.)
- Open-Source
- Generate: `makefile, ninja, etc.`
- Supported by many IDEs: `Visual Studio, Clion, Eclipse, etc.`

CMake Books



Professional CMake: A Practical Guide
(14th)
C. Scott, 2023



Modern CMake for C++
R. Świdziński, 2022

CMake - References

- 19 reasons why CMake is actually awesome
- An Introduction to Modern CMake
- Effective Modern CMake
- Awesome CMake
- Useful Variables

Install CMake

Using PPA repository

```
$ wget -O - https://apt.kitware.com/keys/kitware-archive-latest.asc 2>/dev/null |
  gpg --dearmor - | sudo tee /etc/apt/trusted.gpg.d/kitware.gpg >/dev/null
$ sudo apt-add-repository 'deb https://apt.kitware.com/ubuntu/ focal main' # bionic, xenial
$ sudo apt update
$ sudo apt install cmake cmake-curses-gui
```

Using the installer or the pre-compiled binaries: `cmake.org/download/`

```
# download the last cmake package, e.g. cmake-x.y.z-linux-x86_64.sh
$ sudo sh cmake-x.y.z-linux-x86_64.sh
```

A Minimal Example

CMakeLists.txt:

```
project(my_project)                 # project name

add_executable(program program.cpp) # compile command
```

```
# we are in the project root dir
$ mkdir build  # 'build' dir is needed for isolating temporary files
$ cd build
$ cmake ..     # search for CMakeLists.txt directory
$ make         # makefile automatically generated

Scanning dependencies of target program
[100%] Building CXX object CMakeFiles/out_program.dir/program.cpp.o
Linking CXX executable program
[100%] Built target program
```

Parameters and Message

CMakeLists.txt:

```
project(my_project)
add_executable(program program.cpp)

if (VAR)
    message("VAR is set, NUM is ${NUM}")
else()
    message(FATAL_ERROR "VAR is not set")
endif()
```

```
$ cmake ..
VAR is not set
$ cmake -DVAR=ON -DNUM=4 ..
VAR is set, NUM is 4
...
[100%] Built target program
```

Language Properties

```
project(my_project
        DESCRIPTION  "Hello World"
        HOMEPAGE_URL "github.com/"
        LANGUAGES     CXX)

cmake_minimum_required(VERSION 3.15)

set(CMAKE_CXX_STANDARD           14) # force C++14
set(CMAKE_CXX_STANDARD_REQUIRED  ON)
set(CMAKE_CXX_EXTENSIONS         OFF) # no compiler extensions

add_executable(program ${PROJECT_SOURCE_DIR}/program.cpp)  #$
# PROJECT_SOURCE_DIR is the root directory of the project
```

Target Commands

```
add_executable(program) # also add_library(program)

target_include_directories(program
                           PUBLIC  include/
                           PRIVATE src/)
# target_include_directories(program SYSTEM ...) for system headers

target_sources(program                      # best way for specifying
               PRIVATE src/program1.cpp     # program sources and headers
               PRIVATE src/program2.cpp
               PUBLIC  include/header.hpp)

target_compile_definitions(program PRIVATE MY_MACRO=ABCEF)

target_compile_options(program PRIVATE -g)

target_link_libraries(program PRIVATE boost_lib)

target_link_options(program PRIVATE -s)
```

Build Types

```
project(my_project)                          # project name
cmake_minimum_required(VERSION 3.15)         # minimum version

add_executable(program program.cpp)

if (CMAKE_BUILD_TYPE STREQUAL "Debug")       # "Debug" mode
                                             # cmake already adds "-g -O0"
    message("DEBUG mode")
    if (CMAKE_COMPILER_IS_GNUCXX)            # if compiler is gcc
        target_compile_options(program "-g3")
    endif()
elseif (CMAKE_BUILD_TYPE STREQUAL "Release") # "Release" mode
    message("RELEASE mode")                   # cmake already adds "-O3 -DNDEBUG"
endif()
```

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Custom Targets and File Managing

```
project(my_project)
add_executable(program)

add_custom_target(echo_target            # makefile target name
                  COMMAND echo "Hello"   # real command
                  COMMENT "Echo target")

# find all .cpp file in src/ directory
file(GLOB_RECURSE SRCS ${PROJECT_SOURCE_DIR}/src/*.cpp)
# compile all *.cpp file
target_sources(program PRIVATE ${SRCS}) # prefer the explicit file list instead
```

```
$ cmake ..
$ make echo_target
```

Local and Cached Variables

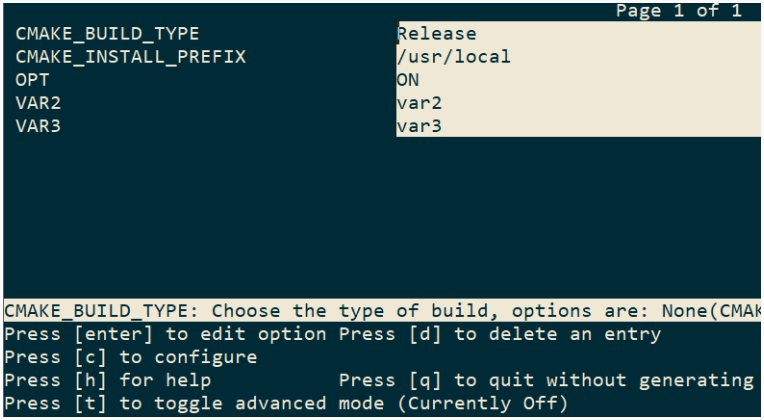
Cached variables can be reused across multiple runs, while *local variables* are only visible in a single run. Cached `FORCE` variables can be modified only after the initialization

```
project(my_project)

set(VAR1 "var1")                                    # local variable
set(VAR2 "var2" CACHE STRING "Description1")        # cached variable
set(VAR3 "var3" CACHE STRING "Description2" FORCE) # cached variable
option(OPT "This is an option" ON)                  # boolean cached variable
                                                    # same of var2
message(STATUS "${VAR1}, ${VAR2}, ${VAR3}, ${OPT}")
```

```
$ cmake .. # var1, var2, var3, ON
$ cmake -DVAR1=a -DVAR2=b -DVAR3=c -DOPT=d .. # var1, b, var3, d
```

Manage Cached Variables

```
$ ccmake .  # or 'cmake-gui'
```



Find Packages

```
project(my_project)                    # project name
cmake_minimum_required(VERSION 3.15) # minimum version

add_executable(program program.cpp)
find_package(Boost 1.36.0 REQUIRED)  # compile only if Boost library
                                       # is found
if (Boost_FOUND)
    target_include_directories("${PROJECT_SOURCE_DIR}/include" PUBLIC ${Boost_INCLUDE_DIRS})
else()
    message(FATAL_ERROR "Boost Lib not found")
endif()
```

Compile Commands

Generate JSON compilation database (`compile_commands.json`)

It contains the exact compiler calls for each file that are used by other tools

```
project(my_project)
cmake_minimum_required(VERSION 3.15)

set(CMAKE_EXPORT_COMPILE_COMMANDS ON)  # <--

add_executable(program program.cpp)
```

Change the C/C++ compiler:

```
CC=clang CXX=clang++ cmake ..
```

CTest is a testing tool (integrated in CMake) that can be used to automate updating, configuring, building, testing, performing memory checking, performing coverage

```
project(my_project)
cmake_minimum_required(VERSION 3.5)
add_executable(program program.cpp)

enable_testing()

add_test(NAME Test1         # check if "program" returns 0
         WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}/build
         COMMAND ./program <args>) # command can be anything

add_test(NAME Test2         # check if "program" print "Correct"
         WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}/build
         COMMAND ./program <args>)

set_tests_properties(Test2
                     PROPERTIES PASS_REGULAR_EXPRESSION "Correct")
```

Basic usage (call `ctest`):

```
$ make test       # run all tests
```

`ctest` usage:

```
$ ctest -R Python      # run all tests that contains 'Python' string
$ ctest -E Iron        # run all tests that not contain 'Iron' string
$ ctest -I 3,5         # run tests from 3 to 5
```

Each `ctest` command can be combined with other tools (e.g. `valgrind`)

`ctest` with Different Compile Options

It is possible to combine a custom target with `ctest` to compile the same code with different compile options

```
add_custom_target(program-compile
                  COMMAND mkdir -p test-release test-ubsan test-asan # create dirs
                  COMMAND cmake .. -B test-release                   # -B change working dir
                  COMMAND cmake .. -B test-ubsan   -DUBSAN=ON
                  COMMAND cmake .. -B test-asan    -DASAN=ON
                  COMMAND make -C test-release -j20 program          # -C run make in a
                  COMMAND make -C test-ubsan   -j20 program          # different dir
                  COMMAND make -C test-asan    -j20 program)

enable_testing()

add_test(NAME Program-Compile
         COMMAND make program-compile)
```

CMake Alternatives - xmake

xmake is a cross-platform build utility based on Lua.

Compared with makefile/CMakeLists.txt, the configuration syntax is more concise and intuitive. It is very friendly to novices and can quickly get started in a short time. Let users focus more on actual project development

Comparison: xmake vs cmake

Code Documentation

Doxygen is the de facto standard tool for generating documentation from annotated C++ sources

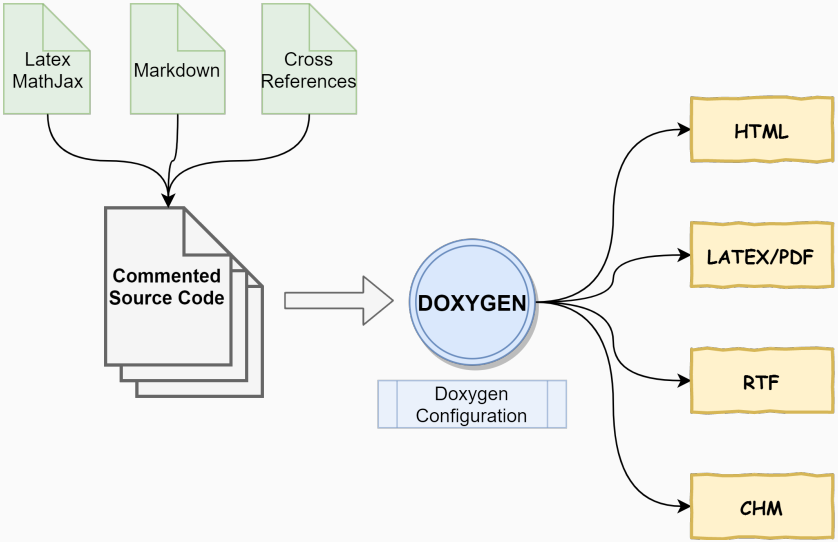
Doxygen usage

- comment the code with `///` or `/** comment */`
- generate `doxygen` base configuration file

```
$ doxygen -g
```

- modify the configuration file `Doxyfile`
- generate the documentation

```
$ doxygen <config_file>
```


Latex
MathJax
Markdown
Cross
References
Commented
Source Code
DOXYGEN
Doxygen
Configuration
HTML
LATEX/PDF
RTF
CHM

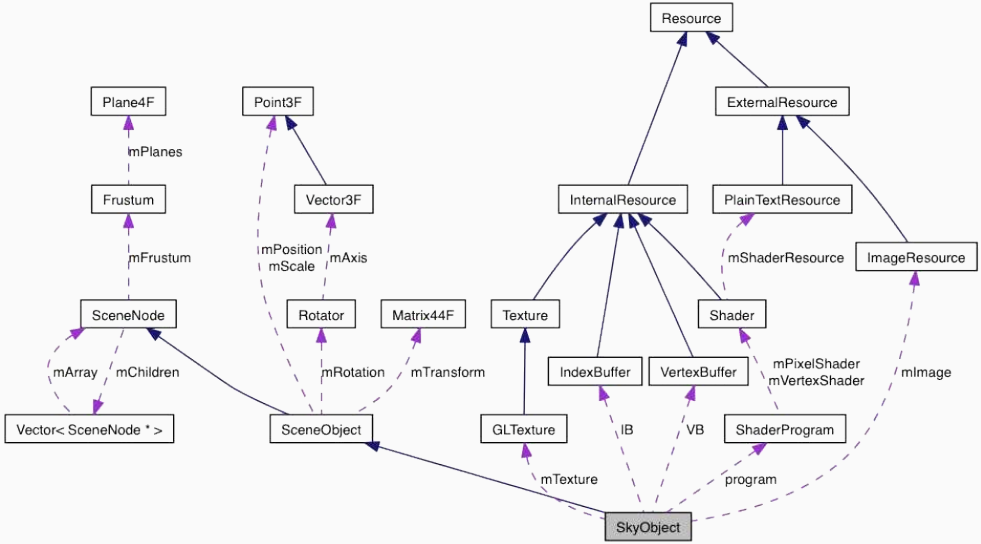
Doxygen requires the following tags for generating the documentation:

- `@file` Document a file
- `@brief` Brief description for an entity
- `@param` Run-time parameter description
- `@tparam` Template parameter description
- `@return` Return value description

- *Automatic cross references* between functions, variables, etc.
- *Specific highlight*. Code `` `<code>` `` , input/output parameters `@param[in] <param>`
- *Latex/MathJax* `$<code>$`
- *Markdown* (Markdown Cheatsheet link), Italic text `*<code>*` , bold text `**<code>**` , table, list, etc.
- Call/Hierarchy graph can be useful in large projects (requires `graphviz`)
  `HAVE_DOT = YES`
  `GRAPHICAL_HIERARCHY = YES`
  `CALL_GRAPH = YES`
  `CALLER_GRAPH = YES`

```
/**
 * @file
 * @copyright MyProject
 * license BSD3, Apache, MIT, etc.
 * @author MySelf
 * @version v3.14159265359
 * @date March, 2018
 */

/// @brief Namespace brief description
namespace my_namespace {

/// @brief "Class brief description"
/// @tparam R "Class template for"
template<typename R>
class A {
```

```
/**
 * @brief "What the function does?"
 * @details "Some additional details",
 *          Latex/MathJax: $\sqrt a$
 * @tparam T Type of input and output
 * @param[in] input Input array
 * @param[out] output Output array
 * @return `true` if correct,
 *         `false` otherwise
 * @remark it is *useful* if ...
 * @warning the behavior is **undefined** if
 *          @p input is `nullptr`
 * @see related_function
 */
template<typename T>
bool my_function(const T* input, T* output);

/// @brief
void related_function();
```


Resource
Plane4F
Point3F
ExternalResource
mPlanes
Frustum
Vector3F
InternalResource
PlainTextResource
mFrustum
mPosition
mScale
mAxis
mShaderResource
ImageResource
SceneNode
Rotator
Matrix44F
Texture
Shader
mArray
mChildren
mRotation
mTransform
IndexBuffer
VertexBuffer
mPixelShader
mVertexShader
mImage
Vector< SceneNode * >
SceneObject
GLTexture
IB
VB
ShaderProgram
mTexture
program
SkyObject

Doxygen Alternatives

`M.CSS`	Doxygen C++ theme
`Doxypress`	Doxygen fork
`clang-doc`	LLVM tool
`Sphinx`	`Clear, Functional C++ Documentation with Sphinx + Breathe + Doxygen + CMake`
`standardese`	The nextgen Doxygen for C++ (experimental)
`HDoc`	The modern documentation tool for C++ (alpha)
`Adobe Hyde`	Utility to facilitate documenting C++

Code Statistics

Count Lines of Code - `cloc`

`cloc` counts blank lines, comment lines, and physical lines of source code in many programming languages

```
$cloc my_project/

4076 text files.
3883 unique files.
1521 files ignored.

http://cloc.sourceforge.net v 1.50  T=12.0 s (209.2 files/s, 70472.1 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                              135          18718          22862         140483
C/C++ Header                   147           7650          12093          44042
Bourne Shell                   116           3402           5789          36882
```

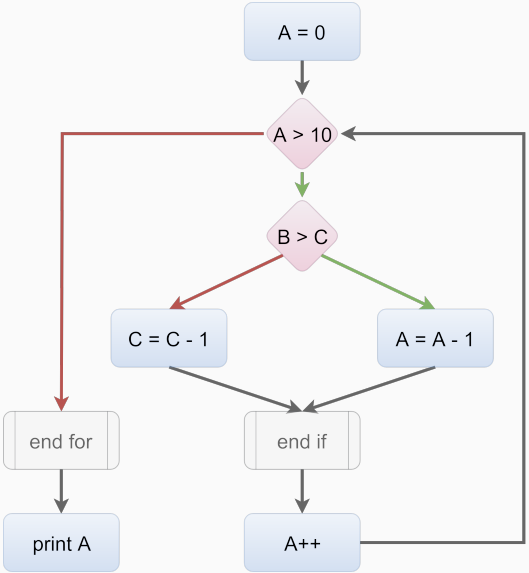
Features: filter by-file/language, SQL database, archive support, line count diff, etc.

`Lizard` is an extensible Cyclomatic Complexity Analyzer for many programming languages including C/C++

Cyclomatic Complexity: is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program source code

```
$lizard my_project/
================================================================
NLOC    CCN  token  param    function@line@file
----------------------------------------------------------------
10      2     29      2     start_new_player@26@./html_game.c
6       1      3      0     set_shutdown_flag@449@./httpd.c
24      3     61      1     server_main@454@./httpd.c
----------------------------------------------------------------
```

- `CCN`: cyclomatic complexity (should not exceed a threshold)
- `NLOC`: lines of code without comments
- `token`: Number of conditional statements



CCN = 3

CC	Risk Evaluation
1-10	a simple program, *without much risk*
11-20	more complex, *moderate risk*
21-50	complex, *high risk*
$>$ **50**	untestable program, *very high risk*

CC	Guidelines
1-5	The routine is probably fine
6-10	Start to think about ways to simplify the routine
$>$ **10**	Break part of the routine

Risk: `Lizard`: 15, `OCLint`: 10

- `www.microsoftpressstore.com/store/code-complete-9780735619678`
- `blog.feabhas.com/2018/07/code-quality-cyclomatic-complexity`

Other Tools

Code Formatting - `clang-format`

`clang-format` is a tool to automatically format C/C++ code (and other languages)

```
$ clang-format <file/directory>
```

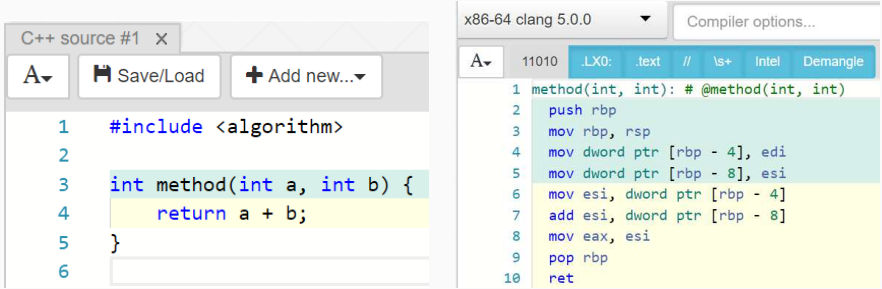
`clang-format` searches the configuration file `.clang-format` file located in the closest parent directory of the input file

`clang-format` example:

```
IndentWidth: 4
UseTab: Never
BreakBeforeBraces: Linux
ColumnLimit: 80
SortIncludes: true
```

Compiler Explorer (assembly and execution)

Compiler Explorer is an interactive tool that lets you type source code and see assembly output, control flow graph, optimization hint, etc.



Key features: support multiple architectures and compilers

Code Transformation - CppInsights

CppInsights See what your compiler does behind the scenes

Code Autocompletion - GitHub CoPilot

CoPilot is an AI pair programmer that helps you write code faster and with less work. It draws context from comments and code to suggest individual lines and whole functions instantly

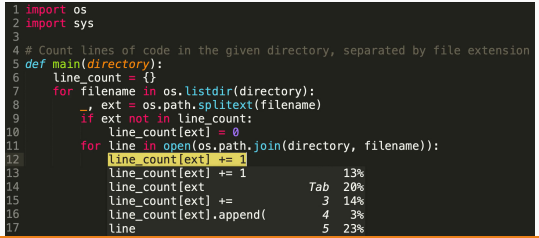
Code Autocompletion - TabNine

TabNine uses deep learning to provide code completion

Features:

- Support all languages
- C++ semantic completion is available through `clangd`
- Project indexing
- Recognize common language patterns
- Use even the documentation to infer this function name, return type, and arguments

Available for Visual Studio Code, IntelliJ, Sublime, Atom, and Vim



Local Code Search - `ugrep`, `ripgrep`, `hypergrep`

`ugrep`, `Ripgrep`, `Hypergrep` are code-searching-oriented tools for regex pattern

Features:

- Default recursively searches
- Skip `.gitignore` patterns, binary and hidden files/directories
- Windows, Linux, Mac OS support
- Up to 100x faster than `GNU grep`

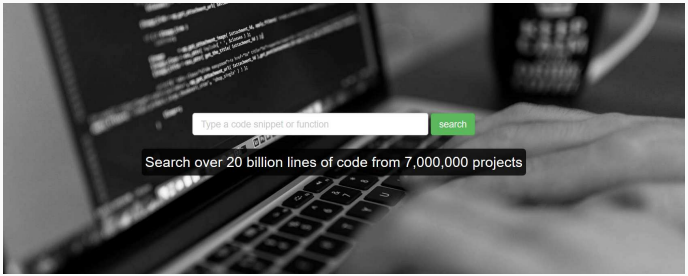
```
[andrew@Cheetah rust] rg -i rustacean
src/doc/book/nightly-rust.md
92:[Mibbit][mibbit]. Click that link, and you'll be chatting with other Rustaceans

src/doc/book/glossary.md
3:Not every Rustacean has a background in systems programming, nor in computer

src/doc/book/getting-started.md
176:Rustaceans (a silly nickname we call ourselves) who can help us out. Other great
376:Cargo is Rust's build system and package manager, and Rustaceans use Cargo to

src/doc/book/guessing-game.md
444:it really easy to re-use libraries, and so Rustaceans tend to write smaller

CONTRIBUTING.md
322:* [rustaceans.org][ro] is helpful, but mostly dedicated to IRC
333:[ro]: http://www.rustaceans.org/
[andrew@Cheetah rust]
```

Code Search Engine - `searchcode`

`Searchcode` is a free source code search engine

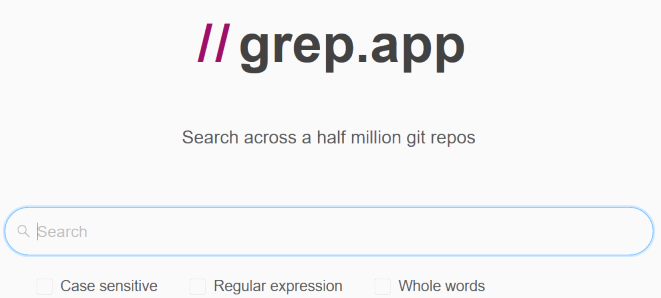
Features:

- Search over 20 billion lines of code from 7,000,000 projects
- Search sources: `github`, `bitbucket`, `gitlab`, `google code`, `sourceforge`, etc.



Code Search Engine - `grep.app`

`grep.app` searches across a half million GitHub repos



Code Benchmarking - `Quick-Bench`

`Quick-benchmark` is a micro benchmarking tool intended to quickly and simply compare the performances of two or more code snippets. The benchmark runs on a pool of AWS machines

Font for Coding

Many editors allow adding optimized fonts for programming which improve legibility and provide extra symbols (ligatures)

Scope	→ ⇒ :: __	-> => :: __
Equality	= == ≠ =/= == === ≠ =/=	== === != =/= == === != !==
Comparisons	⩽ ⩾ ≤ ≥ ⟺	<= >= <= >= <=>

Some examples:

- JetBrain Mono
- Fira Code
- Microsoft Cascadia
- Consolas Ligaturized

Modern C++ Programming

16. Utilities

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

Table of Contents

I/O Stream

I/O Stream

`<iostream>` input/output library refers to a family of classes and supporting functions in the C++ Standard Library that implement stream-based input/output capabilities

There are four predefined iostreams:

- `cin` standard input (`stdin`)
- `cout` standard output (`stdout`) [buffered]
- `cerr` standard error (`stderr`) [unbuffered]
- `clog` standard error (`stderr`) [unbuffered]

buffered: the content of the buffer is not write to disk until some events occur

Basic I/O Stream manipulator:

- `flush` flushes the output stream `cout ≪ flush;`
- `endl` shortcut for `cout ≪ "\n" ≪ flush;`
  `cout ≪ endl`
- `flush` and `endl` force the program to synchronize with the terminal → very slow operation!

- **Set integral representation**: default: dec

```
cout << dec << 0xF;   prints 16
cout << hex << 16;    prints 0xF
cout << oct << 8;     prints 10
```

- Print the underlying **bit representation** of a value:

```
#include <bitset>
std::cout << std::bitset<32>(3.45f); // (32: num. of bits)
// print 01000000010111001100110011001101
```

- **Print true/false text**:

```
cout << boolalpha << 1;   prints true
cout << boolalpha << 0;   prints false
```

`<iomanip>`

- **Set decimal precision**: default: 6

  `cout << setprecision(2) << 3.538;` → 3.54

- **Set float representation**: default: `std::defaultfloat`

  `cout << setprecision(2) << fixed << 32.5;` → 32.50

  `cout << setprecision(2) << scientific << 32.5;` → 3.25e+01

- **Set alignment**: default: right

  `cout << right << setw(7) << "abc" << "##";` → ␣␣␣␣abc##

  `cout << left << setw(7) << "abc" << "##";` → abc␣␣␣␣##

  (better than using tab `\t`)

I/O Stream - `std::cin`

`std::cin` is an example of *input* stream. Data coming from a source is read by the program. In this example `cin` is the standard input

```
#include <iostream>

int main() {
    int a;
    std::cout << "Please enter an integer value:" << endl;
    std::cin >> a;

    int b;
    float c;
    std::cout << "Please enter an integer value "
              << "followed by a float value:" << endl;
    std::cin >> b >> c; // read an integer and store into "b",
}                       // then read a float value, and store
                        // into "c"
```

`ifstream`, `ofstream` are output and input stream too

`<fstream>`

- **Open a file for reading**
  Open a file in input mode: `ifstream my_file("example.txt")`
- **Open a file for writing**
  Open a file in output mode: `ofstream my_file("example.txt")`
  Open a file in append mode: `ofstream my_file("example.txt", ios::out | ios::app)`
- **Read a line** `getline(my_file, string)`
- **Close a file** `my_file.close()`
- **Check the stream integrity** `my_file.good()`

- **Peek the next character**
  `char current_char = my_file.peek()`
- **Get the next character (and advance)**
  `char current_char = my_file.get()`
- **Get the position of the current character in the input stream**
  `int byte_offset = my_file.tellg()`
- **Set the char position in the input sequence**
  `my_file.seekg(byte_offset)` (absolute position)
  `my_file.seekg(byte_offset, position)` (relative position)
  where position can be: `ios::beg` (the begin), `ios::end` (the end), `ios::cur` (current position)

- **Ignore characters until the delimiter is found**
  `my_file.ignore(max_stream_size, <delim>)`
  e.g. skip until end of line \n

- **Get a pointer to the stream buffer object currently associated with the stream**
  `my_file.rdbuf()`
  can be used to redirect file stream

I/O Stream - Example 1

Open a file and print line by line:

```cpp
#include <iostream>
#include <fstream>

int main() {
  std::ifstream fin("example.txt");
  std::string str;
  while (std::getline(fin, str))
     std::cout << str << "\n";
  fin.close();
}
```

An alternative version with redirection:

```cpp
#include <iostream>
#include <fstream>

int main() {
  std::ifstream fin("example.txt");
  std::cout << fin.rdbuf();
  fin.close();
}
```

Reading files line by line in C++ using ifstream

I/O Stream - Example 2

example.txt:

```
23␣70␣␣␣44\n
\t57\t89
```

The input stream is independent from the type of space (multiple space, tab, newline \n, \r\n, etc.)

Another example:

```
#include <iostream>
#include <fstream>

int main() {
    std::ifstream fin("example.txt");
    char c = fin.peek(); // c = '2'
    while (fin.good()) {
        int var;
        fin >> var;
        std::cout << var;
    }        // print 2370445789
    fin.seekg(4);
    c = fin.peek();  // c = '0'
    fin.close();
}
```

I/O Stream -Check the End of a File

- Check the current character

```
while (fin.peek() != std::char_traits<char>::eof()) // C: EOF
    fin >> var;
```

- Check if the read operation fails

```
while (fin >> var)
    ...
```

- Check if the stream past the end of the file

```
while (true) {
    fin >> var
    if (fin.eof())
        break;
}
```

I/O Stream (checkRegularType)

Check if a file is a **regular file** and can be read/written

```
#include <sys/types.h>
#include <sys/stat.h>
bool checkRegularFile(const char* file_path) {
    struct stat info;
    if (::stat( file_path, &info ) != 0)
        return false;        // unable to access
    if (info.st_mode & S_IFDIR)
        return false;             // is a directory
    std::ifstream fin(file_path); // additional checking
    if (!fin.is_open() || !fin.good())
        return false;
    try {                    // try to read
        char c; fin >> c;
    } catch (std::ios_base::failure&) {
        return false;
    }
    return true;
}
```

I/O Stream - File size

Get the **file size** in bytes in a **portable** way:

```
long long int fileSize(const char* file_path) {
    std::ifstream fin(file_path);    // open the file
    fin.seekg(0, ios::beg);          // move to the first byte
    std::istream::pos_type start_pos = fin.tellg();
                                     // get the start offset
    fin.seekg(0, ios::end);          // move to the last byte
    std::istream::pos_type end_pos = fin.tellg();
                                     // get the end offset
    return end_pos - start_pos;      // position difference
}
```

see C++17 file system utilities

Strings and std::print

`std::string` is a wrapper of character sequences

More flexible and safer than raw char array but can be slower

```cpp
#include <string>

int main() {
    std::string a;          // empty string
    std::string b("first");

    using namespace std::string_literals;  // C++14
    std::string c = "second"s;             // C++14
}
```

`std::string` supports `constexpr` in C++20

- `empty()` returns `true` if the string is empty, `false` otherwise
- `size()` returns the number of characters in the string
- `find(string)` returns the position of the first substring equal to the given character sequence or **`npos`** if no substring is found
- `rfind(string)` returns the position of the last substring equal to the given character sequence or **`npos`** if no substring is found
- `find_first_of(char_seq)` returns the position of the first character equal to one of the characters in the given character sequence or `npos` if no characters is found
- `find_last_of(char_seq)` returns the position of the last character equal to one of the characters in the given character sequence or `npos` if no characters is found

`npos` special value returned by string methods

- `new_string substr(start_pos)`
  returns a substring [start_pos, end]
  `new_string substr(start_pos, count)`
  returns a substring [start_pos, start_pos + count)
- `clear()` removes all characters from the string
- `erase(pos)` removes the character at position
  `erase(start_pos, count)`
  removes the characters at positions [start_pos, start_pos + count)
- `replace(start_pos, count, new_string)`
  replaces the part of the string indicated by [start_pos, start_pos + count) with `new_string`
- `c_str()`
  returns a pointer to the raw char sequence

- **access specified character** `string1[i]`
- **string copy** `string1 = string2`
- **string compare** `string1 == string2`
  works also with `!=`, $<$, $\leq$, $>$, $\geq$
- **concatenate two strings** `string_concat = string1 + string2`
- **append characters to the end** `string1 += string2`

Conversion from/to Numeric Values

Converts a string to a numeric value C++11:

- `stoi(string)` string to signed integer
- `stol(string)` string to long signed integer
- `stoul(string)` string to long unsigned integer
- `stoull(string)` string to long long unsigned integer
- `stof(string)` string to floating point value (float)
- `stod(string)` string to floating point value (double)
- `stold(string)` string to floating point value (long double)
- C++17 `std::from_chars(start, end, result, base)` fast string conversion (no allocation, no exception)

Converts a numeric value to a string:

- C++11 `to_string(numeric_value)` numeric value to string

Examples

```
std::string str("si vis pacem para bellum");
cout << str.size();       // print 24
cout << str.find("vis"); // print 3
cout << str.find_last_of("bla"); // print 21, 'l' found

cout << str.substr(7, 5);// print "pacem", pos=7 and count=5
cout << str[1];           // print 'i'
cout << (str == "vis");  // print false
cout << (str < "z");      // print true
const char* raw_str = str.c_str();

cout << string("a") + "b";      // print "ab"
cout << string("ab").erase(0); // print 'b'

char*       str2 = "34";
int         a    = std::stoi(str2);   // a = 34;
std::string str3 = std::to_string(a); // str3 = "34"
```

Tips

- Conversion from integer to char letter (e.g. 3 → 'C'):
  `static_cast<char>('A'+ value)`
  `value` ∈ [0, 26] (English alphabet)

- Conversion from char to integer (e.g. 'C' → 3): `value - 'A'`
  `value` ∈ [0, 26]

- Conversion from digit to char number (e.g. 3 → '3'):
  `static_cast<char>('0'+ value)`
  `value` ∈ [0, 9]

- char to string `std::string(1, char_value)`

C++17 `std::string_view` describes a minimum common interface to interact with string data:

- `const std::string&`
- `const char*`

The purpose of `std::string_view` is to avoid copying data which is already owned by the original object

```cpp
#include <string>
#include <string_view>

std::string str  = "abc";  // new memory allocation + copy
std::string_view = "abc";  // only the reference
```

`std::string_view` provides similar functionalities of `std::string`

```cpp
#include <iostream>
#include <string>
#include <string_view>

void string_op1(const std::string& str) {}
void string_op2(std::string_view   str) {}


string_op1("abcdef"); // allocation + copy
string_op2("abcdef"); // reference

const char* str1 = "abcdef";
std::string str2("abcdef");     // allocation + copy
std::cout << str2.substr(0, 3); // print "abc"

std::string_view str3(str1);    // reference
std::cout << str3.substr(0, 3); // print "abc"
```

`std::string_view` supports `constexpr` constructor and methods

```
constexpr std::string_view str1("abc");
constexpr std::string_view str2 = "abc";

constexpr char c = str1[0];         // 'a'
constexpr bool b = (str1 == str2); // 'true'

constexpr int size = str1.size();                    // '3'
constexpr std::string_view str3 = str1.substr(0, 2); // "ab"

constexpr int pos = str1.find("bc");                 // '1'
```

`printf` *functions*: no automatic type deduction, error prone, not extensible

`stream` *objects*: very verbose, hard to optimize

C++20 `std::format` provides python style formatting:

- Type-safe
- Support positional arguments
- Extensible (support user-defined types)
- Return a `std::string`

Integer formatting

```
std::format("{}", 3);   // "3"
std::format("{:b}", 3); // "101"
```

Floating point formatting

```
std::format("{:.1f}", 3.273); // "3.1"
```

Alignment

```
std::format("{:>6}", 3.27); // "  3.27"
std::format("{:<6}", 3.27); // "3.27  "
```

Argument reordering

```
std::format("{1} - {0}", 1, 3); // "3 - 1"
```

std::print

C++23 introduces `std::print()` `std::println()`

```
std::print("Hello, {}!\n", name);

std::println("Hello, {}!", name);  // prints a newline
```

View

C++20 introduces `std::span` which is a non-owning view of an underlying sequence or array

A `std::span` can either have a *static* extent, in which case the number of elements in the sequence is known at compile-time, or a *dynamic* extent

```
template<
    class       T,
    std::size_t Extent = std::dynamic_extent
> class span;
```

```cpp
#include <span>
#include <array>
#include <vector>

int array1[] = {1, 2, 3};
std::span s1{array1};    // static extent

std::array<int, 3> array2 = {1, 2, 3};
std::span s2{array2};    // static extent

auto array3 = new int[3];
std::span s3{array3, 3}; // dynamic extent

std::vector<int> v{1, 2, 3};
std::span s4{v.data(), v.size()}; // dynamic extent

std::span s5{v};         // dynamic extent
```

```
void f(std::span<int> span) {
    for (auto x : span) // range-based loop (safe)
        cout << x;
    std::fill(span.begin(), span.end(), 3); // std algorithms
}

int array1[] = {1, 2, 3};
f(array1);

auto array2 = new int[3];
f({array2, 3});
```

Math Libraries

<cmath> Math Library

`<cmath>`

- `fabs(x)` computes absolute value, $|x|$, C++11
- `exp(x)` returns e raised to the given power, e^x
- `exp2(x)` returns 2 raised to the given power, 2^x, C++11
- `log(x)` computes natural (base e) logarithm, $\log_e(x)$
- `log10(x)` computes base 10 logarithm, $\log_{10}(x)$
- `log2(x)` computes base 2 logarithm, $\log_2(x)$, C++11
- `pow(x, y)` raises a number to the given power, x^y
- `sqrt(x)` computes square root, $\sqrt{x}$
- `cqrt(x)` computes cubic root, $\sqrt[3]{x}$, C++11

- `sin(x)` computes sine, $\sin(x)$
- `cos(x)` computes cosine, $\cos(x)$
- `tan(x)` computes tangent, $\tan(x)$
- `ceil(x)` nearest integer not less than the given value, $\lceil x \rceil$
- `floor(x)` nearest integer not greater than the given value, $\lfloor x \rfloor$
- `round|lround|llround(x)` nearest integer, $\lfloor x + \frac{1}{2} \rfloor$
  (return type: floating point, `long`, `long long` respectively)

Math functions in C++11 can be applied directly to integral types without implicit/explicit casting (return type: floating point).

`en.cppreference.com/w/cpp/numeric/math`

<limits> Numerical Limits

Get numeric limits of a given type:

<limits> C++11

```
T numeric_limits<T>:: max()  // returns the maximum finite value
                             // value representable

T numeric_limits<T>:: min()  // returns the minimum finite value
                             // value representable

T numeric_limits<T>:: lowest()  // returns the lowest finite
                                // value representable
```

<numeric> Mathematical Constants

`<numeric>` C++20

The header provides numeric constants

- `e` Euler number e
- `pi` π
- `phi` Golden ratio $\frac{1+\sqrt{5}}{2}$
- `sqrt2` $\sqrt{2}$

Integer Division

Integer ceiling division and rounded division:

- **Ceiling Division:** $\left\lceil \frac{\texttt{value}}{\texttt{div}} \right\rceil$

```
unsigned ceil_div(unsigned value, unsigned div) {
    return (value + div - 1) / div;
} // note: may overflow
```

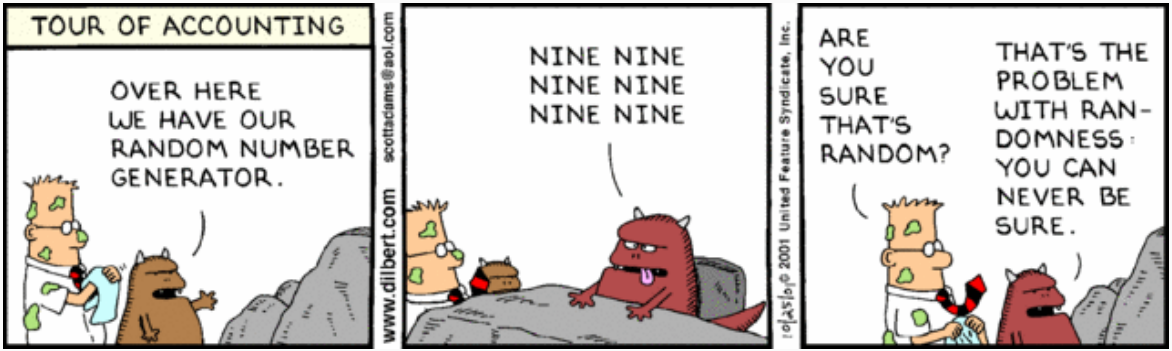
- **Rounded Division:** $\left\lfloor \frac{\texttt{value}}{\texttt{div}} + \frac{1}{2} \right\rfloor$

```
unsigned round_div(unsigned value, unsigned div) {
    return (value + div / 2) / div;
} // note: may overflow
```

Note: do not use floating-point conversion (see `Basic Concept I`)

Random Number

Random Number



"Random numbers should not be generated with a method chosen at random"

*— **Donald E. Knuth***

Applications: cryptography, simulations (e.g. Monte Carlo), etc.

Random Number

see Lavarand

Basic Concepts

- A **pseudorandom (PRNG)** *sequence of numbers* satisfies most of the statistical properties of a truly random sequence but is generated by a *deterministic* algorithm (deterministic finite-state machine)
- A **quasirandom** *sequence of n-dimensional points* is generated by a *deterministic* algorithm designed to fill an *n*-dimensional space evenly
- The **state** of a PRNG describes the status of the generator (the values of its variables), namely where the system is after a certain amount of transitions
- The **seed** is a value that initializes the *starting state* of a PRNG. The same seed always produces the same sequence of results
- The **offset** of a sequence is used to skip ahead in the sequence
- PRNGs produce **uniformly distributed** values. PRNGs can also generate values according to a probability function (binomial, normal, etc.)

The problem:

`C rand()` *function produces poor quality random numbers*

- C++14 discourage the use of `rand()` and `srand()`

C++11 introduces pseudo random number generation (PRNG) facilities to produce random numbers by using combinations of generators and distributions

A random generator requires four steps:

(1) **Select the seed**

(2) **Define the random engine**

```
<type_of_random_engine> generator(seed)
```

(3) **Define the distribution**

```
<type_of_distribution> distribution(range_start, range_end)
```

(4) **Produce the random number**

```
distribution(generator)
```

Simplest example:

```cpp
#include <iostream>
#include <random>

int main() {
    unsigned seed = ...;
    std::default_random_engine generator(seed);
    std::uniform_int_distribution<int> distribution(0, 9);

    std::cout << distribution(generator);  // first random number
    std::cout << distribution(generator);  // second random number
}
```

It generates two random integer numbers in the range [0, 9] by using the default random engine

Given a **seed**, the generator produces always the **same sequence**

The seed could be selected randomly by using the current time:

```
#include <random>
#include <chrono>

unsigned seed = std::chrono::system_clock::now()
                .time_since_epoch().count();
std::default_random_engine generator(seed);
```

`chrono::system_clock::now()` returns an object representing the current point in time

`.time_since_epoch().count()` returns the count of ticks that have elapsed since January 1, 1970 (midnight UTC/GMT)

Problem: Consecutive calls return *very similar* seeds

A **random device** `std::random_device` is a uniformly distributed integer generator that produces non-deterministic random numbers (e.g. from a hardware device)
Note: Not all systems provide a random device

```
#include <random>

std::random_device          rnd_device;
std::default_random_engine generator(rnd_device());
```

`std::seed_seq` consumes a sequence of integer-valued data and produces a number of unsigned integer values in the range $[0, 2^{32} - 1]$. The produced values are distributed over the entire 32-bit range even if the consumed values are close

```cpp
#include <random>
#include <chrono>

unsigned seed1 = std::chrono::system_clock::now()
                 .time_since_epoch().count();
unsigned seed2 = seed1 + 1000;

std::seed_seq seq1{ seed1, seed2 };
std::default_random_engine generator1(seq);

std::random_device rnd;
std::default_random_engine generator1(rnd());
```

PRNG Period and Quality

PRNG Period

The **period** (or **cycle length**) of a PRNG is the length of the sequence of numbers that the PRNG generates before repeating

PRNG Quality

(*informal*) If it is hard to distinguish a generator output from *truly* random sequences, we call it a **high quality** generator. Otherwise, we call it **low quality** generator

Generator	Quality	Period	Randomness
Linear Congruential	Poor	$2^{31} \approx 10^9$	Statistical tests
Mersenne Twister 32/64-bit	High	10^{6000}	Statistical tests
Subtract-with-carry 24/48-bit	Highest	10^{171}	Mathematically proven

Random Engines

- **Linear congruential (LF)**
  The simplest generator engine. Modulo-based algorithm:
  $x_{i+1} = (\alpha x_i + c) mod\ m$ where α, c, m are implementation defined
  C++ Generators: `std::minstd_rand` , `std::minstd_rand0` , `std::knuth_b`

- **Mersenne Twister** *(M. Matsumoto and T. Nishimura, 1997)*
  Fast generation of high-quality pseudorandom number. It relies on Mersenne prime number. (used as default random generator in linux)
  C++ Generators: `std::mt19937` , `std::mt19937_64`

- **Subtract-with-carry (LF)** *(G. Marsaglia and A. Zaman, 1991)*
  Pseudo-random generation based on Lagged Fibonacci algorithm (used for example by physicists at CERN)
  C++ Generators: `std::ranlux24_base` , `std::ranlux48_base` , `std::ranlux24` , `std::ra`

Statistical Tests

The table shows after how many iterations the generator fails the statistical tests

Generator	256M	512M	1G	2G	4G	8G	16G	32G	64G	128G	256G	512G	1T
ranlux24_base	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
ranlux48_base	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
minstd_rand	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
minstd_rand0	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
knuth_b	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
mt19937	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
mt19937_64	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
ranlux24	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ranlux48	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Space and Performance

Generator	Predictability	State	Performance
Linear Congruential	Trivial	4–8 B	**Fast**
Knuth	Trivial	1 KB	**Fast**
Mersenne Twister	Trivial	2 KB	**Good**
randlux_base	Trivial	8–16 B	**Slow**
randlux	Unknown?	∼120 B	**Super slow**

Distribution

- **Uniform distribution** `uniform_int_distribution<T>(range_start, range_end)` where T is integral type

  `uniform_real_distribution<T>(range_start, range_end)` where T is floating point type

- **Normal distribution** $P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

  `normal_distribution<T>(mean, std_dev)`

  where T is floating point type

- **Exponential distribution** $P(x, \lambda) = \lambda e^{-\lambda x}$

  `exponential_distribution<T>(lambda)`

  where T is floating point type

Examples

```
unsigned seed = ...

// Original linear congruential
minstd_rand0  lc1_generator(seed);
// Linear congruential (better tuning)
minstd_rand   lc2_generator(seed);
// Standard mersenne twister (64-bit)
mt19937_64    mt64_generator(seed);
// Subtract-with-carry (48-bit)
ranlux48_base swc48_generator(seed);

uniform_int_distribution<int>    int_distribution(0, 10);
uniform_real_distribution<float> real_distribution(-3.0f, 4.0f);
exponential_distribution<float>  exp_distribution(3.5f);
normal_distribution<double>      norm_distribution(5.0, 2.0);
```

References

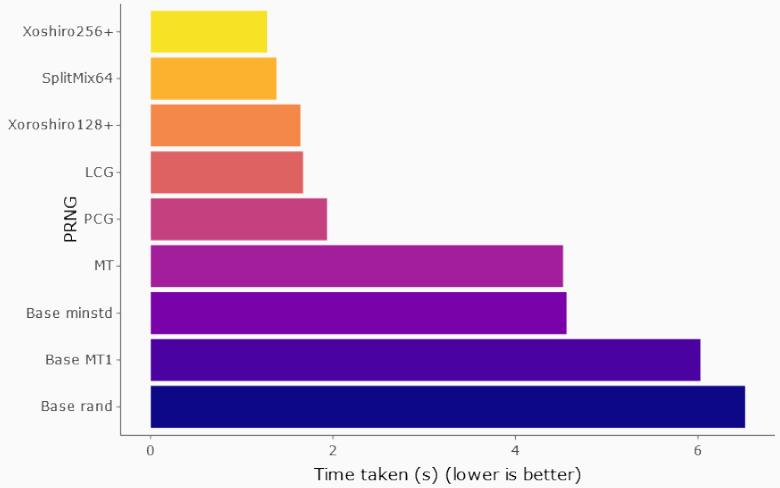
PRNG Quality:

- On C++ Random Number Generator Quality
- It is high time we let go of the Mersenne Twister
- The Xorshift128+ random number generator fails BigCrush

Recent algorithms:

- PCG, A Family of Better Random Number Generators
- Xoshiro / Xoroshiro generators and the PRNG shootout

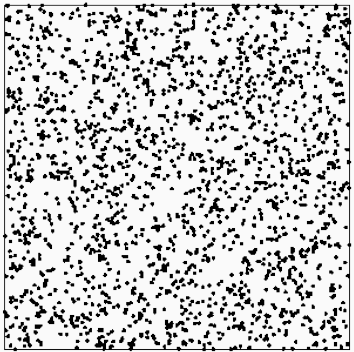
Performance Comparison

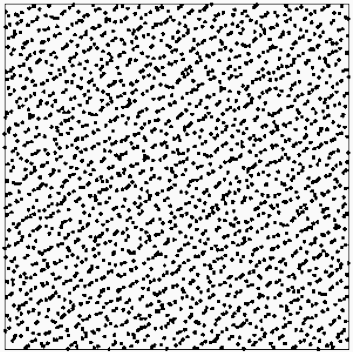


The **quasi-random** numbers have the low-discrepancy property that is a measure of *uniformity for the distribution* of the point for the multi-dimensional case

- Quasi-random sequence, in comparison to pseudo-random sequence, distributes evenly, namely this leads to spread the number over the entire region
- The concept of low-discrepancy is associated with the property that the successive numbers are added in a position as away as possible from the other numbers that is, avoiding *clustering* (grouping of numbers close to each other)

Pseudo-random vs. Quasi random





Time Measuring

Wall-Clock/Real time

It is the human perception of the passage of time from the start to the completion of a task

User/CPU time

The amount of time spent by the CPU to compute in user code

System time

The amount of time spent by the CPU to compute system calls (including I/O calls) executed into kernel code

The *Wall-clock time* measured on a concurrent process platform may include the time elapsed for other tasks

The *User/CPU time* of a multi-thread program is the sum of the execution time of all threads

If the system workload (except the current program) is very low and the program uses only one thread then

```
Wall-clock time = User time + System time
```

`::gettimeofday()` : time resolution $1\mu s$

```
#include <time.h>      //struct timeval
#include <sys/time.h> //gettimeofday()

struct timeval start, end; // timeval {second, microseconds}
::gettimeofday(&start, NULL);
...   // code
::gettimeofday(&end, NULL);

long start_time = start.tv_sec * 1000000 + start.tv_usec;
long end_time   = end.tv_sec * 1000000 + end.tv_usec;
cout << "Elapsed: " << end_time - start_time;  // in microsec
```

Problems: Linux only (not portable), the time is not monotonic increasing (timezone), time resolution is big

`std::chrono` C++11

```
#include <chrono>
auto start_time = std::chrono::system_clock::now();
... // code
auto end_time   = std::chrono::system_clock::now();

std::chrono::duration<double> diff = end_time - start_time;
cout << "Elapsed: " << diff.count(); // in seconds
cout << std::chrono::duration_cast<milli>(diff).count(); // in ms
```

Problems: The time is not monotonic increasing (timezone)

An alternative of `system_clock` is **`steady_clock`** which ensures monotonic increasing time.

`steady_clock` is implemented over `clock_gettime` on POSIX system and has $1ns$ time resolution

```
#include <chrono>
auto start_time = std::chrono::steady_clock::now();
... // code
auto end_time   = std::chrono::steady_clock::now();
```

However, the overhead of C++ API is not always negligible, e.g.
`Linux libstdc++` $\rightarrow$ 20ns, `Mac libc++` $\rightarrow$ 41ns

Time Measuring - User Time

`std::clock`, implemented over `clock_gettime` on POSIX system and has 1*ns* time resolution

```
#include <chrono>

clock_t start_time = std::clock();
... // code
clock_t end_time   = std::clock();

float diff = static_cast<float>(end_time - start_time) / CLOCKS_PER_SEC;
cout << "Elapsed: " << diff; // in seconds
```

Time Measuring - User/System Time

```cpp
#include <sys/times.h>

struct ::tms start_time, end_time;
::times(&start_time);
... // code
::times(&end_time);

auto  user_diff = end_time.tmus_utime - start_time.tms_utime;
auto  sys_diff  = end_time.tms_stime - start_time.tms_stime;
float user      = static_cast<float>(user_diff) / ::sysconf(_SC_CLK_TCK);
float sys       = static_cast<float>(sys_diff)  / ::sysconf(_SC_CLK_TCK);
cout << "user time: "   << user;  // in seconds
cout << "system time: " << sys;   // in seconds
```

Std Classes

`<utility>`

`std::pair` class couples together a pair of values, which may be of different types

Construct a `std::pair`

- `std::pair<T1, T2> pair(value1, value2)`
- `std::pair<T1, T2> pair = {value1, value2}`
- `auto pair = std::make_pair(value1, value2)`

Data members:

- `first` access first field
- `second` access second field

Methods:

- comparison `==`, $<$, $>$, $\geq$, $\leq$
- swap `std::swap`

```cpp
#include <utility>

std::pair<int, std::string> pair1(3, "abc");
std::pair<int, std::string> pair2 = { 4, "zzz" };
auto pair3 = std::make_pair(3, "hgt");

cout << pair1.first;  // print 3
cout << pair1.second; // print "abc"

swap(pair1, pair2);
cout << pair2.first;  // print "zzz"
cout << pair2.second; // print 4

cout << (pair1 > pair2); // print 1
```

`<tuple>`

`std::tuple` is a fixed-size collection of heterogeneous values. It is a generalization of `std::pair`. It allows any number of values

Construct a `std::tuple` (of size 3)

- `std::tuple<T1, T2, T3> tuple(value1, value2, value3)`
- `std::tuple<T1, T2, T3> tuple = {value1, value2, value3}`
- `auto tuple = std::make_tuple(value1, value2, value3)`

Data members:

`std:get<I>(tuple)` returns the *i*-th value of the tuple

Methods:

- comparison `==`, $<$, $>$, $\geq$, $\leq$
- swap `std::swap`

- `auto t3 = std::tuple_cat(t1, t2)`
  concatenate two tuples
- `const int size = std::tuple_size<TupleT>::value`
  returns the number of elements in a tuple at compile-time
- `using T = typename std::tuple_element<TupleT>::type` obtains the type of the specified element
- `std::tie(value1, value2, value3) = tuple`
  creates a tuple of references to its arguments
- `std::ignore`
  an object of unspecified type such that any value can be assigned to it with no effect

```cpp
#include <tuple>
std::tuple<int, float, char> f() { return {7, 0.1f, 'a'}; }

std::tuple<int, char, float> tuple1(3, 'c', 2.2f);
auto tuple2 = std::make_tuple(2, 'd', 1.5f);

cout << std::get<0>(tuple1); // print 3
cout << std::get<1>(tuple1); // print 'c'
cout << std::get<2>(tuple1); // print 2.2f
cout << (tuple1 > tuple2);   // print true

auto concat = std::tuple_cat(tuple1, tuple2);
cout << std::tuple_size<decltype(concat)>::value; // print 6

using T = std::tuple_element<4, decltype(concat)>::type; // T is int
int value1; float value2;
std::tie(value1, value2, std::ignore) = f();
```

`<variant>` C++17

`std::variant` represents a **type-safe union** as the corresponding objects know which type is currently being held

It can be indexed by:

- `std::get<index>(variant)` an integer
- `std::get<type>(variant)` a type

```
#include <variant>

std::variant<int, float, bool> v(3.3f);
int  x = std::get<0>(v);    // return integer value
bool y = std::get<bool>(v); // return bool value
//  std::get<0>(v) = 2.0f;  // run-time exception!!
```

Another useful method is `index()` which returns the position of the type currently held by the variant

```
#include <variant>

std::variant<int, float, bool> v(3.3f);

cout << v.index(); // return 1

std::get<bool>(v) = true
cout << v.index(); // return 2
```

It is also possible to query the index at run-time depending on the type currently being held by providing a **visitor**

```cpp
#include <variant>

struct Visitor {
    void operator()(int& value)   { value *= 2; }

    void operator()(float& value) { value += 3.0f; } // <--

    void operator()(bool& value)  { value = true; }
};

std::variant<int, float, bool> v(3.3f);

std::visit(v, Visitor{});

cout << std::get<float>(v); // 6.3f
```

`<optional>` C++17

`std::optional` provides facilities to represent potential "no value" states

As an example, it can be used for representing the state when an element is not found in a set

```
#include <optional>

std::optional<std::string> find(const char* set, char value) {
    for (int i = 0; i < 10; i++) {
        if (set[i] == value)
            return i;
    }
    return {}; // std::nullopt;
}
```

```cpp
#include <optional>

char set[] = "sdfslgfsdg";
auto x     = find(set, 'a'); // 'a' is not present
if (!x)
    cout << "not found";
if (!x.has_value())
    cout << "not found";

auto y = find(set, 'l');
cout << *y << " " << y.value(); // print '4' '4'

x.value_or(-1); // returns '-1'
y.value_or(-1); // returns '4'
```

std::any

`<any>` C++17

`std::any` holds arbitrary values and provides **type-safety**

```
#include <any>

std::any var = 1;           // int
cout << var.type().name(); // print 'i'

cout << std::any_cast<int>(var);
// cout << std::any_cast<float>(var); // exception!!

var = 3.14; // double
cout << std::any_cast<double>(var);

var.reset();
cout << var.has_value(); // print 'false'
```

C++23 introduces `std::stacktrace` library to get the current function call stack, namely the sequence of calls from the `main()` entry point

```cpp
#include <print>
#include <stacktrace> // the program must be linked with the library
                      // -lstdc++_libbacktrace
                      // (-lstdc++exp with gcc-14 trunk)
void g() {
    auto call_stack = std::stacktrace::current();
    for (const auto& entry : call_stack)
        std::print("{}\n", entry);
}

void f() { g(); }

int main() { f(); }
```

the previous code prints

```
 g() at /app/example.cpp:6
 f() at /app/example.cpp:11
main at /app/example.cpp:13
     at :0
__libc_start_main at :0
_start at :0
```

The library also provides additional functions for `entry` to allow fine-grained control of the output `description()`, `source_file()`, `source_line()`

```
for (const auto& entry : call_stack) { // same output
    std::print("{} at {}:{}\n", entry.description(), entry.source_file(),
                                entry.source_line());
}
```

Filesystem Library

Filesystem Library

C++17 introduces abstractions and facilities for performing operations on file systems and their components, such as **paths**, **files**, and **directories**

- Follow the Boost filesystem library
- Based on POSIX
- Fully-supported from `clang 7`, `gcc 8`, etc.
- Work on Windows, Linux, Android, etc.

Basic concepts

- **file**: a file system object that holds data
  - **directory** a container of directory entries
  - **hard link** associates a name with an existing file
  - **symbolic link** associates a name with a path
  - **regular file** a file that is not one of the other file types
- **file name**: a string of characters that names a file. Names `.` (dot) and `..` (dot-dot) have special meaning at library level
- **path**: sequence of elements that identifies a file
  - **absolute path**: a path that unambiguously identifies the location of a file
  - **canonical path**: an absolute path that includes no symlinks, `.` or `..` elements
  - **relative path**: a path that identifies a file relative to some location on the file system

path Object

A `path` object stores the pathname in native form

```
#include <filesystem> // required
namespace fs = std::filesystem;

fs::path p1 = "/usr/lib/sendmail.cf"; // portable format
fs::path p2 = "C:\\users\\abcdef\\";  // native format

cout << "p1: " << p1;           // /usr/lib/sendmail.cf
cout << "p2: " << p2;           // C:\users\abcdef\

out << "p3: " << p2 + "xyz\\"; // C:\users\abcdef\xyz\
```

`path` Methods

Decomposition (member) methods:

- **Return root-name of the path**
  `root_name()`
- **Return path relative to the root path**
  `relative_path()`
- **Return the path of the parent path**
  `parent_path()`
- **Return the filename path component**
  `filename()`
- **Return the file extension path component**
  `extension()`

Filesystem Methods - Query

- **Check if a file or path exists**
  `exists(path)`
- **Return the file size**
  `file_size(path)`
- **Check if a file is a directory**
  `is_directory(path)`
- **Check if a file (or directory) is empty**
  `is_empty(path)`
- **Check if a file is a regular file**
  `is_regular_file(path)`
- **Returns the current path**
  `current_path()`

Directory Iterators

Iterate over files of a directory (recursively/non-recursively)

```
#include <filesystem>

namespace fs = std::filesystem;

for(auto& path : fs::directory_iterator("/usr/tmp/"))
    cout << path << '\n';

for(auto& path : fs::recursive_directory_iterator("/usr/tmp/"))
    cout << path << '\n';
```

Filesystem Methods - Modify

- **Copy files or directories**
  `copy(path1, path2)`
- **Copy files**
  `copy_file(src_path, src_path, [fs::copy_options::recursive])`
- **Create new directory**
  `create_directory(path)`
- **Remove a file or empty directory**
  `remove(path)`
- **Remove a file or directory and all its contents, recursively**
  `remove_all(path)`
- **Rename a file or directory**
  `rename(old_path, new_path)`

Examples

```cpp
#include <filesystem> // required

namespace fs = std::filesystem;
fs::path p1 = "/usr/tmp/my_file.txt";

cout << p1.exists();          // true
cout << p1.parent_path();     // "/usr/tmp/"
cout << p1.filename();        // "my_file"
cout << p1.extension();       // "txt"
cout << p1.is_directory();    // false
cout << p1.is_regular_file(); // true

fs::create_directory("/my_dir/");
fs::copy(p1.parent_path(), "/my_dir/", fs::copy_options::recursive);
fs::copy_file(p1, "/my_dir/my_file2.txt");
fs::remove(p1);
fs::remove_all(p1.parent_path());
```

Modern C++ Programming

17. Containers, Iterators, Ranges, and Algorithms

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Containers and Iterators

Containers and Iterators

Container

A **container** is a class, a data structure, or an abstract data type, whose instances are collections of other objects

- *Containers* store objects following specific access rules

Iterator

An **iterator** is an object allowing to traverse a container

- *Iterators* are a generalization of pointers
- A pointer is the simplest *iterator*, and it supports all its operations

C++ Standard Template Library (STL) is strongly based on *containers* and *iterators*

Reasons to use Standard Containers

- STL containers eliminate redundancy, and save time avoiding writing your own code (productivity)
- STL containers are implemented correctly, and they do not need to spend time to debug (reliability)
- STL containers are well-implemented and fast
- STL containers do not require external libraries
- STL containers share common interfaces, making it simple to utilize different containers without looking up member function definitions
- STL containers are well-documented and easily understood by other developers, improving the understandability and maintainability
- STL containers are thread safe. Sharing objects across threads preserve the consistency of the container

Container Properties

C++ Standard Template Library (STL) Containers have the following properties:

- Default constructor
- Destructor
- Copy constructor and assignment (deep copy)
- Iterator methods `begin()`, `end()`
- Support `std::swap`
- Content-based and order equality (`==`, `!=`)
- Lexicographic order comparison (`>`, `>=`, `<`, `<=`)
- `size()` *, `empty()`, and `max_size()` methods

* except for `std::forward_list`

Iterator Concept

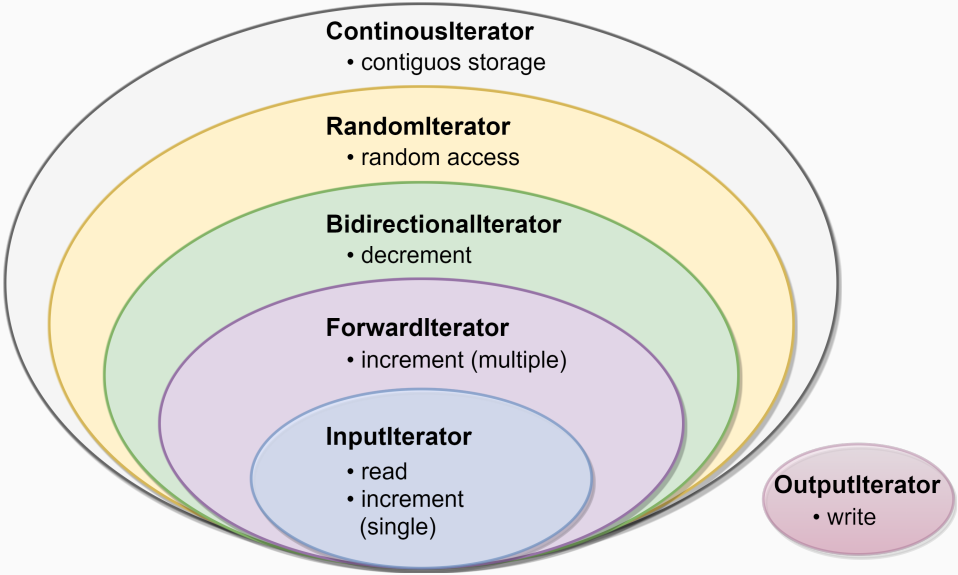
STL containers provide the following methods to get iterator objects:

- `begin()` returns an iterator pointing to the first element
- `end()` returns an iterator pointing to the end of the container (i.e. the element after the last element)

There are different categories of **iterators** and each of them supports a subset of the following operations:

Operation	Example
Read	`*it`
Write	`*it =`
Increment	`it++`
Decrement	`it--`
Comparison	`it1 < it2`
Random access	`it + 4`, `it[2]`

Iterator Categories/Tags



Iterator

- Copy Constructible `It(const It&)`
- Copy Assignable `It operator=(const It&)`
- Destructible `~X()`
- Dereferenceable `It_value& operator*()`
- Pre-incrementable `It& operator++()`

Input/Output Iterator

- Satisfy `Iterator`
- Equality `bool operator==(const It&)`
- Inequality `bool operator!=(const It&)`
- Post-incrementable `It operator++(int)`

Forward Iterator

- Satisfy `Input/Output Iterator`
- Default constructible `It()`

Bidirectional Iterator

- Satisfy `Forward Iterator`
- Pre/post-decrementable `It& operator--(), It operator--(int)`

Random Access Iterator

- Satisfy `Bidirectional Iterator`
- Addition/Subtraction
  `void operator+(const It& it)`, `void operator+=(const It& it)`,
  `void operator-(const It& it)`, `void operator-=(const It& it)`
- Comparison
  `bool operator<(const It& it)`, `bool operator>(const It& it)`,
  `bool operator<=(const It& it)`, `bool operator>=(const It& it)`
- Subscripting `It_value& operator[](int index)`

anderberg.me/2016/07/04/c-custom-iterators/

Sequence Containers

Overview

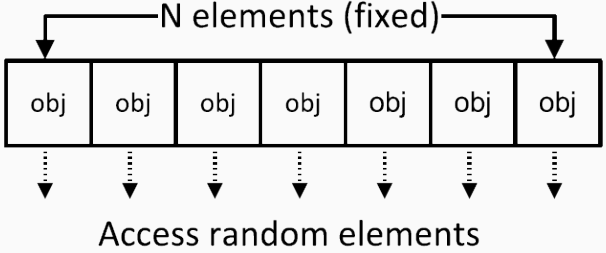
Sequence containers are data structures storing objects of the same data type in a linear mean manner

The *STL Sequence Container* types are:

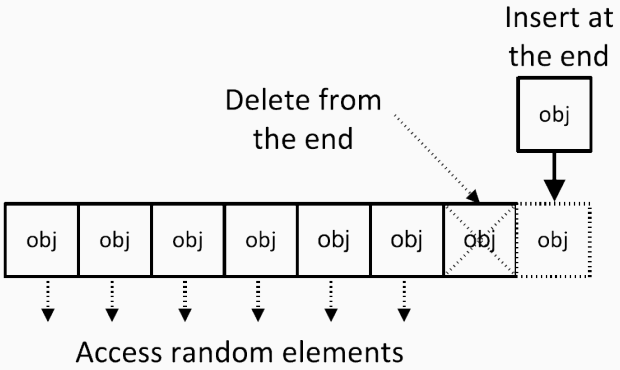
- `std::array` provides a *fixed-size* contiguous array (on stack)
- `std::vector` provides a *dynamic* contiguous array (`constexpr` in C++20)
- `std::list` provides a *double-linked list*
- `std::deque` provides a *double-ended queue* (implemented as array-of-array)
- `std::forward_list` provides a *single-linked list*

While `std::string` is not included in most container lists, it actually meets the requirements of a Sequence Container

std::array

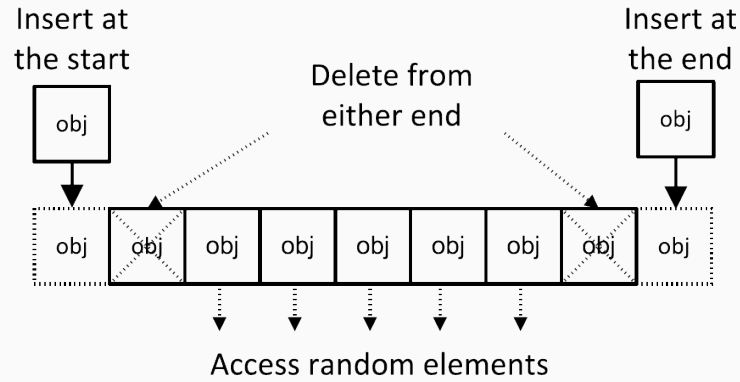

N elements (fixed)
obj
obj
obj
obj
obj
obj
obj
Access random elements

std::vector



Other methods:

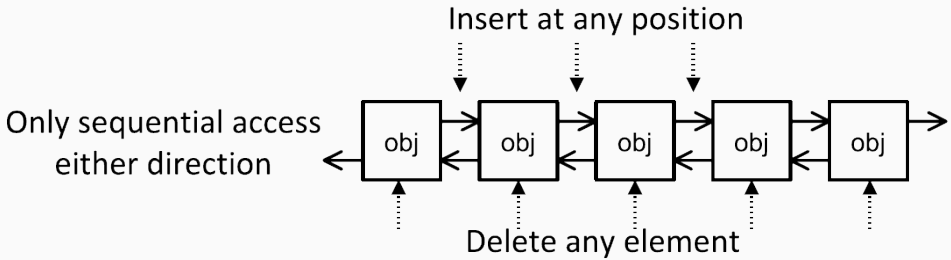
- `resize()` resizes the allocated elements of the container
- `capacity()` number of allocated elements
- `reserve()` resizes the allocated memory of the container (not size)
- `shrink_to_fit()` reallocate to remove unused capacity
- `clear()` removes all elements from the container (no reallocation)



Other methods:

- `resize()` resizes the allocated elements of the container
- `shrink_to_fit()` reallocate to remove unused capacity
- `clear()` removes all elements from the container (no reallocation)

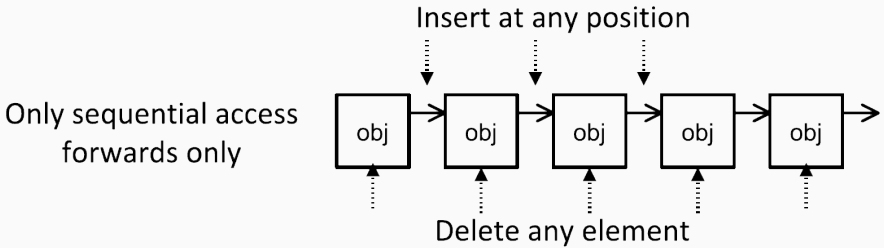
std::list



Other methods:

- `resize()` resizes the allocated elements of the container
- `shrink_to_fit()` reallocate to remove unused capacity
- `clear()` removes all elements from the container (no reallocation)
- `remove()` removes all elements satisfying specific criteria
- `reverse()` reverses the order of the elements
- `unique()` removes all consecutive duplicate elements
- `sort()` sorts the container elements

std::forward_list



Other methods:

- `resize()` resizes the allocated elements of the container
- `shrink_to_fit()` reallocate to remove unused capacity
- `clear()` removes all elements from the container (no reallocation)
- `remove()` removes all elements satisfying specific criteria
- `reverse()` reverses the order of the elements
- `unique()` removes all consecutive duplicate elements
- `sort()` sorts the container elements

Supported Operations and Complexity

CONTAINERS	operator[]/at	front	back
std::array	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
std::vector	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
std::list		$\mathcal{O}(1)$	$\mathcal{O}(1)$
std::deque	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
std::forward_list		$\mathcal{O}(1)$	

CONTAINERS	push_front	pop_front	push_back	pop_back	insert$^{(it)}$	erase$^{(it)}$
std::array						
std::vector			$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
std::list	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
std::deque	$\mathcal{O}(1)^*$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)^*/\mathcal{O}(n)^\dagger$	$\mathcal{O}(1)$
std::forward_list	$\mathcal{O}(1)$	$\mathcal{O}(1)$			$\mathcal{O}(1)$	$\mathcal{O}(1)$

*Amortized time †Worst case (middle insertion)

std::array example

```cpp
#include <algorithm> // std::sort
#include <array>
// std::array supports initialization only throw initialization list
std::array<int, 3> arr1 = { 5, 2, 3 };
std::array<int, 4> arr2 = { 1, 2 };          // [3]: 0, [4]: 0
// std::array<int, 3> arr3 = { 1, 2, 3, 4 }; // compiler error
std::array<int, 3> arr4(arr1);               // copy constructor
std::array<int, 3> arr5 = arr1;              // assign operator

arr5.fill(3);                          // equal to { 3, 3, 3 }
std::sort(arr1.begin(), arr1.end()); // arr1: 2, 3, 5
cout << (arr1 >= arr5);              // true

cout << sizeof(arr1);           // 12
cout << arr1.size();            // 3
for (const auto& it : arr1)
    cout << it << ", ";         // 2, 3, 5
cout << arr1[0];                // 2
cout << arr1.at(0);             // 2, throw if the index is not within the range
cout << arr1.data()[0];         // 2 (raw array)
```

std::vector example

```cpp
#include <vector>

std::vector<int>         vec1   { 2, 3, 4 };
std::vector<std::string> vec2 = { "abc", "efg" };
std::vector<int>         vec3(2);     // [0, 0]
std::vector<int>         vec4{2};     // [2]
std::vector<int>         vec5(5, -1); // [-1, -1, -1, -1, -1]

fill(vec5.begin(), vec5.end(), 3);    // equal to { 3, 3, 3 }
cout << sizeof(vec1);                 // 24
cout << vec1.size();                  // 3
for (const auto& it : vec1)
    std::cout << it << ", ";          // 2, 3, 4

cout << vec1[0];         // 2
cout << vec1.at(0);      // 2 (bound check)
cout << vec1.data()[0]   // 2 (raw array)
vec1.push_back(5);       // [2, 3, 4, 5]
```

std::list example

```cpp
#include <list>

std::list<int>         list1   { 2, 3, 2 };
std::list<std::string> list2 = { "abc", "efg" };
std::list<int>         list3(2);          // [0, 0]
std::list<int>         list4{2};          // [2]
std::list<int>         list5(2, -1);      // [-1, -1]
std::fill(list5.begin(), list5.end(), 3); // [3, 3]

list1.push_back(5);                       // [2, 3, 2, 5]
list1.sort();                             // [2, 2, 3, 5]
list1.merge(list5);                       // [-1, -1, 2, 2, 3, 5] merge two sorted lists
list1.remove(2);                          // [-1, -1, 3, 5]
list1.unique();                           // [-1, 3, 5]
list1.reverse();                          // [5, 3, -1]
```

std::deque example

```cpp
#include <deque>

std::deque<int>         queue1   { 2, 3, 2 };
std::deque<std::string> queue2 = { "abc", "efg" };
std::deque<int>         queue3(2);          // [0, 0]
std::deque<int>         queue4{2};          // [2]
std::deque<int>         queue5(2, -1);      // [-1, -1]
std::fill(queue5.begin(), queue5.end(), 3); // [3, 3]

queue1.push_front(5);                       // [5, 2, 3, 2]
queue1[0];                                  // retuns 5
```

std::forward_list example

```cpp
#include <forward_list>

std::forward_list<int>         flist1   { 2, 3, 2 };
std::forward_list<std::string> flist2 = { "abc", "efg" };
std::forward_list<int>         flist3(2);     // [0, 0]
std::forward_list<int>         flist4{2};     // [2]
std::forward_list<int>         flist5(2, -1); // [-1, -1]
std::fill(flist5.begin(), flist5.end(), 4);   // [4, 4]

flist1.push_front(5);                    // [5, 2, 3, 2]
flist1.insert_after(flist1.begin(), 0); // [5, 0, 2, 3, 2]
flist1.erase_after(flist1.begin());      // [5, 2, 3, 2]
flist1.remove(2);                        // [5, 3, 3]
flist1.unique();                         // [5, 3]
flist1.reverse();                        // [3, 5]
flist1.sort();                           // [3, 5]
flist1.merge(flist5);                    // [3, 4, 4, 5] merge two sorted lists
```

Associative Containers

Overview

An **associative container** is a collection of elements not necessarily indexed with sequential integers and that supports efficient retrieval of the stored elements through keys

Keys are unique

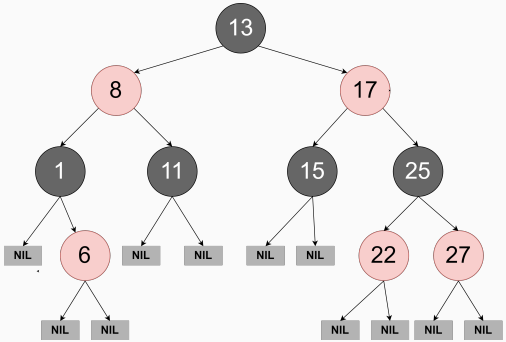
- `std::set` is a collection of sorted unique elements (`operator<`)
- `std::unordered_set` is a collection of unsorted unique keys
- `std::map` is a collection of unique <key, value> pairs, sorted by keys
- `std::unordered_map` is a collection of unique <key, value> pairs, unsorted

Multiple entries for the same key are permitted

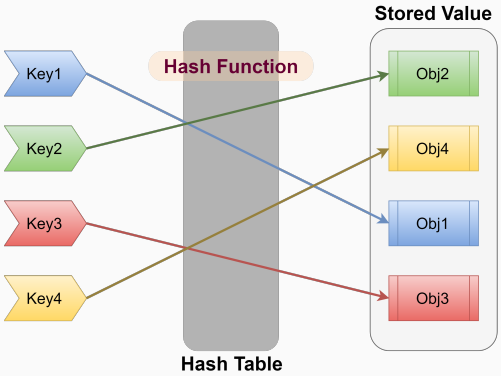
- `std::multiset` is a collection of sorted elements (`operator<`)
- `std::unordered_multiset` is a collection of unsorted elements
- `std::multimap` is a collection of <key, value> pairs, sorted by keys

Internal Representation

Sorted associative containers are typically implemented using *red-black trees*, while unordered associative containers (C++11) are implemented using *hash tables*



Red-Black Tree



Hash Table

Supported Operations and Complexity

CONTAINERS	insert	erase	count	find	lower_bound upper_bound
Ordered Containers	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$
Unordered Containers	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	

* $\mathcal{O}(n)$ worst case

- `count()` returns the number of elements with `key` equal to a specified argument
- `find()` returns the element with `key` equal to a specified argument
- `lower_bound()` returns an iterator pointing to the first element that is *not less* than `key`
- `upper_bound()` returns an iterator pointing to the first element that is *greater* than `key`

Other Methods

Ordered/Unordered containers:

- `equal_range()` returns a range containing all elements with the given `key`

std::map, std::unordered_map

- `operator[]/at()` returns a reference to the element having the specified key in the container. A new element is generated in the set unless the key is found

Unordered containers:

- `bucket_count()` returns the number of buckets in the container
- `reserve()` sets the number of buckets to the number needed to accommodate at least count elements without exceeding maximum load factor and rehashes the container

std::set example

```cpp
#include <set>

std::set<int>          set1  { 5, 2, 3, 2, 7 };
std::set<int>          set2 = { 2, 3, 2 };
std::set<std::string> set3 = { "abc", "efg" };
std::set<int>          set4;          // empty set

set2.erase(2);                        // [ 3 ]
set3.insert("hij");                   // [ "abc", "efg", "hij" ]
for (const auto& it : set1)
    cout << it << " ";                // 2, 3, 5, 7  (sorted)

auto search = set1.find(2);           // iterator
cout << search != set1.end();         // true
auto it     = set1.lower_bound(4);
cout << *it;                          // 5
set1.count(2);                        // 1, note: it can only be 0 or 1
auto it_pair = set1.equal_range(2); // iterator between [2, 3)
```

std::map example

```cpp
#include <map>

std::map<std::string, int> map1 { {"bb", 5}, {"aa", 3} };
std::map<double, int> map2;         // empty map

cout << map1["aa"];                 // prints 3
map1["dd"] = 3;                     // insert <"dd", 3>
map1["dd"] = 7;                     // change <"dd", 7>
cout << map1["cc"];                 // insert <"cc", 0>
for (const auto& it : map1)
    cout << it.second << " ";       // 3, 5, 0, 7

map1.insert( {"jj", 1} );           // insert pair
auto search = map1.find("jj");      // iterator
cout << (search != map1.end());     // true
auto it = map1.lower_bound("bb");
cout << (*it).second;               // 5
```

std::multiset example

```cpp
#include <set> // std::multiset

std::multiset<int>    mset1 {1, 2, 5, 2, 2}; // 1, 2, 2, 2, 5
std::multiset<double> mset2;     // empty set

mset1.insert(5);
for (const auto& it : mset1)
    cout << it << " ";           // 1, 2, 2, 2, 5, 5
cout << mset1.count(2);          // 3

auto it = mset1.find(5);         // iterator
cout << *it;                     // 5

it      = mset1.lower_bound(4);
cout << *it;                     // 5
```

Container Adaptors

Overview

Container adaptors are interfaces for reducing the number of functionalities normally available in a container

The underlying container of a container adaptors can be optionally specified in the declaration

The *STL Container Adaptors* are:

- `std::stack` LIFO data structure
  default underlying container: **`std::deque`**
- `std::queue` FIFO data structure
  default underlying container: **`std::deque`**
- `std::priority_queue` (max) priority queue
  default underlying container: **`std::vector`**

Container Adaptors Methods

`std::stack` interface for a FILO (first-in, last-out) data structure

- `top()` accesses the top element
- `push()` inserts element at the top
- `pop()` removes the top element

`std::queue` interface for a FIFO (first-in, first-out) data structure

- `front()` access the first element
- `back()` access the last element
- `push()` inserts element at the end
- `pop()` removes the first element

`std::priority_queue` interface for a priority queue data structure (lookup to the largest element by default)

- `top()` accesses the top element
- `push()` inserts element at the end
- `pop()` removes the first element

Container Adaptor Examples

```cpp
#include <stack>          // <--
#include <queue>          // <-- also include priority_queue

std::stack<int> stack1;
stack1.push(1); stack1.push(4);   // [1, 4]
stack1.top();     // 4
stack1.pop();     // [1]

std::queue<int> queue1;
queue1.push(1); queue1.push(4);   // [1, 4]
queue1.front();   // 1
queue1.pop();     // [4]

std::priority_queue<int> pqueue1;
pqueue1.push(1); pqueue1.push(5); pqueue1.push(4);   // [5, 4, 1]
pqueue1.top();    // 5
pqueue1.pop();    // [4, 1]
```

Implement a Custom Iterator

Goal: implement a simple iterator to iterate over a `List` of elements:

```
#include <iostream>
#include <algorithm>
// !! List implementation here

int main() {
    List list;
    list.push_back(2);
    list.push_back(4);
    list.push_back(7);
    std::cout << *std::find(list.begin(), list.end(), 4); // print 4

    for (const auto& it : list)  // range-based loop
        std::cout << it << " ";  // 2, 4, 7
}
```

Range-based loops require: `begin()`, `end()`, pre-increment `++it`, not equal comparison `it != end()`, dereferencing `*it`

```cpp
using value_t = int;

struct List {
    struct Node {             // Internal Node Structure
        value_t _value;       // Node value
        Node*   _next;        // Pointer to next node
    };
    Node* _head { nullptr }; // head of the list
    Node* _tail { nullptr }; // tail of the list

    void push_back(const value_t& value); // insert a value at the end

    // !! here we have to define the List iterator "It"

    It begin() { return It{_head};   }    // begin of the list
    It end()   { return It{nullptr}; }    // end of the list
};
```

```
void List::push_back(const value_t& value) {
    auto new_node = new Node{value, nullptr};
    if (_head == nullptr) { // empty list
         _head = new_node; // head is updated
         _tail = _head;
        return;
    }
    assert(_tail != nullptr);
    _tail->_next = new_node; // add new node at the end
    _tail        = new_node; // tail is updated
}
```

```
struct It {
    Node* _ptr;            // internal pointer

    It(Node* ptr);         // Constructor

    value_t& operator*(); // Deferencing

    // Not equal -> stop traversing
    friend bool operator!=(const It& itA, const It& itB);

    It& operator++();      // Pre-increment

    It operator++(int);    // Post-increment

// !! Type traits here
};
```

```
List::It::It(Node* ptr) :_ptr(ptr) {}

value_t& Lis::It::operator*() { return _ptr->_value; }

bool operator!=(const It& itA, const It& itB) {
    return itA._ptr != itB._ptr;
}

List::It& List::It::operator++() {
    _ptr = _ptr->_next;
    return *this;
}
List::It List::It::operator++(int) {
    auto tmp = *this;
    ++(*this);
    return tmp;
}
```

The *type traits* of an iterator describe its properties, e.g. the type of the value held, and they are widely used in the `std` algorithms

`std::iterator` class template defines the type traits for an iterator. It has been deprecated in C++17, so users need to provide the type traits explicitly

```cpp
#include <iterator>

// !! Type traits
using iterator_category = std::forward_iterator_tag;
using difference_type   = std::ptrdiff_t;
using value_type        = value_t;
using pointer           = value_t*;
using reference         = value_t&;
```

internalpointers.com/post/writing-custom-iterators-modern-cpp
Preparation for std::iterator Being Deprecated

Iterator Notes

Common Errors

Modify a container with a “active” iterators

```cpp
#include <vector>

std::vector<int> vec{1, 2, 3, 4, 5};
for (auto x : vec)
    vec.push_back(x);   // iterator invalidation!!
```

Iterator Utility Methods

- `std::advance(InputIt& it, Distance n)`

  Increments a given iterator it by n elements
  - `InputIt` must support `input` iterator requirements
  - Modifies the iterator
  - Returns `void`
  - More general than adding a value `it + 4`
  - No performance loss if `it` satisfies `random access` iterator requirements

- `std::next(ForwardIt it, Distance n)` C++11

  Returns the n-th successor of the iterator
  - `ForwardIt` must support `forward` iterator requirements
  - Does not modify the iterator
  - More general than adding a value `it + 4`
  - The compiler should optimize the computation if `it` satisfies `random access` iterator requirements
  - Supports negative values if `it` satisfies `bidirectional` iterator requirements

- `std::prev(BidirectionalIt it, Distance n)` C++11

  Returns the `n`-th predecessor of the iterator
  - `InputIt` must support `bidirectional` iterator requirements
  - Does not modify the iterator
  - More general than adding a value `it + 4`
  - The compiler should optimize the computation if `it` satisfies `random access` iterator requirements

- `std::distance(InputIt start, InputIt end)`

  Returns the number of elements from `start` to `last`
  - `InputIt` must support `input` iterator requirements
  - Does not modify the iterator
  - More general than adding iterator difference `it2 - it1`
  - The compiler should optimize the computation if `it` satisfies `random access` iterator requirements
  - C++11 Supports negative values if `it` satisfies `random` iterator requirements

Examples

```cpp
#include <iterator>
#include <iostream>
#include <vector>
#include <forward_list>
int main() {
    std::vector<int> vector { 1, 2, 3 }; // random access iterator

    auto it1 = std::next(vector.begin(), 2);
    auto it2 = std::prev(vector.end(), 2);
    std::cout << *it1;   // 3
    std::cout << *it2;   // 2
    std::cout << std::distance(it2, it1); // 1

    std::advance(it2, 1);
    std::cout << *it2;   // 3

    //-------------------------------------
    std::forward_list<int> list { 1, 2, 3 }; // forward iterator
//  std::prev(list.end(), 1);                // compile error
}
```

Container Access Methods

C++11 provides a generic interface for containers, plain arrays, and `std::initializer_list` to access to the corresponding iterator.
Standard method `.begin()`, `.end()` etc., are not supported by plain array and initializer list

- `std::begin` begin iterator
- `std::cbegin` begin const iterator
- `std::rbegin` begin reverse iterator
- `std::crbegin` begin const reverse iterator
- `std::end` end iterator
- `std::cend` end const iterator
- `std::rend` end reverse iterator
- `std::crend` end const reverse iterator

```cpp
#include <iterator>
#include <iostream>

int main() {
    int array[] = { 1, 2, 3 };

    for (auto it = std::crbegin(array); it != std::crend(array); it++)
        std::cout << *it << ", ";   // 3, 2, 1
}
```

`std::iterator_traits` allows retrieving iterator properties

- `difference_type` a type that can be used to identify distance between iterators
- `value_type` the type of the values that can be obtained by dereferencing the iterator. This type is void for output iterators
- `pointer` defines a pointer to the type iterated over `value_type`
- `reference` defines a reference to the type iterated over `value_type`
- `iterator_category` the category of the iterator. Must be one of iterator category tags

```cpp
#include <iterator>

template<typename T>
void f(const T& list) {
    using D = std::iterator_traits<T>::difference_type;   // D is std::ptrdiff_t
                                                          // (pointer difference)
                                                          // (signed size_t)
    using V = std::iterator_traits<T>::value_type;        // V is double
    using P = std::iterator_traits<T>::pointer;           // P is double*
    using R = std::iterator_traits<T>::reference;         // R is double&

    // C is BidirectionalIterator
    using C = std::iterator_traits<T>::iterator_category;
}

int main() {
    std::list<double> list;
    f(list);
}
```

Algorithms Library

C++ STL Algorithms library

The algorithm library provides functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements

- STL Algorithm library allow great flexibility which makes included functions suitable for solving real-world problem
- The user can adapt and customize the STL through the use of function objects
- Library functions work independently on containers and plain array
- Many of them support `constexpr` in C++20

```cpp
#include <algorithm>
#include <vector>
struct Unary {
    bool operator()(int value) {
        return value <= 6 && value >= 3;
    }
};
struct Descending {
    bool operator()(int a, int b) {
         return a > b;
    }
};

int main() {
    std::vector<int> vector { 7, 2, 9, 4 };
    // returns an iterator pointing to the first element in the range[3, 6]
    std::find_if(vector.begin(), vector.end(), Unary());
    // sort in descending order :  { 9, 7, 4, 2 };
    std::sort(vector.begin(), vector.end(), Descending());
}
```

```cpp
#include <algorithm> // it includes also std::multiplies
#include <vector>
#include <cstdlib>   // std::rand
#include <numeric>   // std::accumulate
struct Unary {
    bool operator()(int value) { return value > 100; }
};
int main() {
    std::vector<int> vector { 7, 2, 9, 4 };
    int product = std::accumulate(vector.begin(), vector.end(), // product = 504
                                  1, std::multiplies<int>());
    std::generate(vector.begin(), vector.end(), std::rand);
    // now vector has 4 random values

    // remove all values > 100 using Erase-remove idiom
    auto new_end = std::remove_if(vector.begin(), vector.end(), Unary());
    // elements are removed, but vector size is still unchanged
    vector.erase(new_end, vector.end()); // shrink vector to finish removal
}
```

STL Algorithms Library (Possible Implementations)

std::find

```
template<class InputIt, class T>
InputIt find(InputIt first, InputIt last, const T& value) {
    for (; first != last; ++first) {
        if (*first == value)
            return first;
    }
    return last;
}
```

std::generate

```
template<class ForwardIt, class Generator>
void generate(ForwardIt first, ForwardIt last, Generator g) {
    while (first != last)
        *first++ = g();
}
```

- `swap(v1, v2)` Swaps the values of two objects
- `min(x, y)` Finds the minimum value between `x` and `y`
- `max(x, y)` Finds the maximum value between `x` and `y`
- `min_element(begin, end)` (returns a pointer)
  Finds the minimum element in the range [`begin, end`)
- `max_element(begin, end)` (returns a pointer)
  Finds the maximum element in the range [`begin, end`)
- `minmax_element(begin, end)` C++11 (returns pointers <min,max>)
  Finds the minimum and the maximum element in the range [`begin, end`)

- `equal(begin1, end1, begin2)`
  Determines if two sets of elements are the same in
  [begin1, end1), [begin2, begin2 + end1 - begin1)

- `mismatch(begin1, end1, begin2)` (returns pointers <pos1,pos2>)
  Finds the first position where two ranges differ in
  [begin1, end1), [begin2, begin2 + end1 - begin1)

- `find(begin, end, value)` (returns a pointer)
  Finds the first element in the range [begin, end) equal to value

- `count(begin, end, value)`
  Counts the number of elements in the range [begin, end) equal to value

- `sort(begin, end)` (in-place)
  Sorts the elements in the range [`begin, end`) in ascending order
- `merge(begin1, end1, begin2, end2, output)`
  Merges two sorted ranges [`begin1, end1`), [`begin2, end2`), and store the results in [`output, output + end1 - start1`)
- `unique(begin, end)` (in-place)
  Removes consecutive duplicate elements in the range [`begin, end`)
- `binary search(begin, end, value)`
  Determines if an element `value` exists in the (sorted) range [`begin, end`)
- `accumulate(begin, end, value)`
  Sums up the range [`begin, end`) of elements with initial `value` (common case equal to zero)
- `partial_sum(begin, end)` (in-place)
  Computes the inclusive prefix-sum of the range [`begin, end`)

- `fill(begin, end, value)`
  Fills a range of elements [begin, end) with value
- `iota(begin, end, value)` C++11
  Fills the range [begin, end) with successive increments of the starting value
- `copy(begin1, end1, begin2)`
  Copies the range of elements [begin1, end1) to the new location
  [begin2, begin2 + end1 - begin1)
- `swap_ranges(begin1, end1, begin2)`
  Swaps two ranges of elements
  [begin1, end1), [begin2, begin2 + end1 - begin1)
- `remove(begin, end, value)` (in-place)
  Removes elements equal to value in the range [begin, end)
- `includes(begin1, end1, begin2, end2)`
  Checks if the (sorted) set [begin1, end1) is a subset of [begin2, end2)

- `set_difference(begin1, end1, begin2, end2, output)`
  Computes the difference between two (sorted) sets
- `set_intersection(begin1, end1, begin2, end2, output)`
  Computes the intersection of two (sorted) sets
- `set_symmetric_difference(begin1, end1, begin2, end2, output)`
  Computes the symmetric difference between two (sorted) sets
- `set_union(begin1, end1, begin2, end2, output)`
  Computes the union of two (sorted) sets
- `make_heap(begin, end)` Creates a max heap out of the range of elements
- `push_heap(begin, end)` Adds an element to a max heap
- `pop_heap(begin, end)` Remove an element (top) to a max heap

Algorithm Library - Other Examples

```cpp
#include <algorithm>

int a        = std::max(2, 5); // a = 5
int array1[] = {7, 6, -1, 6, 3};
int array2[] = {8, 2, 0, 3, 7};

int  b = *std::max_element(array1, array1 + 5);   // b = 7
auto c = std::minmax_element(array1, array1 + 5);
//*c.first = -1, *c.second = 7
bool d = std::equal(array1, array1 + 5, array2); // d = false

std::sort(array1, array1 + 5);                  // [-1, 3, 6, 6, 7]
std::unique(array1, array1 + 5);                // [-1, 3, 6, 7]
int e = std::accumulate(array1, array1 + 4, 0); // 15
std::partial_sum(array1, array1 + 4, array1);   // [-1, 2, 8, 15]
std::iota(array1, array1 + 5, 2);               // [2, 3, 4, 5, 6]
std::make_heap(array2, array2 + 5);             // [8, 7, 0, 3, 2]
```

C++20 Ranges

C++20 Ranges

Ranges *are an abstraction that allows to operate on elements of data structures uniformly*. They are an extension of the standard *iterators*

A **range** is an object that provides `begin()` and `end()` methods (an *iterator* + a *sentinel*)

`begin()` returns an *iterator*, which can be incremented until it reaches `end()`

```
template<typename T>
concept range = requires(T& t) {
    ranges::begin(t);
    ranges::end(t);
};
```

- An Overview of Standard Ranges
- Range, Algorithms, Views, and Actions - A Comprehensive Guide
- Eric Nielbler - Range v3
- Range by Example

Key Concepts

Range View is a *range* defined on top of another *range*

Range Adaptors are utilities to transform a *range* into a *view*

Range Factory is a *view* that contains no elements

Range Algorithms are library-provided functions that directly operate on ranges (corresponding to `std` iterator algorithm)

Range Action is an object that modifies the underlying data of a range

A **range view** is a *range* defined on top of another *range* that transforms the underlying way to access internal data

- *Views* do not own any data
- *copy*, *move*, *assignment* operations perform in constant time
- *Views* are *composable*
- *Views* are *lazy evaluated*

Syntax:

```
range/view | view
```

```cpp
#include <iostream>
#include <ranges>
#include <vector>

std::vector<int> v{1, 2, 3, 4};

for (int x : v | std::views::reverse)
    std::cout << x << " ";  // print: "4, 3, 2, 1"

auto rv2 = v | std::views::reverse;  // cheap, it does not copy "v"

auto rv3 = v | std::views::drop(2) | // drop the first two elements
               std::views::reverse;
for (int x : rv3) // lazy evaluated
    std::cout << x << " ";  // print: "4, 3"
```

Range Adaptors are utilities to transform a *range* into a *view* with custom behaviors

- *Range adaptors* produce lazily evaluated *views*
- *Range adaptors* can be chained or composed (pipeline)

Syntax:

```
adaptor(range/view, args...)
adaptor(args...)(range/view)
range/view | adaptor(args...) // preferred syntax
```

```cpp
#include <iostream>
#include <ranges>
#include <vector>

std::vector<int> v{1, 2, 3, 4};

for (int x : v | std::ranges::reverse_view(v))     // @\textbf{adaptor}@
    std::cout << x << " ";  // print: "4, 3, 2, 1"

auto rv2 = std::ranges::reverse_view(v);  // cheap, it does not copy "v"

auto rv3 = std::ranges::reverse_view(
           std::ranges::drop_view(2, v)); // drop the first two elements
for (int x : rv3) // lazy evaluated
    std::cout << x << " ";  // print: "4, 3"
```

Range Factory

Range Factory produces a *view* that contains no elements

```
#include <iostream>
#include <ranges>

for (int x : std::ranges::iota_view{1, 4}) // factory (adaptor)
    std::cout << x << " ";                 // print: "1, 2, 3, 4"

for (int x : std::view::repeat('a', 4))    // factory (view)
    std::cout << x << " ";                 // print: "a, a, a, a"
```

Range Algorithms

The **range algorithms** are almost identical to the corresponding *iterator-pair* algorithms in the `std` namespace, except that they have *concept*-enforced constraints and accept *range* arguments

- *Range algorithms* are immediately evaluated
- *Range algorithms* can work directly on containers (`begin()` , `end()` are no more explicitly needed) and *view*

```cpp
#include <algorithm>
#include <vector>

std::vector<int> vec{3, 2, 1};
std::ranges::sort(vec); // 1, 2, 3
```

Algorithm Operators and Projections

```cpp
#include <algorithm>
#include <vector>

struct Data {
    char value1;
    int  value2;
};

std::vector<int> vec{4, 2, 5};
auto cmp = [](auto a, auto b) { return a > b; }; // Unary boolean predicate
std::ranges::sort(vec, cmp); // 5, 4, 2

std::vector<Data> vec2{{'a', 4}, {'b', 2}, {'c', 5}};
std::ranges::sort(vec2, {}, &Data::value2); // Projection: 2, 4, 5
                                            // {'b', 2}, {'a', 4}, {'c', 5}
```

Algorithms and Views

```cpp
// sum of the squares of the first 'count' numbers
auto sum_of_squares(int count) {
    auto squares = std::views::iota(1, count) |
                   std::views::transform([](int x) { return x * x; });
    return std::accumulate(squares, 0);
}
```

The **range actions** mimic *std algorithms* and *range algorithms* adding the **composability** property

- *Range actions* are *eager* evaluated
- *Range algorithms* work directly on *ranges*
- Not included in the `std` library

```cpp
#include <algorithm>
#include <vector>

std::vector<int> vec{3, 5, 6, 3, 5}
// in-place
vec = vec | actions::sort    // 3, 3, 5, 5, 6
          | actions::unique; // 3, 5, 6

vec |= actions::sort    // 3, 3, 5, 5, 6
    |  actions::unique; // 3, 5, 6
// out-of-place
auto vec2 = std::move(vec) | actions::sort    // 3, 3, 5, 5, 6
                           | actions::unique; // 3, 5, 6
```

Modern C++ Programming

18. Advanced Topics I

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

Move Semantic

Move semantics refers in transferring ownership of resources from one object to another

Differently from *copy semantic*, *move semantic* does not duplicate the original resource

In C++ every expression is either an **rvalue** or an **lvalue**

- a **lvalue** (left) represents an expression that occupies some identifiable location in memory
- a **rvalue** (right) is an expression that does not represent an object occupying some identifiable location in memory

```
int x = 5;       // "x" is an lvalue, "5" is an rvalue
int y = 10;      // "y" is an lvalue

int z = (x * y); // "z" is an lvalue, (x * y) is an rvalue
```

C++11 introduces a new kind of *reference* called **rvalue reference** `X&&`

- An **rvalue reference** only binds to an **rvalue**, that is a temporary
- An **lvalue reference** only binds to an **lvalue**
- A **const lvalue reference** binds to both **lvalue** and **rvalue**

```
int         x  = 5;         // "x" is an lvalue
int&        r1 = x;         // "r1" is an lvalue reference
// int&     r2 = 5;         // compile error, "5" is an rvalue
const int& cr = (x * y); // "cr" is an const lvalue reference

int&&       rv = (x * y); // "rv" is an rvalue reference, "(x * y)" is an rvalue
// int&&    rv1 = x;        // compile error, "x" is NOT an rvalue
```

```
struct A {};

void f(A& a) {}       // lvalue reference

void g(const A& a) {} // const lvalue reference

void h(A&& a) {}      // rvalue reference

A a;
f(a);      // ok, f() can modify "a"
g(a);      // ok, f() cannot modify "a"
// h(a);   // compile error f() does not accept lvalues

// f(A{}); // compile error f() does not accept rvalues
g(A{});    // ok, f() cannot modify the object A{}
h(A{});    // ok, f() can modify the object A{}
```

```cpp
#include <algorithm>
class Array { // Array Wrapper
public:
    Array() = default;

    Array(int size) : _size{size}, _array{new int[size]} {}

    Array(const Array& obj) : _size{obj._size}, _array{new int[obj._size]}  {
        // EXPENSIVE COPY (deep copy)
        std::copy(obj._array, obj._array + _size, _array);
    }

    ~Array() { delete[] _array; }
private:
    int  _size;
    int* _array;
};
```

```cpp
#include <vector>

int main() {
    std::vector<Array> vector;
    vector.push_back( Array{1000} ); // call push_back(const Array&)
}                                    // expensive copy
```

Before C++11: `Array{1000}` is created, passed by `const-reference`, copied, and then destroyed

Note: `Array{1000}` is no more used outside `push_back`

After C++11: `Array{1000}` is created, and moved to `vector` (fast!)

Class prototype with support for *move semantic*:

```
class X {
public:
    X();                          // default constructor

    X(const X& obj);              // copy constructor

    X(X&& obj);                   // move constructor

    X& operator=(const X& obj); // copy assign operator

    X& operator=(X&& obj);        // move assign operator

    ~X();                        // destructor
};
```

Move constructor semantic

```
X(X&& obj);
```

(1) *Shallow copy* of `obj` data members (in contrast to deep copy)

(2) *Release* any `obj` resources and reset all data members (pointer to `nullptr`, size to 0, etc.)

Move assignment semantic

```
X& operator=(X&& obj);
```

(1) *Release* any resources of `this`

(2) *Shallow copy* of `obj` data members (in contrast to deep copy)

(3) *Release* any `obj` resources and reset all data members (pointer to `nullptr`, size to 0, etc.)

(4) Return `*this`

Move constructor

```
Array(Array&& obj) {
    _size      = obj._size;  // (1) shallow copy
    _array     = obj._array; // (1) shallow copy
    obj._size  = 0;          // (2) release obj (no more valid)
    obj._array = nullptr;    // (2) release obj
}
```

Move assignment

```
Array& operator=(Array&& obj) {
    delete[] _array;         // (1) release this
    _size      = obj._size;  // (2) shallow copy
    _array     = obj._array; // (2) shallow copy
    obj._array = nullptr;    // (3) release obj
    obj._size  = 0;          // (3) release obj
    return *this;            // (4) return *this
}
```

C++11 provides the method `std::move` (`<utility>`) to indicate that an object may be "moved from"

It allows to efficient transfer resources from an object to another one

```cpp
#include <vector>

int main() {
    std::vector<Array> vector;
    vector.push_back( Array{1000} );     // call "push_back(Array&&)"

    Array arr{1000};
    vector.push_back( arr );             // call "push_back(const Array&)"

    vector.push_back( std::move(arr) ); // call "push_back(Array&&)"
                                         // efficient!!
//  "arr" is not more valid here
}
```

Move Semantic Notes

If an object requires the *copy constructor/assignment*, then it should also define the *move constructor/assignment*. The opposite could not be true

The *defaulted move constructor/assignment* `=default` recursively applies the move semantic to its *base class* and *data members*.
Important: *it does not release the resources*. It is very dangerous for classes with manual resource management

```
// Suppose: Array(Array&&) = default;
Array x{10};
Array y = std::move(x); // call the move constructor
// "x" calls ~Array() when it is out of scope, but now the internal pointer
// "_array" is NOT nullptr -> double free or corruption!!
```

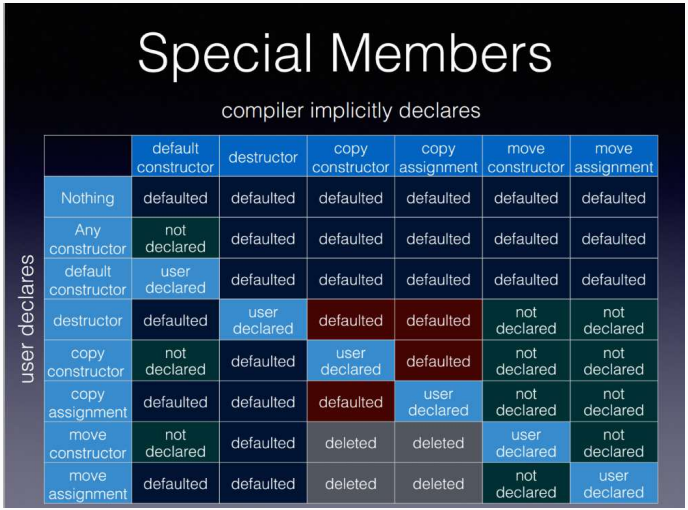
Move Semantic and Code Reuse

Some operations can be expressed as a function of the move semantic

```
A& operator=(const A& other) {
    *this = std::move(A{other}); // copy constructor + move assignment
    return *this;
}
```

```
void init(... /* any paramters */) {
    *this = std::move(A{...});  // user-declared constructor + move assignment
}
```

Class Declaration Semantic - Compiler Implicit



Everything You Ever Wanted To Know About Move Semantics

A Quick Note of Copy and Move Control in C++

Class Declaration Semantic

User-declared Entity	Meaning / Implications
non- `static` `const` `members`	*Copy/Move constructors* are not trivial (not provided by the compiler). *Copy/move assignment* is not supported
`reference members`	*Copy/Move constructors/assignment* are not trivial (not provided by the compiler)
`destructor`	The resource management is not trivial. *Copy constructor/assignment* is very likely to be implemented
`copy constructor/assignment`	Resource management is not trivial. *Move constructors/assignment* need to be implemented by the user
`move constructor/assignment`	There is an efficient way to move the object. *Copy constructor/assignment* cannot fall back safely to *copy constructors/assignment*, so they are deleted

Universal Reference and Perfect Forwarding

The `&&` syntax has two different meanings depending on the context it is used

- **rvalue reference**
- **Universal reference**: Either **rvalue reference** or **lvalue reference**

Universal references (also called *forwarding references*) are **rvalues** that appear in a type-deducing context. `T&&`, `auto&&` accept any expression regardless it is an **lvalue** or **rvalue** and preserve the `const` property

```
void f1(int&& t) {} // rvalue reference

template<typename T>
void f2(T&& t) {}   // universal reference

int&&  v1 = ...;    // rvalue reference
auto&& v2 = ...;    // universal reference
```

```
int         f_copy()                   { return x; }
int&        f_ref(int& x)              { return x; }
const int&  f_const_ref(const int& x)  { return x; }

auto          v1 = ...; // f_copy(), f_const_ref(), only lvalues
auto&         v2 = ...; // f_ref(), only lvalue ref
const auto&   v3 = ...; // f_copy(), f_ref(), f_const_ref()
                        // only const lvalue ref (decay), cannot be modified
const auto&&  v4 = ...; // f_copy(), only rvalues, cannot be modified

auto&&        v5 = ...; // everything
```

```
struct A {};
void f1(A&& a) {} // rvalue only

template<typename T>
void f2(T&& t) {} // universal reference

A a;
f1(A{});  // ok
// f1(a); // compile error (only rvalue)
f2(A{});  // universal reference
f2(a);    // universal reference

A&&    a2 = A{}; // ok
// A&& a3 = a;   // compile error (only rvalue)
auto&& a4 = A{}; // universal reference
auto&& a5 = a;   // universal reference
```

Universal Reference - Misleading Cases

```
template<typename T>
void f(std::vector<T>&&) {} // rvalue reference

template<typename T>
void f(const T&&) {}        // rvalue reference (const)

const auto&& v = ...;       // const rvalue reference
```

Reference Collapsing Rules

Before C++11 (C++98, C++03), it was not allowed to take a reference to a reference (`A& &` causes a compile error)

C++11, by contrast, introduces the following **reference collapsing rules**:

```
template<typename T>
void f(T&) {} // compile error in C++98/03 (with gcc),
              // no errors in C++11 (and clang with C++98/03)
int a = 3;    //
f<int&>(a);   //
```

Type	Reference		Result
A&	&	→	A&
A&	&&	→	A&
A&&	&	→	A&
A&&	&&	→	A&&

Perfect Forwarding

Perfect forwarding allows preserving argument *value category* and `const`/`volatile` modifiers

`std::forward` (`<utility>`) forwards the argument to another function with the *value category* it had when passed to the calling function (*perfect forwarding*)

```
#include <utility> // std::forward
template<typename T> void f(T& t)  { cout << "lvalue"; }
template<typename T> void f(T&& t) { cout << "rvalue"; } // overloading

template<typename T> void g1(T&& obj) { f(obj); } // call only f(T&)
template<typename T> void g2(T&& obj) { f(std::forward<T>(obj)); }

struct A{};
f ( A{10} ); // print "rvalue"
g1( A{10} ); // print "lvalue"!!
g2( A{10} ); // print "rvalue"
```

Value Categories

Taxonomy (simplified)

Every expression is either an **rvalue** or an **lvalue**

- An **lvalue** (*left* value of an assignment for historical reason or *locator* value) represents an expression that occupies an *identity*, namely a memory location (it has an address)

- An **rvalue** is movable; an **lvalue** is not

glvalue (*generalized lvalue*) is an expression that has an identity

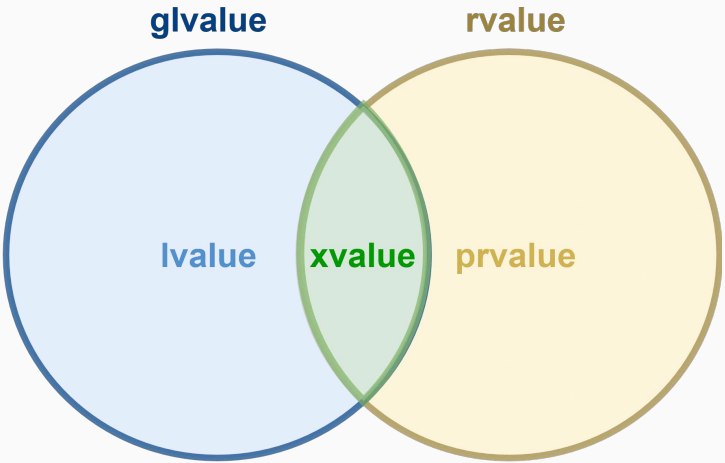
lvalue is a **glvalue** but it is not movable (it is not an **xvalue**). An *named rvalue reference* is a **lvalue**

xvalue (*eXpiring*) has an identity and it is movable. It is a **glvalue** that denotes an object whose resources can be reused. An *unnamed rvalue reference* is a **xvalue**

prvalue (*pure rvalue*) doesn't have identity, but is movable. It is an expression whose evaluation initializes an object or computes the value of an operand of an operator

rvalue is movable. It is a **prvalue** or an **xvalue**

en.cppreference.com/w/cpp/language/value_category


glvalue
rvalue
lvalue
xvalue
prvalue

Examples

```
struct A {
    int x;
};

void f(A&&) {}
A&&  g();
//-----------------------------------------------------------------
int a = 4;        // "a" is an lvalue, "4" is a prvalue
f(A{4});          // "A{4}" is a prvalue

A&& b = A{3};     // "A&& b" is a named rvalue reference → lvalue

A c{4};
f(std::move(c)); // "std::move(c)"  is a xvalue
f(A{}.x);         // "A{}.x" is a xvalue
g();              // "A&&" is a xvalue
```

`&`, `&&` Ref-qualifiers and `volatile` Overloading

C++11 allows overloading member functions depending on the **lvalue**/**rvalue** property of their object. This is also known as **ref-qualifiers overloading** and can be useful for optimization purposes, namely, moving a variable instead of copying it

```
struct A {
//  void f()    {}  // already covered by "f() &"
    void f() &  {}
    void f() && {}
};

A a1;
a1.f();             // call "f() &"

A{}.f();            // call "f() &&"
std::move(a1).f(); // call "f() &&"
```

Ref-qualifiers overloading can be also combined with `const` methods

```
struct A {
//  void f() const    {} // already covered by "f() const &"
    void f() const &  {}
    void f() const && {}
};

const A a1;
a1.f();             // call "f() const &"

std::move(a1).f(); // call "f() const &&"
```

A simple example where *ref-qualifiers overloading* is useful

```
struct ArrayWrapper {
    ArrayWrapper(/*params*/) { /* something expensive */ }

    ArrayWrapper copy() const &  { /* expensive copy with std::copy() */ }
    ArrayWrapper copy() const && { /* just move the pointer as the original
                                      object is no more used */  }
};
```

`volatile` Overloading

```
struct A {
    void f()                  {}
    void f() volatile         {} // e.g. propagate volatile to data members
    void f() const volatile   {}
//  void f() volatile &       {} // combining ref-qualifier and volatile
//  void f() const volatile & {} // overloading is also fine
//  void f() volatile &&      {}
//  void f() const volatile && {}
};

volatile A a1;
a1.f();   // call "f() volatile"

const volatile A a2;
a2.f();   // call "f() const volatile"
```

Copy Elision and RVO

Copy Elision and RVO

Copy elision is a compiler optimization technique that eliminates unnecessary copying/moving of objects (it is defined in the C++ standard)

A compiler avoids omitting copy/move operations with the following optimizations:

- **RVO (Return Value Optimization)** means the compiler is allowed to avoid creating *temporary* objects for return values
- **NRVO (Named Return Value Optimization)** means the compiler is allowed to return an object (with automatic storage duration) without invokes copy/move constructors

RVO Example

Returning an object from a function is *very expensive* without RVO/NVRO:

```
struct Obj {
    Obj() = default;

    Obj(const Obj&) { // non-trivial
        cout << "copy constructor\n";
    }
};

Obj f() { return Obj{}; } // first copy

auto x1 = f();            // second copy (create "x")
```

If provided, the compiler uses the *move constructor* instead of *copy constructor*

RVO - Where it works

RVO Copy elision is always guaranteed if the operand is a **prvalue** of the same class type and the *copy constructor* is trivial and non-deleted

```
struct Trivial {
    Trivial()               = default;
    Trivial(const Trivial&) = default;
};

// sigle instance
Trivial f1() {
   return Trivial{};  // Guarantee RVO
}
// distinct instances and run-time selection
Trivial f2(bool b) {
    return b ?  Trivial{} : Trivial{}; // Guarantee RVO
}
```

Guaranteed Copy Elision (C++17)

In C++17, *RVO Copy elision* is always guaranteed if the operand is a **prvalue** of the same class type, even if the *copy constructor* is not trivial or deleted

```
struct S1 {
  S1()          = default;
  S1(const S1&) = delete; // deleted
};
struct S2 {
  S2()          = default;
  S2(const S2&) {}        // non-trivial
};

S1 f() { return S1{}; }
S2 g() { return S2{}; }

auto x1 = f(); // compile error in C++14
auto x2 = g(); // RVO only in C++17
```

NRVO is not always guarantee even in C++17

```
Obj f1() {
    Obj a;
    return a; // most compilers apply NRVO
}

Obj f2(bool v) {
    Obj a;
    if (v)
       return a;  // copy/move constructor
    return Obj{}; // RVO
}
```

```
Obj f3(bool v) {
    Obj a, b;
    return v ? a : b;    // copy/move constructor
}

Obj f4() {
    Obj a;
    return std::move(a); // force move constructor
}

Obj f5() {
    static Obj a;
    return a;            // only copy constructor is possible
}
```

```
Obj f6(Obj& a) {
    return a; // copy constructor (a reference cannot be elided)
}

Obj f7(const Obj& a) {
    return a; // copy constructor (a reference cannot be elided)
}

Obj f8(const Obj a) {
    return a; // copy constructor (a const object cannot be elided)
}

Obj f9(Obj&& a) {
     return a; // copy constructor (the object is instantiated in the function)
}
```

Type Deduction

Type Deduction

When you call a template function, you may omit any template argument that the compiler can determine or deduce (inferred) by the usage and context of that template function call [IBM]

- The compiler tries to deduce a template argument by comparing the type of the corresponding template parameter with the type of the argument used in the function call
- Similar to function default parameters, (any) template parameters can be deduced only if they are at end of the parameter list

Full Story: IBM Knowledge Center

Example

```
template<typename T>
int add1(T a, T b) {  return a + b; }

template<typename T, typename R>
int add2(T a, R b) { return a + b;  }

template<typename T, int B>
int add3(T a) { return a + B; }

template<int B, typename T>
int add4(T a) { return a + B; }

add1(1, 2);       // ok
// add1(1, 2u);      // the compiler expects the same type
add2(1, 2u);      // ok (add2 is more generic)
add3<int, 2>(1); // "int" cannot be deduced
add4<2>(1);       // ok
```

Type Deduction - Pass by-Reference

Type deduction with references

```
template<typename T>
void f(T& a) {}

template<typename T>
void g(const T& a) {}

int         x = 3;
int&        y = x;
const int& z = x;
f(x);   // T: int
f(y);   // T: int
f(z);   // T: const int // <-- !! it works...but it does not
g(x);   // T: int       //     for "f(int& a)"!!
g(y);   // T: int       //     (only non-const references)
g(z);   // T: int       // <-- note the difference
```

Type deduction with pointers

```
template<typename T>
void f(T* a) {}

template<typename T>
void g(const T* a) {}

int*       x = nullptr;
const int* y = nullptr;
auto       z = nullptr;
f(x);    // T: int
f(y);    // T: const int
// f(z);    // compile error!! z: "nullptr_t != T*"
g(x);    // T: int
g(y);    // T: int   <-- note the difference
```

```cpp
template<typename T>
void f(const T* a) {} // pointer to const-values

template<typename T>
void g(T* const a) {} // const pointer

int*             x = nullptr;
const int*       y = nullptr;
int* const       z = nullptr;
const int* const w = nullptr;
f(x);    // T: int
f(y);    // T: int
f(z);    // T: int
g(x);    // T: int
g(y);    // T: const int
g(z);    // T: int
g(w);    // T: const int
```

Type deduction with values

```
template<typename T>
void f(T a) {}

template<typename T>
void g(const T a) {}

int        x = 2;
const int  y = 3;
const int& z = y;
f(x);   // T: int
f(y);   // T: int!!  (drop const)
f(z);   // T: int!!  (drop const&)
g(x);   // T: int
g(y);   // T: int
g(z);   // T: int!!  (drop reference)
```

```cpp
template<typename T>
void f(T a) {}


int*       x = nullptr;
const int* y = nullptr;
int* const z = x;
f(x);  // T = int*
f(y);  // T = const int*
f(z);  // T = int* !! (const drop)
```

Type Deduction - Array

Type deduction with arrays

```
template<typename T, int N>
void f(T (&array)[N]) {}   // type and size deduced

template<typename T>
void g(T array) {}

int       x[3] = {};
const int y[3] = {};
f(x);   // T: int, N: 3
f(y);   // T: const int, N: 3
g(x);   // T: int*
g(y);   // T: const int*
```

```cpp
template<typename T>
void add(T a, T b) {}

template<typename T, typename R>
void add(T a, R b) {}

template<typename T>
void add(T a, char b) {}

add(2, 3.0f);         // call add(T, R)
// add(2, 3);         // error!! ambiguous match
add<int>(2, 3);       // call add(T, T)
add<int, int>(2, 3); // call add(T, R)
add(2, 'b');          // call add(T, char) -> nearest match
```

```cpp
template<typename T, int N>
void f(T (&array)[N]) {}

template<typename T>
void f(T* array) {}

// template<typename T>
// void f(T array) {} // ambiguous

int x[3];
f(x); // call f(T*) not f(T(&)[3]) !!
```

`auto` Deduction

- `auto x =` copy by-value/by-const value
- `auto& x =` copy by-reference/by-const-refernce
- `auto* x =` copy by-pointer/by-const-pointer
- `auto&& x =` copy by-universal reference
- `decltype(auto) x =` automatic type deduction

```
int            f1(int& x) { return x; }
int&           f2(int& x) { return x; }
auto           f3(int& x) { return x; }
decltype(auto) f4(int& x) { return x; }

int  v  = 3;
int  x1 = f1(v);
int& x2 = f2(v);
// int& x3 = f3(v); // compile error 'x' is copied by-value
int& x4 = f4(v);
```

The problem: implement a function to remove the first element of a container

```
template<typename T>
void pop_v1(T& x) {
    std::remove(x.begin(), x.end(), x.front()); // undefined behavior!!
}
```

This is *undefined behavior* because

- `x.front()` returns a reference
- `std::remove` takes the element to remove by-const-reference
- `std::remove` modifies the container, invalidating iterators and references. The reference must not be an element of the range `[first, last)`

Sub-optimal solutions:

```
template<typename T>
void pop_v2(T& x) {
    auto tmp = x.front();                  // lvalue copy
    std::remove(x.begin(), x.end(), tmp); // ok
}
```

```
template<typename T>
void pop_v3(T& x) {
    using R = std::decay_t<decltype(x.front())>; // verbose/non-trivial solution
    std::remove(x.begin(), x.end(), R(x));       // ok, create a temporary (rvalue)
}                                                // copy
// decltype(x.front()) -> retrieve the type of x.front()
// std::decay_t        -> get the 'decay' type as pass by-value,
//                        e.g. 'const int' to 'int'
```

C++23 introduces `auto(x)` *decay-copy* utility to express the `rvalue` copy in a clear way

```
template<typename T>
void pop_v4(T& x) {
    std::remove(x.begin(), x.end(), auto(x.front())); // ok, rvalue copy
}                                                     // equivalent to R(x)
```

`const` Correctness

`const` Correctness

`const` correctness refers to guarantee object/variable `const` consistency throughout its lifetime and ensuring safety from unintentional modifications

References:

- `Isocpp: const-correctness`
- `GotW: Const-Correctness`
- `Abseil: Meaningful ‘const’ in Function Declarations`
- `const is a contract`
- `Why const Doesn’t Make C Code Faster`
- `Constant Optimization?`

- `const` entities do not change their values at run-time. This does not imply that they are evaluated at compile-time
- `const T*` is different from `T* const`. The first case means *"the content does not change"*, while the later *"the value of the pointer does not change"*
- Pass *by-const-value* and *by-value* parameters imply the *same* function signature
- Return *by-const-value* and *by-value* have different meaning
- `const_cast` can *break* const-correctness

`const` and member functions:

- `const` member functions do not change the internal status of an object
- `mutable` fields can be modified by a `const` member function (they should not change the external view)

`const` and code optimization:

- `const` keyword purpose is for correctness (*type safety*), not for performance
- `const` may provide performance advantages in a few cases, e.g. non-trivial copy semantic

Function Declarations Example

```
void f(int);
void f(const int); // the declaration is exactly the same of
                   // "void f(int)"!!
void f(int*);
void f(const int*); // different declaration

void f(int&);
void f(const int&); // different declaration
```

```
int          f();
// const int f(); // compile error conflicting declaration
```

const Return Example

```
   const int const_value = 3;

   const int& f2() { return const_value; }
// int&      f1() { return const_value; } // WRONG
   int        f3() { return const_value; }   // ok
```

```
struct A {
    void f()       { cout << "non-const"; }
    void f() const { cout << "const";     }
};

const A getA() { return A{}; }

auto a = getA(); // "a" is a copy
a.f();           // print "non-const"

getA().f();      // print "const"
```

`struct` Example

```
struct A {          // struct A_const { // equal to "const A"
    int* ptr;       //     int* const ptr;
    int  value;     //     const int  value;
};                  // };

void f(A a) {
    a.value  = 3;
    a.ptr[0] = 3;
}
void g(const A a) {  // the same with g(const A&)
//  a.value  = 3;    // compile error
    a.ptr[0] = 3;    // "const" does not apply to "ptr" content!!
}

A a{new int[10]};
f(a);
g(a);
```

Member Functions Example

```
struct A {
   int value = 0;

   int&       f1() { return value; }
   const int& f2() { return value; }

// int&       f3() const { return value; } // WRONG
   const int& f4() const { return value; }

   int       f5() const { return value; } // ok
   const int f6() const { return value; }
};
```

Modern C++ Programming

19. Advanced Topics II

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

Undefined Behavior

Undefined Behavior Overview

Undefined behavior means that the semantic of certain operations is

- *Unspecified behavior*: outside the language/library specification, two or more choices
- *Illegal*: the compiler presumes that such operations never happen, e.g. integer overflow
- *Implementation-defined behavior*: depends on the compiler and/or platform (not portable)

Motivations behind undefined behavior:

- *Compiler optimizations*, e.g. signed overflow or NULL pointer dereferencing
- *Simplify compile checks*
- *Unfeasible/expensive* to check

- `What Every C Programmer Should Know About Undefined Behavior`, *Chris Lattner*
- `What are all the common undefined behaviors that a C++ programmer should know about?`
- `Enumerating Core Undefined Behavior`

- `const_cast` **applied to a** `const` **variables**

```
const int        var  = 3;
const_cast<int&>(var) = 4;
... // use var
```

- **Memory alignment**

```
char* ptr  = new char[512];
auto  ptr2 = reinterpret_cast<uint64_t*>(ptr + 1);
ptr2[3]; //  ptr2 is not aligned to 8 bytes (sizeof(uint64_t))
```

- **Memory initialization**

```
int  var;            // undefined value
auto var2 = new int; // undefined value
```

- **Memory access-related**: *Out-of-bound access:* the code could crash or not depending on the platform/compiler

- **Strict aliasing**

```
float x = 3;
auto  y = reinterpret_cast<unsigned&>(x);
// x, y break the strict aliasing rule
```

- **Lifetime issues**

```
int* f() {
    int tmp[10];
    return tmp;
}
int* ptr = f();
ptr[0];
```

- **One Definition Rule violation**
  - Different definitions of `inline` functions in distinct translation units

- **Missing `return` statement**

```
int f(float x) {
    int y = x * 2;
}
```

- **Dangling reference**

```
iint n = 1;
const int& r = std::max(n-1, n+1); // dagling
// GCC 13 experimental -Wdangling-reference (enabled by -Wall)
```

- **Illegal arithmetic and conversion operations**
  - Division by zero `0 / 0`, `fp_value / 0.0`
  - Floating-point to integer conversion

Platform Specific Behavior

- **Memory access-related**: `NULL` *pointer dereferencing*: the `0x0` address is valid in some platforms
- **Endianness**

```
union U {
    unsigned x;
    char     y;
};
```

- **Type definition**

```
long x = 1ul << 32u; // different behavior depending on the OS
```

- **Intrinsic functions**

Legal operations but the C++ standard does not document the result → different compilers/platforms can show different behavior

- Signed shift `-2 ≪ x` (before C++20), large-than-type shift `3u ≪ 32`, etc.
- Floating-point narrowing conversion between floating-point types `double → float`
- Arithmetic operation ordering `f(i++, i++)`
- Function evaluation ordering

```
auto x = f() + g(); // C++ doesn't ensure that f() is evaluated before g()
```

- Signed overflow

```
for (int i = 0; i <= N; i++)
```

if `N` is `INT_MAX`, the last iteration is undefined behavior. The compiler can assume that the loop is finite and enable important optimizations, as opposite to `unsigned` (wrap around)

- Trivial infinite loops, until C++26

```
int main() {
  while (true)   // -> std::this_thread::yield(); in C++26
    ;
}
void unreachable() { cout << "Hello world!" << endl; }
```

the code print `Hello world!` with some `clang` versions

Detecting Undefined Behavior

There are several ways to detect or prevent undefined behavior at compile-time and at run-time:

- Modify the compiler behavior, see `Debugging and Testing: Hardening Techniques`
- Using undefined behavior sanitizer, see `Debugging and Testing: Sanitizer`
- Static analysis tools
- `constexpr` expressions doesn't allow undefined behavior

```
constexpr int   x1 = 2147483647 + 1;   // compile error
constexpr int   x2 = (1 << 32);        // compile error
constexpr int   x3 = (1 << -1);        // compile error
constexpr int   x4 =  3 / 0;           // compile error
constexpr int   x5 = *((int*) nullptr) // compile error
constexpr int   x6 = 6
constexpr float x7 = reinterpret_cast<float&>(x6); // compile error
```

Error Handing

Recoverable Error Handing

Recoverable *Conditions that are not under the control of the program*. They indicate *"exceptional"* run-time conditions. e.g. file not found, bad allocation, wrong user input, etc.

A *recoverable* should be considered *unrecoverable* if it is extremely rare and difficult to handle, e.g. bad allocation due to out-of-memory error

The common ways for handling recoverable errors are:

Exceptions	Robust but slower and requires more resources
Return code	Fast but difficult to handle in complex programs

Error Handing References

- Modern C++ best practices for exceptions and error handling
- Back to Basics: Exceptions - CppCon2020
- ISO C++ FAQ: Exceptions and Error Handling
- Zero-overhead deterministic exceptions: Throwing values, P0709
- C++ exceptions are becoming more and more problematic, P2544
- std::expected

Return Code

Historically, C programs handled errors with return codes, even for unrecoverable errors

```
enum Status { IllegalValue, Success };

Status f(int* ptr) { return (ptr == nullptr) ? IllegalValue : Success; }
```

Why such behavior? Debugging → need to understand what / where / why the program failed

A better approach in C++ involves `std::source_location()` C++20 and `std::stacktrace()` C++23

ABI related issues:

- Removing an enumerator value is an API breaking change
- Adding a new enumerator value associated to a return type is also problematic as it causes ABI breaking change

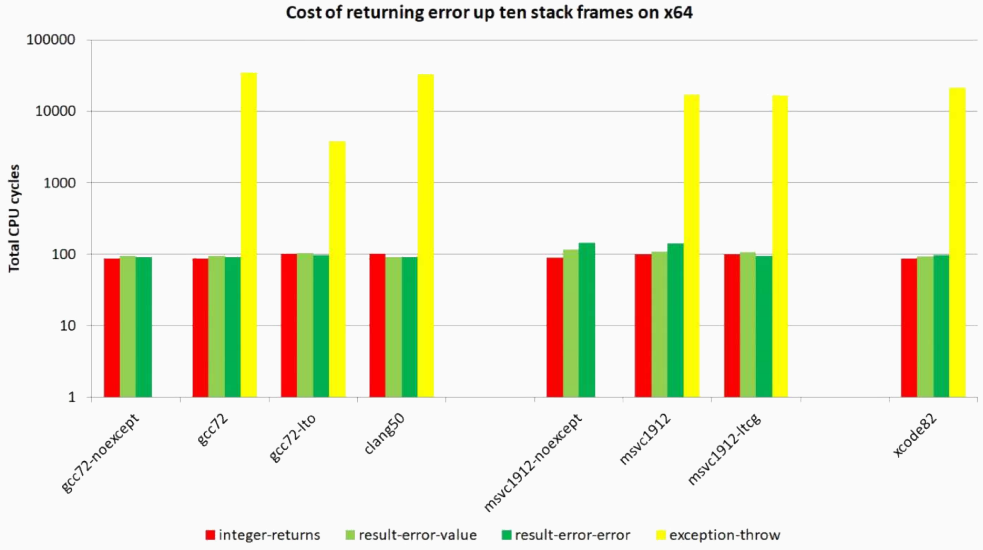
C++ Exceptions - Advantages

C++ Exceptions provide a well-defined mechanism to detect errors passing the information up the call stack

- **Exceptions cannot be ignored**. Unhandled exceptions stop program execution (call `std::terminate()`)
- **Intermediate functions are not forced to handle them**. They don't have to coordinate with other layers and, for this reason, they provide good composability
- Throwing an exception **acts like a return statement** destroying all objects in the current scope
- An exception enables a **clean separation** between the code that detects the error and the code that handles the error
- Exceptions work well with object-oriented semantic (constructor)

- **Code readability**: Using exception can involve more code than the functionality itself
- **Code comprehension**: Exception control flow is invisible and it is not explicit in the function signature
- **Performance:** Extreme performance overhead in the failure case (violate the zero-overhead principle)
- **Dynamic behavior**: `throw` requires dynamic allocation and `catch` requires `RTTI`. It is not suited for real-time, safety-critical, or embedded systems
- **Code bloat**: Exceptions could increase executable size by 5-15% (or more*)

*Binary size and exceptions


Cost of returning error up ten stack frames on x64
Total CPU cycles
100000
10000
1000
100
10
1
gcc72-noexcept
gcc72
gcc72-lto
clang50
msvc1912-noexcept
msvc1912
msvc1912-ltcg
xcode82
integer-returns
result-error-value
result-error-error
exception-throw

C++ Exception Basics

C++ provides three keywords for exception handling:

- `throw` Throws an exception
- `try` Code block containing potential throwing expressions
- `catch` Code block for handling the exception

```
void f() { throw 3; }

int main() {
    try {
        f();
    } catch (int x) {
        cout << x; // print "3"
    }
}
```

`std` Exceptions

`throw` can throw everything such as integers, pointers, objects, etc. The standard way consists in using the `std` library exceptions `<stdexcept>`

```cpp
#include <stdexcept>

void f(bool b) {
    if (b)
        throw std::runtime_error("runtime error");
    throw std::logic_error("logic error");
}
int main() {
    try {
        f(false);
    } catch (const std::runtime_error& e) {
        cout << e.what();
    } catch (const std::exception& e) {
        cout << e.what();  // print: "logic error"
    }
}
```

Exception Capture

NOTE: C++, differently from other programming languages, does not require explicit dynamic allocation with the keyword `new` for throwing an exception. The compiler implicitly generates the appropriate code to construct and clean up the exception object. Dynamically allocated objects require a `delete` call

The right way to capture an exception is by `const`-reference. Capturing by-value is also possible but, it involves useless copy for non-trivial exception objects

`catch(...)` can be used to capture any thrown exception

```
int main() {
    try {
        throw "runtime error"; // throw const char*
    } catch (...) {
        cout << "exception";   // print "exception"
    }
}
```

Exception Propagation

Exceptions are automatically propagated along the call stack. The user can also control how they are propagated

```
int main() {
    try {
        ...
    } catch (const std::runtime_error& e) {
        throw e; // propagate a copy of the exception
    } catch (const std::exception& e) {
        throw;   // propagate the exception
    }
}
```

Defining Custom Exceptions

```cpp
#include <exception> // to not confuse with <stdexcept>

struct MyException : public std::exception {
    const char* what() const noexcept override { // could be also "constexpr"
        return "C++ Exception";
    }
};

int main() {
    try {
        throw MyException();
    } catch (const std::exception& e) {
        cout << e.what(); // print "C++ Exception"
    }
}
```

`noexcept` Keyword

C++03 allows listing the exceptions that a function might directly or indirectly throw, e.g. `void f() throw(int, const char*) {`

C++11 deprecates `throw` and introduces the `noexcept` keyword

```
void f1();                  // may throw
void f2() noexcept;         // does not throw
void f3() noexcept(true);   // does not throw
void f4() noexcept(false); // may throw
template<bool X>
void f5() noexcept(X);      // may throw if X is false
```

If a `noexcept` function throw an exception, the runtime calls `std::terminate()`

`noexcept` should be used when throwing an exception is impossible or unacceptable. It is also useful when the function contains code outside user control, e.g. `std` functions/objects

Function-try-block

Exception handlers can be defined around the body of a function

```
void f() try {
    ... // do something
} catch (const std::runtime_error& e) {
    cout << e.what();
} catch (...) { // other exception
    ...
}
```

The `new` operator automatically throws an exception (`std::bad_alloc`) if it cannot allocate the memory

`delete` never throws an exception (unrecoverable error)

```
int main() {
   int* ptr = nullptr;
   try {
       ptr = new int[1000];
   }
   catch (const std::bad_alloc& e) {
       cout << "bad allocation: " << e.what();
   }
   delete[] ptr;
}
```

C++ also provides an overload of the `new` operator with non-throwing memory allocation

```
#include <new> // std::nothrow

int main() {
    int* ptr = new (std::nothrow) int[1000];
    if (ptr == nullptr)
        cout << "bad allocation";
}
```

Throwing exceptions in *constructors* is fine while it is not allowed in *destructors*

```
struct A {
    A()  { new int[10]; }
    ~A() { throw -2;    }
};
int main() {
    try {
       A a; // could throw "bad_alloc"
            // "a" is out-of-scope -> throw 2
    } catch (...) {
            // two exceptions at the same time
    }
}
```

Destructors should be marked `noexcept`

```
struct A {
    int* ptr1, *ptr2;

    A() {
        ptr1 = new int[10];
        ptr2 = new int[10]; // if bad_alloc here, ptr1 is lost
    }
};
```

```
struct A {
    std::unique_ptr<int> ptr1, ptr2;

    A() {
        ptr1 = std::make_unique<int[]>(10);
        ptr2 = std::make_unique<int[]>(10); // if bad_alloc here,
    }                                       // ptr1 is deallocated
};
```

Return Code and Exception Summary

	Exception	Return Code
Pros	▪ Cannot be ignored ▪ Work well with object-oriented semantic ▪ Information: Exceptions can be arbitrarily rich ▪ Clean code: Conceptually, clean separation between the code that detects errors and the code that handles the error, but... * ▪ Non-Intrusive wrt. API: Proper communication channel	▪ Visibility: prototype of the called function ▪ No performance overhead ▪ No code bloat ▪ Easy to debug
Cons	▪ Visibility: Not visible without further analysis of the code or documentation ▪ Clean code: *... handling exception can generate more code than the functionality itself ▪ Dynamic behavior: memory and RTTI ▪ Extreme performance overhead in the failure case ▪ Code bloat ▪ Non-trivial to debug	▪ Easy to ignore, `[[deprecated]]` can help ▪ Cannot be used with object-oriented semantic ▪ Information: Historically, a simple integer. Nowadays, richer error code ▪ Clean code: At least, an if statement after each function call ▪ Non-Intrusive wrt. API: Monopolization of the return channel

C++23 introduces `std::expected` to get the best properties of return codes and exceptions

The class template `expected<T, E>` contains either:

- A value of type `T`, the expected value type; or
- A value of type `E`, an error type used when an unexpected outcome occured

```
enum class Error { Invalid };

std::expected<int, Error> f(int v) {
    if (v > 0)
        return 3;
    return std::unexpected(Error::Invalid);
}
```

The user chooses how to handle the error depending on the context

```
auto ret = f(n);

// Return code handling
if (!ret)
    // error handling
int v = *ret + 3; // execute without checking

// Exception handling
ret.value(); // throw an exception if there is a problem

// Monadic operations
auto lambda = [](int x) { return (x > 3) ? 4 : std::unexpected(Error::Invalid); };
ret.and_then(lambda)                          // pass the value to another function
   .tranform([](int x) { return x + 4; };)    // transform the previous value
   .transform_error([](auto error_code){ /*error handling*/ };
```

- **Global state**, e.g. `errno`
  - Easily forget to check for failures
  - Error propagation using `if` statements and early `return` is manual
  - No compiler optimizations due to global state

- **Simple error code**, e.g. `int`, `enum`, etc.
  - Easily forget to check for failures (workaround `[[nodiscard]]`)
  - Error propagation using `if` statements and early `return` is manual
  - Potential error propagation through different contexts and losing initial error information
  - Constructor errors cannot be handled

- `std::error_code`, standardized error code
  - Easily forget to check for failures (workaround `[[nodiscard]]`)
  - Error propagation using `if` statements and early `return` is manual
  - Code bloating for adding new enumerators (see `Your own error code`)
  - Constructor errors cannot be handled

- **Supporting libraries**, e.g. `Boost Outcome`, `STX`, etc.
  - Require external dependencies
  - Constructor errors cannot be handled in a direct way
  - Extra logic for managing return values

Smart pointers

Smart Pointers

Smart pointer is a pointer-like type with some additional functionality, e.g. *automatic memory deallocation* (when the pointer is no longer in use, the memory it points to is deallocated), reference counting, etc.

C++11 provides three smart pointer types:

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`

Smart pointers prevent most situations of memory leaks by making the memory deallocation automatic

Smart Pointers Benefits

- If a smart pointer goes *out-of-scope*, the appropriate method to release resources is called automatically. The memory is not left dangling
- Smart pointers will automatically be set to `nullptr` if not initialized or when memory has been released
- `std::shared_ptr` provides automatic reference count
- If a special `delete` function needs to be called, it will be specified in the pointer type and declaration, and will automatically be called on delete

`std::unique_ptr` is used to manage any dynamically allocated object that is not shared by multiple objects

```
#include <iostream>
#include <memory>
struct A {
    A()  { std::cout << "Constructor\n"; } // called when A()
    ~A() { std::cout << "Destructor\n";  } // called when u_ptr1,
};                                         // u_ptr2 are out-of-scope
int main() {
    auto                raw_ptr = new A();
    std::unique_ptr<A> u_ptr1(new A());
    std::unique_ptr<A> u_ptr2(raw_ptr);
//  std::unique_ptr<A> u_ptr3(raw_ptr); // no compile error, but wrong!! (not unique)

//  u_ptr1 = raw_ptr;                   // compile error (not unique)
//  u_ptr1 = u_ptr2;                    // compile error (not unique)
    u_ptr1 = std::move(u_ptr2); // delete u_ptr1;
}                               // u_ptr1 = u_ptr2;
                                // u_ptr2 = nullptr
```

`std::unique_ptr` methods

- `get()` returns the underlying pointer
- `operator*` `operator->` dereferences pointer to the managed object
- `operator[]` provides indexed access to the stored array (if it supports random access iterator)
- `release()` returns a pointer to the managed object and releases the ownership
- `reset(ptr)` replaces the managed object with `ptr`

Utility method: `std::make_unique<T>()` creates a unique pointer to a class `T` that manages a new object

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::unique_ptr<A> u_ptr1(new A());
    u_ptr1->value;      // dereferencing
    (*u_ptr1).value;    // dereferencing

    auto u_ptr2 = std::make_unique<A>();  // create a new unique pointer

    u_ptr1.reset(new A());            // reset
    auto raw_ptr = u_ptr1.release();  // release
    delete[] raw_ptr;

    std::unique_ptr<A[]> u_ptr3(new A[10]);
    auto& obj = u_ptr3[3];            // access
}
```

Implement a custom deleter

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    auto DeleteLambda = [](A* x) {
        std::cout << "delete" << std::endl;
        delete x;
    };

    std::unique_ptr<A, decltype(DeleteLambda)>
        x(new A(), DeleteLambda);
} // print "delete"
```

`std::shared_ptr` is the pointer type to be used for memory that can be owned by multiple resources at one time

`std::shared_ptr` maintains a reference count of pointer objects. Data managed by `std::shared_ptr` is only freed when there are no remaining objects pointing to the data

```
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::shared_ptr<A> sh_ptr1(new A());
    std::shared_ptr<A> sh_ptr2(sh_ptr1);
    std::shared_ptr<A> sh_ptr3(new A());
    sh_ptr3 = nullptr; // allowed, the underlying pointer is deallocated
                       //   sh_ptr3 : zero references
    sh_ptr2 = sh_ptr1; // allowed. sh_ptr1, sh_ptr2: two references
    sh_ptr2 = std::move(sh_ptr1); // allowed // sh_ptr1: zero references
}                                            // sh_ptr2: one references
```

`std::shared_ptr` methods

- `get()` returns the underlying pointer
- `operator*` `operator->` dereferences pointer to the managed object
- `use_count()` returns the number of objects referring to the same managed object
- `reset(ptr)` replaces the managed object with `ptr`

Utility method: `std::make_shared()` creates a shared pointer that manages a new object

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::shared_ptr<A> sh_ptr1(new A());
    auto sh_ptr2 = std::make_shared<A>();  // std::make_shared
    std::cout << sh_ptr1.use_count();  // print 1

    sh_ptr1 = sh_ptr2;                 // copy
//  std::shared_ptr<A> sh_ptr2(sh_ptr1); // copy (constructor)
    std::cout << sh_ptr1.use_count();  // print 2
    std::cout << sh_ptr2.use_count();  // print 2

    auto raw_ptr = sh_ptr1.get();      // get
    sh_ptr1.reset(new A());            // reset
    (*sh_ptr1).value = 3;              // dereferencing
    sh_ptr1->value = 2;                // dereferencing
}
```

A **std::weak_ptr** is simply a std::shared_ptr that is allowed to dangle (pointer not deallocated)

```cpp
#include <memory>

std::shared_ptr<int> sh_ptr(new int);
std::weak_ptr<int>   w_ptr = sh_ptr;

sh_ptr = nullptr;
cout << w_ptr.expired(); // print 'true'
```

It must be converted to `std::shared_ptr` in order to access the referenced object

`std::weak_ptr` methods

- `use_count()` returns the number of objects referring to the same managed object
- `reset(ptr)` replaces the managed object with `ptr`
- `expired()` checks whether the referenced object was already deleted (`true`, `false`)
- `lock()` creates a `std::shared_ptr` that manages the referenced object

```cpp
#include <memory>

auto sh_ptr1 = std::make_shared<int>();
cout << sh_ptr1.use_count();  // print 1
std::weak_ptr<int> w_ptr = sh_ptr1;
cout << w_ptr.use_count();    // print 1

auto sh_ptr2 = w_ptr.lock();
cout << w_ptr.use_count();    // print 2 (sh_ptr1 + sh_ptr2)

sh_ptr1 = nullptr;
cout << w_ptr.expired();      // print false
sh_ptr2 = nullptr;
cout << w_ptr.expired();      // print true
```

Concurrency

Overview

C++11 introduces the **Concurrency** library to simplify managing OS threads

```cpp
#include <iostream>
#include <thread>

void f() {
    std::cout << "first thread" << std::endl;
}

int main(){
    std::thread th(f);
    th.join();          // stop the main thread until "th" complete
}
```

How to compile:

```
$g++ -std=c++11 main.cpp -pthread
```

Example

```cpp
#include <iostream>
#include <thread>
#include <vector>
void f(int id) {
    std::cout << "thread " << id << std::endl;
}
int main() {
    std::vector<std::thread> thread_vect; // thread vector
    for (int i = 0; i < 10; i++)
        thread_vect.push_back( std::thread(&f, i) );

    for (auto& th : thread_vect)
        th.join();

    thread_vect.clear();
    for (int i = 0; i < 10; i++) {  // thread + lambda expression
        thread_vect.push_back(
            std::thread( [](){ std::cout << "thread\n"; } );
    }
}
```

Library methods:

- `std::this_thread::get_id()` returns the thread id
- `std::thread::sleep_for( sleep_duration )`
  Blocks the execution of the current thread for at least the specified `sleep_duration`
- `std::thread::hardware_concurrency()` returns the number of concurrent threads supported by the implementation

Thread object methods:

- `get_id()` returns the thread id
- `join()` waits for a thread to finish its execution
- `detach()` permits the thread to execute independently of the thread handle

```cpp
#include <chrono>   // the following program should (not deterministic)
#include <iostream> // produces the output:
#include <thread>   //    child thread exit
                    //    main thread exit
int main() {
    using namespace std::chrono_literals;
    std::cout << std::this_thread::get_id();
    std::cout << std::thread::hardware_concurrency(); // e.g. print 6

    auto lambda = []() {
        std::this_thread::sleep_for(1s); // t2
        std::cout << "child thread exit\n";
    };
    std::thread child(lambda);
    child.detach(); // without detach(), child must join() the
                    // main thread (run-time error otherwise)
    std::this_thread::sleep_for(2s);    // t1
    std::cout << "main thread exit\n";
}
// if t1 < t2 the should program prints:
```

Parameters Passing

Parameters passing *by-value* or *by-pointer* to a thread function works in the same way of a standard function. *Pass-by-reference* requires a special wrapper (`std::ref` , `std::cref`) to avoid wrong behaviors

```
#include <iostream>
#include <thread>
void f(int& a, const int& b) {
    a = 7;
    const_cast<int&>(b) = 8;
}
int main() {
    int a = 1, b = 2;
    std::thread th1(f, a, b);                       // wrong!!!
    std::cout << a << ", " << b << std::endl;       // print 1, 2!!

    std::thread th2(f, std::ref(a), std::cref(b)); // correct
    std::cout << a << ", " << b << std::endl;       // print 7, 8!!
    th1.join(); th2.join();
}
```

The following code produces (in general) a value < 1000:

```
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>
void f(int& value) {
    for (int i = 0; i < 10; i++) {
        value++;
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
int main() {
    int value = 0;
    std::vector<std::thread> th_vect;
    for (int i = 0; i < 100; i++)
        th_vect.push_back( std::thread(f, std::ref(value)) );
    for (auto& it : th_vect)
        it.join();
    std::cout << value;
}
```

C++11 provides the **mutex** class as synchronization primitive to protect shared data from being simultaneously accessed by multiple threads

mutex methods:

- `lock()` locks the *mutex*, blocks if the *mutex* is not available
- `try_lock()` tries to lock the *mutex*, returns if the *mutex* is not available
- `unlock()` unlocks the *mutex*

More advanced mutex can be found here: `en.cppreference.com/w/cpp/thread`

C++ includes three mutex wrappers to provide safe copyable/movable objects:

- `lock_guard` (C++11) implements a strictly scope-based mutex ownership wrapper
- `unique_lock` (C++11) implements movable mutex ownership wrapper
- `shared_lock` (C++14) implements movable shared mutex ownership wrapper

```cpp
#include <thread> // iostream, vector, chrono

void f(int& value, std::mutex& m) {
    for (int i = 0; i < 10; i++) {
        m.lock();
        value++;   // other threads must wait
        m.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
int main() {
    std::mutex m;
    int value = 0;
    std::vector<std::thread> th_vect;
    for (int i = 0; i < 100; i++)
        th_vect.push_back( std::thread(f, std::ref(value), std::ref(m)) );
    for (auto& it : th_vect)
        it.join();
    std::cout << value;
}
```

Atomic

`std::atomic` (C++11) class template defines an atomic type that are implemented with lock-free operations (much faster than locks)

```cpp
#include <atomic> // chrono, iostream, thread, vector
void f(std::atomic<int>& value) {
    for (int i = 0; i < 10; i++) {
        value++;
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
int main() {
    std::atomic<int> value(0);
    std::vector<std::thread> th_vect;
    for (int i = 0; i < 100; i++)
        th_vect.push_back( std::thread(f, std::ref(value)) );
    for (auto& it : th_vect)
        it.join();
    std::cout << value;    // print 1000
}
```

The **future** library provides facilities to obtain values that are returned and to catch exceptions that are thrown by *asynchronous* tasks

Asynchronous call: `std::future async(function, args...)`
runs a function asynchronously (potentially in a new thread)
and returns a `std::future` object that will hold the result

std::future methods:

- `T get()` returns the result
- `wait()` waits for the result to become available

async() can be called with two launch policies for a task executed:

- `std::launch::async` a new thread is launched to execute the task asynchronously
- `std::launch::deferred` the task is executed on the calling thread the first time its result is requested (lazy evaluation)

```cpp
#include <future> // numeric, algorithm, vector, iostream
template <typename RandomIt>
int parallel_sum(RandomIt beg, RandomIt end) {
    auto len = end - beg;
    if (len < 1000)    // base case
        return std::accumulate(beg, end, 0);

    RandomIt mid = beg + len / 2;
    auto handle = std::async(std::launch::async, // right side
                             parallel_sum<RandomIt>, mid, end);
    int sum = parallel_sum(beg, mid);            // left side
    return sum + handle.get();                   // left + right
}
int main() {
    std::vector<int> v(10000, 1); // init all to 1
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end());
}
```

Modern C++ Programming

20. Performance Optimization I

Basic Concepts

Federico Busato

2024-03-29

Table of Contents

Table of Contents

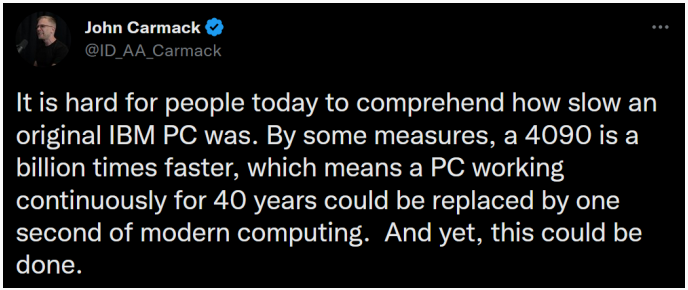
2 Basic Concepts

3 Basic Architecture Concepts

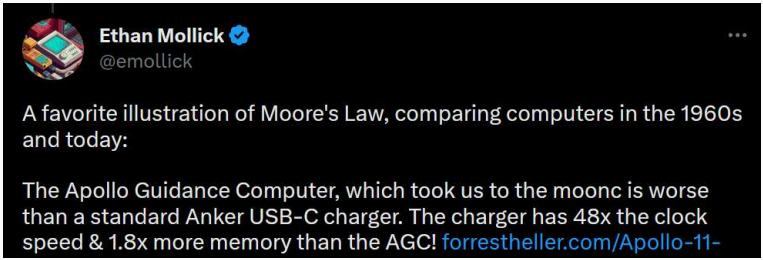
Table of Contents

Introduction

Performance and Technological Progress

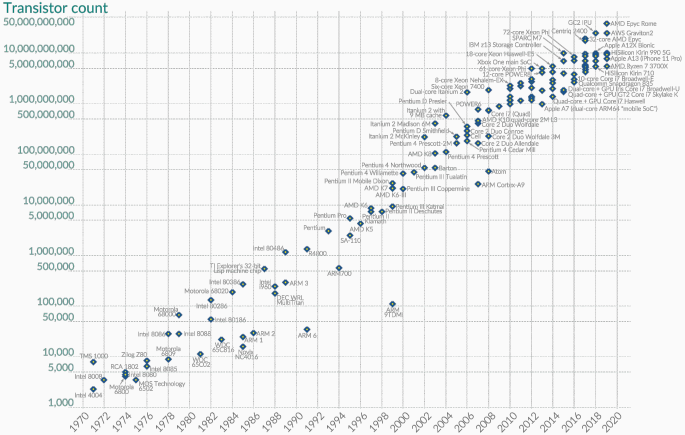


Performance and Technological Progress

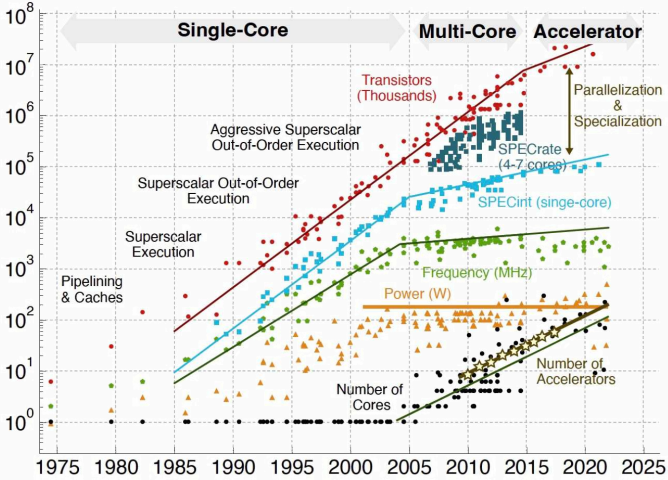


"The number of transistors incorporated in a chip will approximately double every 24 months." (40% per year)

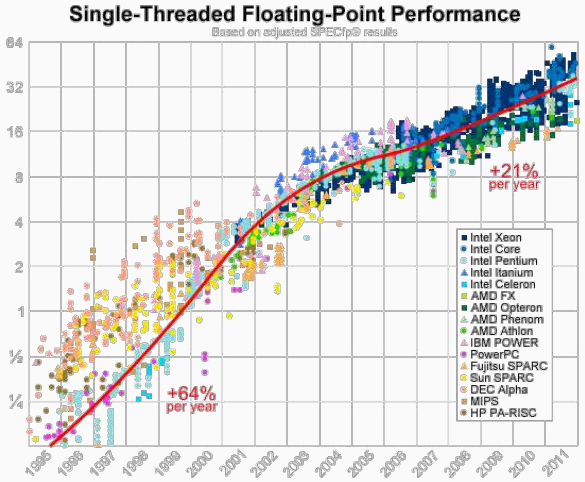
***Gordon Moore**, Intel co-founder*



The Moore's Law is not (yet) dead, but the same concept is not true for *clock frequency*, *single-thread performance*, *power consumption*, and *cost*



Single-Thread Performance Trend

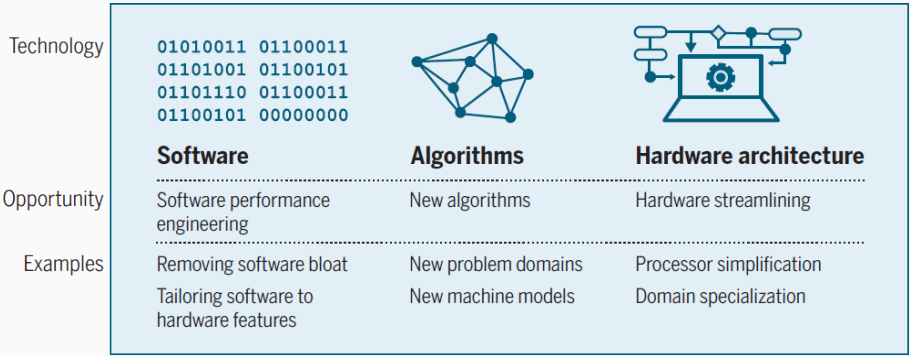


A Look Back at Single-Threaded CPU Performance

Herb Sutter, The Free Lunch Is Over

Higher performance over time is not merely dictated by the number of transistors.

Specific hardware improvements, software engineering, and algorithms play a crucial rule in driving the computer performance.



Specialized Hardware

Reduced precision, matrix multiplication engine, and sparsity provided orders of magnitude performance improvement for AI applications

Forget Moore's Law. Algorithms drive technology forward

"Algorithmic improvements make more efficient use of existing resources and allow computers to do a task faster, cheaper, or both. Think of how easy the smaller MP3 format made music storage and transfer. That compression was because of an algorithm."

- Forget Moore's Law
- What will drive computer performance after Moore's law?
- Heeding Huang's Law

Poisson's equation solver on a cube of size $N = n^3$

Year	Method	Reference	Storage	Complexity
1947	GE (banded)	Von Neumann & Goldstine	n^5	$\rightarrow \boldsymbol{n^7}$
1950	Optimal SOR	Reid	n^3	$n^4 \log n$
1971	CG	Young	n^3	$n^{3.5} \log n$
1984	MG	Brandt	n^3	$\rightarrow \boldsymbol{n^3}$

Reasons for Optimizing

- In the first decades, the *computer performance was extremely limited*. Low-level optimizations were essential to fully exploit the hardware
- Modern systems provide much higher performance, but *we cannot more rely on hardware improvement* on short-period
- Performance and efficiency add market value (fast program for a given task), e.g. search, page loading, etc.
- Optimized code uses less resources, e.g. in a program that runs on a server for months or years, a small reduction in the execution time/power consumption translates in a big saving of power consumption

Going the Other Way

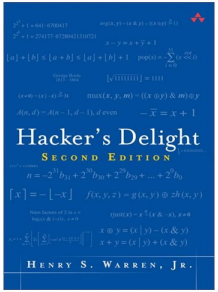
- Computing systems are unfathomably complex
- Optimization is complicated and surprising
- Doing something sensible had opposite effect
- We often try clever things that don't work

- How about trying something silly then?

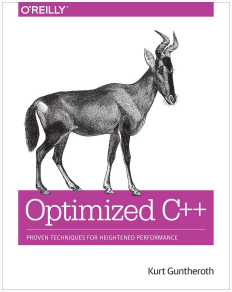
25 / 72

from *"Speed is Found in the Minds of People"*,
Andrei Alexandrescu, CppCon 2019

Optimization Books



Hacker's Delight (2nd)
H. S. Warren, 2016



Optimized C++
K. Guntheroth, 2014

References

- `Awesome C/C++ performance optimization resources`, *Bartlomiej Filipek*
- `Optimizing C++`, *wikibook*
- `Optimizing software in C++`, *Agner Fog*
- `Algorithmica: Algorithms for Modern Hardware`
- `What scientists must know about hardware to write fast code`

Basic Concepts

The **asymptotic analysis** refers to estimate the execution time or memory usage as function of the input size (the *order of growing*)

The *asymptotic behavior* is opposed to a *low-level analysis* of the code (instruction/loop counting/weighting, cache accesses, etc.)

Drawbacks:

- The *worst-case* is not the *average-case*
- Asymptotic complexity does not consider small inputs (think to *insertion sort*)
- The hidden constant can be relevant in practice
- Asymptotic complexity does not consider instructions cost and hardware details

Be aware that only **real-world problems** with a small asymptotic complexity or small size can be solved in a *"user" acceptable time*

Three examples:

- *Sorting*: $\mathcal{O}(n \log n)$, try to sort an array of some billion elements
- *Diameter of a (sparse) graph*: $\mathcal{O}(V^2)$, just for graphs with a few hundred thousand vertices it becomes impractical without advanced techniques
- *Matrix multiplication*: $\mathcal{O}(N^3)$, even for small sizes N (e.g. 8K, 16K), it requires special accelerators (e.g. GPU, TPU, etc.) for achieving acceptable performance

Time-Memory Trade-off

The **time-memory trade-off** is a way of solving a problem or calculation in less time by using more storage space (less often the opposite direction)

Examples:

- *Memoization* (e.g. used in dynamic programming): returning the cached result when the same inputs occur again
- *Hash table*: number of entries vs. efficiency
- *Lookup tables*: precomputed data instead branches
- *Uncompressed data*: bitmap image vs. jpeg

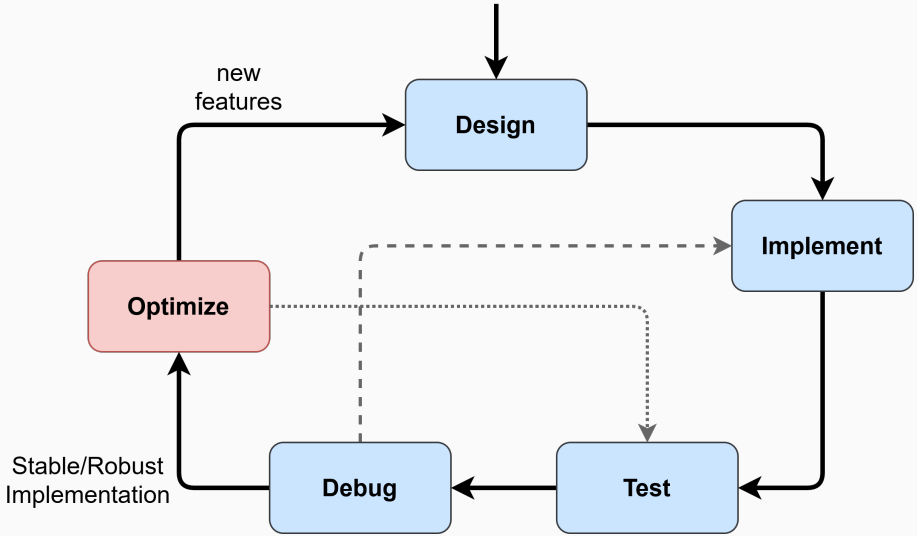
"If you're not writing a program, don't use a programming language"
***Leslie Lamport**, Turing Award*

"First solve the problem, then write the code"

"Inside every large program is an algorithm trying to get out"
***Tony Hoare**, Turing Award*

"Premature optimization is the root of all evil"
***Donald Knuth**, Turing Award*

"Code for correctness first, then optimize!"


new
features
Design
Implement
Optimize
Stable/Robust
Implementation
Debug
Test

- One of the most important phase of the optimization cycle is the **application profiling** for finding regions of code that are *critical for performance* (**hotspot**)
  - → Expensive code region (absolute)
  - → Code regions executed many times (cumulative)

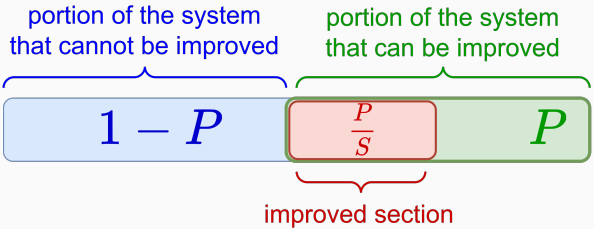
- Most of the time, **there is no the perfect algorithm for all cases** (e.g. insertion, merge, radix sort). Optimizing also refers in finding the correct heuristics for different program inputs/platforms instead of modifying the existing code

Ahmdal's Law

The **Ahmdal's law** expresses the maximum improvement possible by improving a particular part of a system

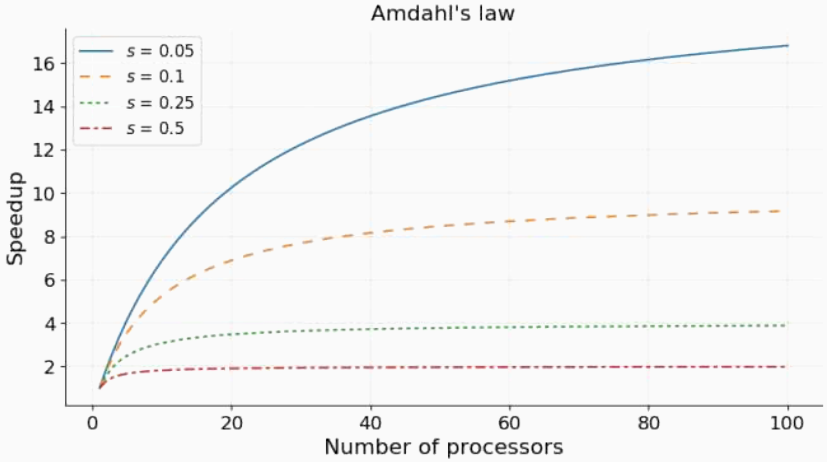
Observation: The performance of any system is constrained by the speed of the slowest point

S : improvement factor expressed as a factor of P



$$\text{Overall Improvement} = \frac{1}{(1-P) + \frac{P}{S}}$$

P \ S	25%	50%	75%	2x	3x	4x	5x	10x	∞
10%	1.02x	1.03x	1.04x	1.05x	1.07x	1.08x	1.09x	1.10x	1.11x
20%	1.04x	1.07x	1.09x	1.11x	1.15x	1.18x	1.19x	1.22x	1.25x
30%	1.06x	1.11x	1.15x	1.18x	1.25x	1.29x	1.31x	1.37x	1.49x
40%	1.09x	1.15x	1.20x	1.25x	1.36x	1.43x	1.47x	1.56x	1.67x
50%	1.11x	1.20x	1.27x	1.33x	1.50x	1.60x	1.66x	1.82x	2.00x
60%	1.37x	1.25x	1.35x	1.43x	1.67x	1.82x	1.92x	2.17x	2.50x
70%	1.16x	1.30x	1.43x	1.54x	1.88x	2.10x	2.27x	2.70x	3.33x
80%	1.19x	1.36x	1.52x	1.67x	2.14x	2.50x	2.78x	3.57x	5.00x
90%	1.22x	1.43x	1.63x	1.82x	2.50x	3.08x	3.57x	5.26x	10.00x



note: **s** is the portion of the system that cannot be improved

Throughput, Bandwidth, Latency

The **throughput** is the rate at which operations are performed

Peak throughput:
(CPU speed in Hz) x (CPU instructions per cycle) x
(number of CPU cores) x (number of CPUs per node)

NOTE: modern processors have more than one computation unit

The **memory bandwidth** is the amount of data that can be loaded from or stored into a particular memory space

Peak bandwidth:
(Frequency in Hz) x (Bus width in bit / 8) x (Pump rate, memory type multiplier)

The **latency** is the amount of time needed for an operation to complete

The performance of a program is *bounded* by one or more aspects of its computation. This is also strictly related to the underlying hardware

- **Memory-bound**. The program spends its time primarily in performing *memory accesses*. The performance is limited by the *memory bandwidth* (rarely memory-bound also refers to the amount of memory available)
- **Compute-bound** (Math-bound). The program spends its time primarily in computing *arithmetic instructions*. The performance is limited by the *speed of the CPU*

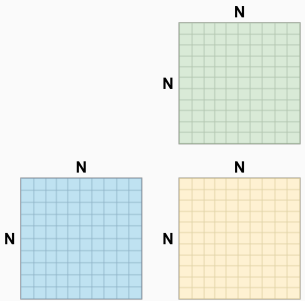
- **Latency-bound**. The program spends its time primarily in waiting *the data are ready* (instruction/memory dependencies). The performance is limited by the *latency of the CPU/memory*

- **I/O Bound**. The program spends its time primarily in performing *I/O operations* (network, user input, storage, etc.). The performance is limited by the *speed of the I/O subsystem*

Arithmetic Intensity

Arithmetic/Operational Intensity is the ratio of total operations to total data movement (bytes or words)

The naive matrix multiplication algorithm requires $N^3 \cdot 2$ floating-point operations* (multiplication + addition), while it involves $(N^2 \cdot 4\text{B}) \cdot 3$ data movement



* What Is a Flop?

$$R = \frac{ops}{bytes} = \frac{2n^3}{12n^2} = \frac{n}{6}$$

which means that for every byte accessed, the algorithm performs $\frac{n}{6}$ operations $\rightarrow$ **compute-bound**

N	Operations	Data Movement	Ratio	Exec. Time
512	$268 \cdot 10^6$	3 MB	85	2 ms
1024	$2 \cdot 10^9$	12 MB	170	21 ms
2048	$17 \cdot 10^9$	50 MB	341	170 ms
4096	$137 \cdot 10^9$	201 MB	682	1.3 s
8192	$1 \cdot 10^{12}$	806 MB	1365	11 s
16384	$9 \cdot 10^{12}$	3 GB	2730	90 s

A modern CPU performs 100 GFlops, and has about 50 GB/s memory bandwidth

Basic Architecture Concepts

Instruction Throughput, In-Order, and Out-of-Order Execution

The *processor throughput*, namely the number of instructions that can be executed in a unit of time, is measured in **Instruction per Cycle (IPC)**.
It is worth noting that most instructions require multiple clock cycles (**Cycles Per Instruction, CPI**). Therefore improving the IPC requires advanced hardware support

In-Order Execution (IOE) refers to the sequential processing of instructions in the exact order they appear in the program

Out-of-Order Execution (**OOE**) refers to the execution of instructions based on the availability of input data and execution units, rather than their original order in a program executed in a unit of time

Out-of-order execution on a *scalar processor* (single instruction at a time) is implemented through **instruction pipeling** which consists in dividing instructions into stages performed by different processor units, allowing different parts of instructions to be processed in parallel

Instruction pipeling breaks up the processing of instructions into several steps, allowing the processor to avoid stalls that occur when the data needed to execute an instruction is not immediately available. The processor avoid stalls by filling slots with other instructions that are ready

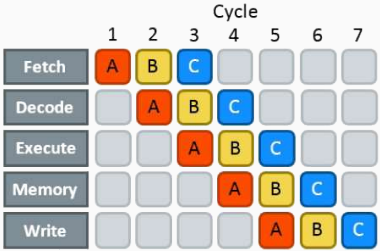
Fetch: The processor retrieves an instruction from memory

Decode: Instruction interpretation and preparation for execution, determining what operations it calls for

Execute: The processor carries out the instruction

Memory Access: Reading from or writing to memory (if needed)

Write-back: The results of the instruction execution are written back to the processor's registers or memory



Microarchitecture	Pipeline stages
Core	14
Bonnell	16
Sandy Bridge	14
Silvermont	14 to 17
Haswell	14
Skylake	14
Kabylake	14

The *pipeline efficiency* is affected by

- **Instruction stalls**, e.g. cache miss, an execution unit not available, etc.
- **Bad speculation**, branch misprediction

A **superscalar processor** is a type of microprocessor architecture that allows for the execution of *multiple instructions in parallel during a single clock cycle*. This is achieved by incorporating multiple execution units within the processor

The concept should not be confused with *instruction pipelining*, which decompose the instruction processing in stages. Modern processors combine both techniques to improve the IPC

Instruction-Level Parallelism (ILP) is a measure of how many instructions in a computer program can be executed simultaneously by issuing *independent* instructions in sequence.
ILP is achieved with *out-of-order execution* or the *SIMT* programming model

```
for (int i = 0; i < N; i++) // with no optimizations, the loop
    C[i] = A[i] * B[i];     // is executed in sequence
```

can be rewritten as:

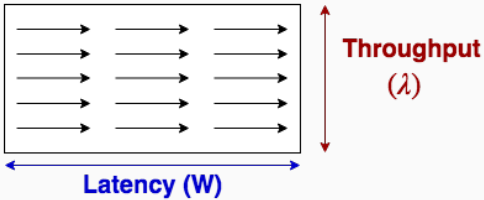
```
for (int i = 0; i < N; i += 4) {     // four independent multiplications
    C[i]     = A[i]     * B[i];      // per iteration
    C[i + 1] = A[i + 1] * B[i + 1]; // A, B, C are not alias
    C[i + 2] = A[i + 2] * B[i + 2];
    C[i + 3] = A[i + 3] * B[i + 3];
}
```

ILP and Little's Law

The **Little's Law** expresses the relation between *latency* and *throughput*. The *throughput* of a system λ is equal to the number of elements in the system divided by the average time spent (*latency*) W for each element in the system:

$$L = \lambda W \quad \rightarrow \quad \lambda = \frac{L}{W}$$

- L: average number of customers in a store
- λ: arrival rate (*throughput*)
- W: average time spent (*latency*)



Data-Level Parallelism (DLP) and SIMD

Data-Level Parallelism (DLP) refers to the execution of the same operation on multiple data in parallel

Vector processors or *array processors* provide SIMD (*Single Instruction-Multiple Data*) or vector instructions for exploiting data-level parallelism

The popular vector instruction sets are:

`MMX` *MultiMedia eXtension*. 80-bit width (Intel, AMD)

`SSE` (SSE2, SSE3, SSE4) *Streaming SIMD Extensions*. 128-bit width (Intel, AMD)

`AVX` (AVX, AVX2, AVX-512) *Advanced Vector Extensions*. 512-bit width (Intel, AMD)

`NEON` *Media Processing Engine*. 128-bit width (ARM)

`SVE` (SVE, SVE2) *Scalable Vector Extension*. 128-2048 bit width (ARM)

Thread-Level Parallelism (TLP)

A **thread** is a single sequential execution flow within a program with its state (instructions, data, PC, register state, and so on)

Thread-level parallelism (TLP) refers to the execution of separate computation *"thread"* on different processing units (e.g. CPU cores)

Single Instruction Multiple Threads (SIMT)

An alternative approach to the classical data-level parallelism is **Single Instruction Multiple Threads (SIMT)**, where multiple threads execute the same instruction simultaneously, with each thread operating on different data.

GPUs are successful examples of SIMT architectures.

SIMT can be thought of as an evolution of *SIMD* (Single Instruction Multiple Data). *SIMD* requires that all data processed by the instruction be of the same type and requires no dependencies or inter-thread communication. On the other hand, **SIMT** is more flexible and does not have these restrictions. Each thread has access to its own memory and can operate independently.

RISC, CISC Instruction Sets

The **Instruction Set Architecture** (ISA) is an abstract model of the CPU to represent its behavior. It consists of addressing modes, instructions, data types, registers, memory architecture, interrupt, etc.
It does not define how an instruction is processed

The **microarchitecture** (μarch) is the implementation of an **ISA** which includes pipelines, caches, etc.

Complex Instruction Set Computer (CISC)

- Complex instructions for special tasks even if used infrequently
- Assembly instructions follow software. Little compiler effort for translating high-level language into assembly
- Initially designed for saving cost of computer memory and disk storage (1960)
- High number of instructions with different size
- Instructions require complex micro-ops decoding (translation) for exploiting ILP
- Multiple low-level instructions per clock but with high latency

Hardware implications

- High number of transistors
- Extra logic for decoding. Heat dissipation
- Hard to scale

RISC

Reduced Instruction Set Computer (RISC)

- Simple instructions
- Small number of instructions with fixed size
- 1 clock per instruction
- Assembly instructions does not follow software
- No instruction decoding

Hardware implications

- High ILP, easy to schedule
- Small number of transistors
- Little power consumption
- Easy to scale

Instruction Set Comparison

x86 Instruction set

```
MOV AX, 15; AH = 00, AL = 0Fh
AAA; AH = 01, AL = 05
RET
```

ARM Instruction set

```
MOV R3, # 10
AND R2, R0, # 0xF
CMP R2, R3
IT LT
BLT elsebranch
ADD R2. # 6
ADD R1. #1
elsebranch:
END
```

CISC vs. RISC

- **Hardware market**:
  - *RISC* (ARM, IBM): Qualcomm Snapdragon, Amazon Graviton, Nvidia Grace, Nintendo Switch, Fujitsu Fukaku, Apple M1, Apple Iphone/Ipod/Mac, Tesla Full Self-Driving Chip, PowerPC
  - *CISC* (Intel, AMD): all x86-64 processors

- **Software market**:
  - *RISC*: Android, Linux, Apple OS, Windows
  - *CISC*: Windows, Linux

- **Power consumption**:
  - *CISC*: Intel i5 10th Generation: 64W
  - *RISC*: Arm-based smartphone $<$ 5W

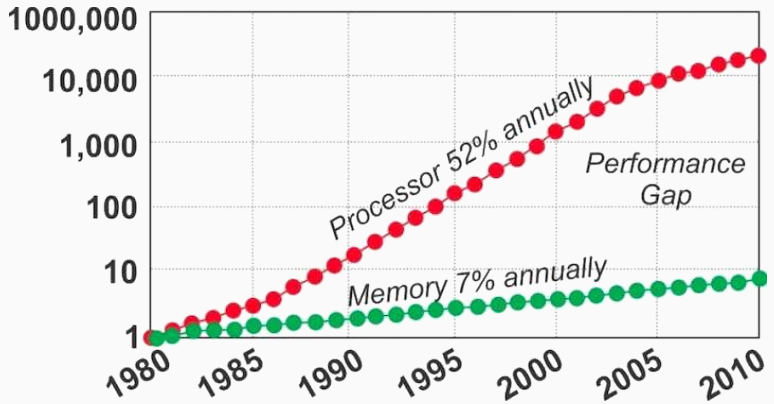
ARM Quote

"Incidentally, the first ARM1 chips required so little power, when the first one from the factory was plugged into the development system to test it, the microprocessor immediately sprung to life by drawing current from the IO interface – before its own power supply could be properly connected."

Happy birthday, ARM1. It is 35 years since Britain's Acorn RISC Machine chip sipped power for the first time

Memory Concepts

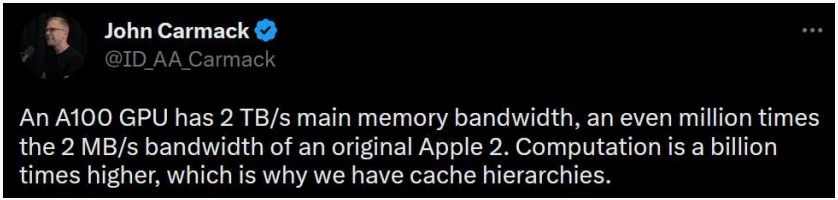
Access to memory dominates other costs in a processor



The Von Neumann Bottleneck

The efficiency of computer architectures is limited by the **Memory Wall** problem, namely the memory is the slowest part of the system

Moving data to and from main memory consumes the vast majority of *time* and *energy* of the system


John Carmack
@ID_AA_Carmack
An A100 GPU has 2 TB/s main memory bandwidth, an even million times the 2 MB/s bandwidth of an original Apple 2. Computation is a billion times higher, which is why we have cache hierarchies.

Modern architectures rely on complex memory hierarchy (primary memory, caches, registers, scratchpad memory, etc.). Each level has different characteristics and constrains (size, latency, bandwidth, concurrent accesses, etc.)

1 byte of RAM (1946)

IBM 5MB hard drive (1956)

Memory hierarchies	Intel Haswell E5-2650 v3	Intel KNL 7250 DDR5|MCDRAM	ARM Cortex A57
	10 cores 368 Gflop/s 105 Watts	68 cores 2662 Gflop/s 215 Watts	4 cores 32 Gflop/s 7 Watts
REGISTERS	16/core AVX2	32/core AVX-512	32/core
L1 CACHE & GPU SHARED MEMORY	32 KB/core	32 KB/core	32 KB/core
L2 CACHE	256 KB/core	1024 KB/2cores	2 MB
L3 CACHE	25 MB	0...16 GB	N/A
MAIN MEMORY	64 GB	384 |16 GB	4 GB
MAIN MEMORY BW	68 GB/s 5.4 flops/byte	115 | 421 GB/s 23 | 6 Flops/byte	26 GB/s 1.2 flops/byte
PCI EXPRESS GEN3x16 NVLINK	16 GB/s 23 flops/byte	16 GB/s 166 flops/byte	16 GB/s 2 flops/byte
INTERCONNECT INFINIBAND EDR	12 GB/s 30 flops/byte	12 GB/s 221 flops/byte	12 GB/s 2.6 flops/byte

Memory hierarchies for different type of architectures
Flops per byte transfer (all flop rates for 64 bit operands)

Source:

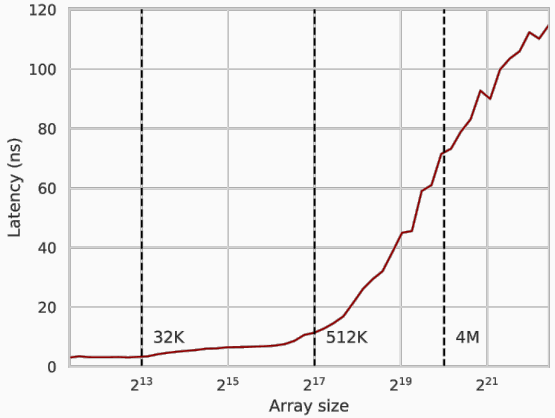
"Accelerating Linear Algebra on Small Matrices from Batched BLAS to Large Scale Solvers", ICL, University of Tennessee

`Intel Alder Lake 12th-gen Core-i9-12900k (Q1'21) + DDR4-3733` example:

Hierarchy level	Size	Latency	Latency Ratio	Bandwidth	Bandwidth Ratio
`L1 cache`	192 KB	1 ns	1.0x	1,600 GB/s	1.0x
`L2 cache`	1.5 MB	3 ns	3x	1,200 GB/s	1.3x
`L3 cache`	12 MB	6 - 20 ns	6-20x	900 GB/s	1.7x
`DRAM`	/	50 - 90 ns	50-90x	80 GB/s	20x
`SDD Disk (swap)`	/	70μs	10^5x	2 GB/s	800x
`HDD Disk (swap)`	/	10 ms	10^7x	2 GB/s	800x

- `en.wikichip.org/wiki/WikiChip`
- `Memory Bandwidth Napkin Math`

"Thinking differently about memory accesses, a good start is to get rid of the idea of $\mathcal{O}(1)$ memory access and replace it with $\mathcal{O}\sqrt{N}$" - The Myth of RAM



Algorithmica: Memory Latency

A **cache** is a small and fast memory located close to the processor that stores frequently used instructions and data. It is part of the processor package and takes 40 to 60 percent of the chip area

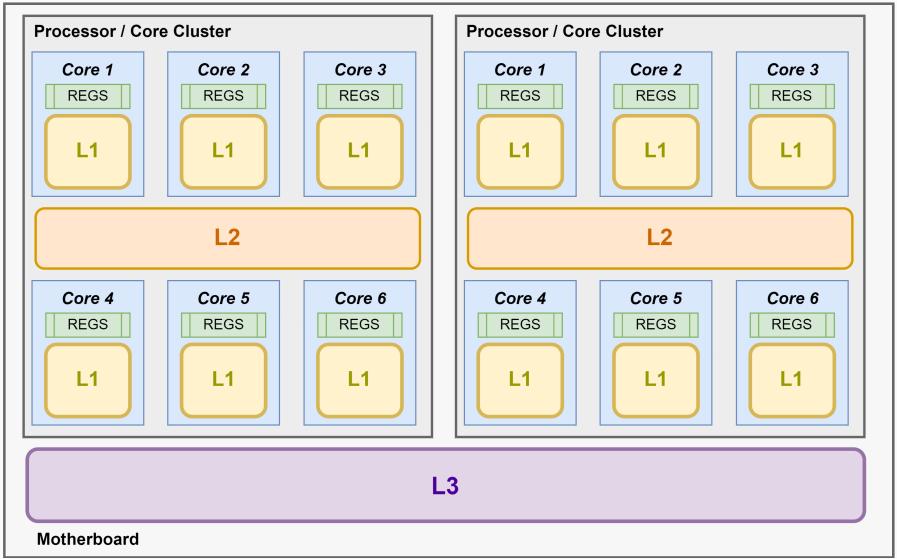
Characteristics and content:

Registers Program counter (PC), General purpose registers, Instruction Register (IR), etc.

L1 Cache Instruction cache and data cache, private/exclusive per CPU core, located on-chip

L2 Cache Private/exclusive per single CPU core or a cluster of cores, located off-chip

L3 Cache Shared between all cores and located off-chip (e.g. motherboard), up to 128/256MB


Processor / Core Cluster
Core 1
REGS
L1
Core 2
REGS
L1
Core 3
REGS
L1
L2
Core 4
REGS
L1
Core 5
REGS
L1
Core 6
REGS
L1
Processor / Core Cluster
Core 1
REGS
L1
Core 2
REGS
L1
Core 3
REGS
L1
L2
Core 4
REGS
L1
Core 5
REGS
L1
Core 6
REGS
L1
L3
Motherboard

A **cache line** or **cache block** is the unit of data transfer between the cache and main memory, namely the memory is loaded at the *granularity* of a cache line. A cache line can be further organized in banks or sectors

The typical size of the cache line is 64 bytes on x86-64 architectures (Intel, AMD), while it is 128 bytes on Arm64

Cache access type:

Hot	Closest-processor cached, L1
Warm	L2 or L3 caches
Cold	First load, cache empty

- A **cache hit** occurs when a requested data is *successfully found* in the cache memory
- The **cache hit rate** is the number of *cache hits divided by the number of memory requests*
- A **cache miss** occurs when a requested data is *not found* in the cache memory
- The **miss penalty** refers to the *extra time required to load the data* into cache from the main memory when a cache miss occurs
- A **page fault** occurs when a requested data is in the process address space, but *it is not currently located in the main memory* (swap/pagefile)
- Page **thrashing** occurs when page faults are frequent and the OS spends significant time to swap data in and out the physical RAM

Memory Locality

- **Spatial Locality** refers to the use of data elements within *relatively close storage locations* e.g. scan arrays in increasing order, matrices by row. It involves mechanisms such as *memory prefetching* and *access granularity* When spatial locality is low, many words in the cache line are not used

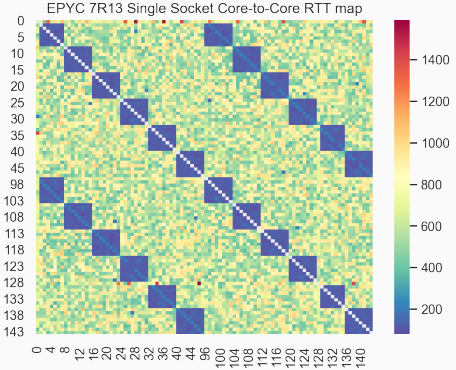
- **Temporal Locality** refers to the reuse of the same data within a relatively *small-time duration*, and, as consequence, exploit lower levels of the memory hierarchy (caches), e.g. multiple sparse accesses

  Heavily used memory locations can be accessed more quickly than less heavily used locations

Core-to-Core Latency

The slowing of Moore's Law and the collapse of Dennard scaling necessitated the hierarchical organization of caches and processors in the CPU. *Today, CPUs organize their cores into clusters, chiplets, and multi-sockets*. As a result, how execution threads are mapped to cores has a significant impact on the overall performance

Core-to-Core Latency Heatmap:



Thread Affinity

The **thread affinity** refers to the binding of a thread to a specific execution unit. The goal of *thread affinity* is improving the application performance by taking advantage of cache locality and optimizing resource usage

Setting CPU affinity can be done programmatically, such as using the `pthread_setaffinity_np` function for POSIX threads, or at OS level with the `taskset` command and the `sched_setaffinity` system call on Linux

*__Dennard Scaling__: power is proportional to the area of the transistor

Memory Ordering Model

- **Source code order**: The order in which the memory operations are specified in the source code, e.g. *subscript, dereferencing*
- **Program order**: The order in which the memory operations are specified at assembly level. Compilers can reorder instructions as part of the optimization process
- **Execution order**: The order in which the individual memory-reference instructions are executed on a given CPU, e.g., *out-of-order execution*
- **Perceived order**: The order in which a CPU perceives its memory operations. The perceived order can differ from the execution order due to caching, interconnect, and memory-system optimizations

Modern C++ Programming

21. Performance Optimization II

Code Optimization

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

Table of Contents

I/O Operations

I/O Operations are orders of magnitude slower than memory accesses

I/O Streams

In general, input/output operations are one of the most expensive

- Use `endl` for `ostream` only when it is strictly necessary (prefer `\n`)
- Disable *synchronization* with `printf/scanf`:
  `std::ios_base::sync_with_stdio(false)`
- Disable IO *flushing* when mixing `istream/ostream` calls:
  `<istream_obj>.tie(nullptr);`
- Increase IO *buffer size*:
  `file.rdbuf()->pubsetbuf(buffer_var, buffer_size);`

I/O Streams - Example

```cpp
#include <iostream>

int main() {
    std::ifstream fin;
    // ------------------------------------------------------
    std::ios_base::sync_with_stdio(false); // sync disable
    fin.tie(nullptr);                      // flush disable
                                           // buffer increase
    const int BUFFER_SIZE = 1024 * 1024;   // 1 MB
    char buffer[BUFFER_SIZE];
    fin.rdbuf()->pubsetbuf(buffer, BUFFER_SIZE);
    // ------------------------------------------------------
    fin.open(filename); // Note: open() after optimizations

    // IO operations
    fin.close();
}
```

printf

- `printf` is faster than `ostream` (see speed test link)
- A `printf` call with a simple format string ending with `\n` is converted to a `puts()` call

```
printf("Hello World\n");
printf("%s\n", string);
```

- No optimization if the string is not ending with `\n` or one or more `%` are detected in the format string

www.ciselant.de/projects/gcc_printf/gcc_printf.html

Memory Mapped I/O

A **memory-mapped file** is a segment of virtual memory that has been assigned a direct byte-for-byte correlation with some portion of a file

Benefits:

- Orders of magnitude faster than system calls
- Input can be "cached" in RAM memory (page/file cache)
- A file requires disk access only when a new page boundary is crossed
- Memory-mapping may bypass the page/swap file completely
- Load and store *raw* data (no parsing/conversion)

Memory Mapped I/O - Example


```
#if !defined(__linux__)
    #error It works only on linux
#endif
#include <fcntl.h>          //::open
#include <sys/mman.h>       //::mmap
#include <sys/stat.h>       //::open
#include <sys/types.h>      //::open
#include <unistd.h>         //::lseek
// usage: ./exec <file> <byte_size> <mode>
int main(int argc, char* argv[]) {
   size_t file_size = std::stoll(argv[2]);
   auto   is_read   = std::string(argv[3]) == "READ";
   int fd = is_read ? ::open(argv[1], O_RDONLY) :
                      ::open(argv[1], O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
   if (fd == -1)
       ERROR("::open")                    // try to get the last byte
   if (::lseek(fd, static_cast<off_t>(file_size - 1), SEEK_SET) == -1)
       ERROR("::lseek")
   if (!is_read && ::write(fd, "", 1) != 1) // try to write
       ERROR("::write")
```

```
auto mm_mode = (is_read) ? PROT_READ : PROT_WRITE;

// Open Memory Mapped file
auto mmap_ptr = static_cast<char*>(
                ::mmap(nullptr, file_size, mm_mode, MAP_SHARED, fd, 0) );

if (mmap_ptr == MAP_FAILED)
    ERROR("::mmap");
// Advise sequential access
if (::madvise(mmap_ptr, file_size, MADV_SEQUENTIAL) == -1)
    ERROR("::madvise");

// MemoryMapped Operations
// read from/write to "mmap_ptr" as a normal array: mmap_ptr[i]

// Close Memory Mapped file
if (::munmap(mmap_ptr, file_size) == -1)
    ERROR("::munmap");
if (::close(fd) == -1)
    ERROR("::close");
```

Consider using optimized (low-level) numeric conversion routines:

```
template<int N, unsigned MUL, int INDEX = 0>
struct fastStringToIntStr;

inline unsigned fastStringToUnsigned(const char* str, int length) {
    switch(length) {
        case 10: return fastStringToIntStr<10, 1000000000>::aux(str);
        case  9: return fastStringToIntStr< 9, 100000000>::aux(str);
        case  8: return fastStringToIntStr< 8, 10000000>::aux(str);
        case  7: return fastStringToIntStr< 7, 1000000>::aux(str);
        case  6: return fastStringToIntStr< 6, 100000>::aux(str);
        case  5: return fastStringToIntStr< 5, 10000>::aux(str);
        case  4: return fastStringToIntStr< 4, 1000>::aux(str);
        case  3: return fastStringToIntStr< 3, 100>::aux(str);
        case  2: return fastStringToIntStr< 2, 10>::aux(str);
        case  1: return fastStringToIntStr< 1, 1>::aux(str);
        default: return 0;
    }
}
```

```cpp
template<int N, unsigned MUL, int INDEX>
struct fastStringToIntStr {
    static inline unsigned aux(const char* str) {
        return static_cast<unsigned>(str[INDEX] - '0') * MUL  +
               fastStringToIntStr<N - 1, MUL / 10, INDEX + 1>::aux(str);
    }
};

template<unsigned MUL, int INDEX>
struct fastStringToIntStr<1, MUL, INDEX> {
    static inline unsigned aux(const char* str) {
        return static_cast<unsigned>(str[INDEX] - '0');
    }
};
```

Faster parsing: lemire.me/blog/tag/simd-swar-parsing

- Hard disk is orders of magnitude slower than RAM
- Parsing is faster than data reading
- Parsing can be avoided by using *binary* storage and `mmap`
- Decreasing the number of hard disk accesses improves the performance → **compression**

LZ4 is lossless compression algorithm providing *extremely fast decompression* up to 35% of `memcpy` and good compression ratio
`github.com/lz4/lz4`

Another alternative is **Facebook zstd**
`github.com/facebook/zstd`

Performance comparison of different methods for a file of 4.8 GB of integer values

Load Method	Exec. Time	Speedup
`ifstream`	102 667 ms	1.0x
`memory mapped + parsing (first run)`	30 235 ms	3.4x
`memory mapped + parsing (second run)`	22 509 ms	4.5x
`memory mapped + lz4 (first run)`	3 914 ms	26.2x
`memory mapped + lz4 (second run)`	1 261 ms	81.4x

NOTE: the size of the Lz4 compressed file is 1,8 GB

Memory Optimizations

Heap Memory

- *Dynamic heap allocation is expensive*: implementation dependent and interact with the operating system
- *Many small heap allocations are more expensive than one large memory allocation* The default page size on Linux is 4 KB. For smaller/multiple sizes, C++ uses a sub-allocator
- *Allocations within the page size is faster than larger allocations* (sub-allocator)

Stack Memory

- *Stack memory is faster than heap memory*. The stack memory provides high locality, it is small (cache fit), and its size is known at compile-time
- `static` stack allocations produce better code. It avoids filling the stack each time the function is reached
- `constexpr` arrays with dynamic indexing produces very inefficient code with GCC. Use `static constexpr` instead

```
void f(int x) {
// bad performance with GCC
//  constexpr        int array[] = {1,2,3,4,5,6,7,8,9};
    static constexpr int array[] = {1,2,3,4,5,6,7,8,9};
    return array[x];
}
```

Cache Utilization

Maximize cache utilization:

- Maximize spatial and temporal locality (see next examples)
- Prefer small data types
- Prefer `std::vector<bool>` over array of `bool`
- Prefer `std::bitset<N>` over `std::vector<bool>` if the data size is known in advance or bounded
- Prefer *stack* data structures *instead* of head data structures, e.g. `std::vector` vs. `static_vector`

A, B, C matrices of size $N \times N$

`C = A * B`

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        int sum = 0;
        for (int k = 0; k < N; k++)
            sum += A[i][k] * B[k][j]; // row × column
        C[i][j] = sum;
    }
}
```

$C = A * B^T$

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        int sum = 0;
        for (int k = 0; k < N; k++)
            sum += A[i][k] * B[j][k]; // row × row
        C[i][j] = sum;
    }
}
```

Benchmark:

N	64	128	256	512	1024
A * B	< 1 ms	5 ms	29 ms	141 ms	1,030 ms
A * B^T	< 1 ms	2 ms	6 ms	48 ms	385 ms
Speedup	/	2.5x	4.8x	2.9x	2.7x

Temporal-Locality Example

Speeding up a random-access function

```
for (int i = 0; i < N; i++)       // V1
    out_array[i] = in_array[hash(i)];
```

```
for (int K = 0; K < N; K += CACHE) { // V2
    for (int i = 0; i < N; i++) {
        auto x = hash(i);
        if (x >= K && x < K + CACHE)
            out_array[i] = in_array[x];
    }
}
```

`V1` : 436 ms, `V2` : 336 ms → 1.3x speedup (temporal locality improvement)
.. but it needs a careful evaluation of `CACHE`, and it can even decrease the performance for other sizes

pre-sorted `hash(i)` : 135 ms → 3.2x speedup (spatial locality improvement)

lemire.me/blog/2019/04/27

Data Alignment

Data alignment refers to placing data in memory at addresses that conform to certain boundaries, typically powers of two (e.g., 1, 2, 4, 8, 16 bytes, etc.)

Note: For multidimensional data, alignment only means that the start address of the data is aligned, not that all start offsets for all dimensions are aligned., e.g. for a 2D matrix, if `row[0][0]` is aligned doesn't imply that `row[0][1]` has the same property. Also the strides between rows need to be multiple of the alignment

Data alignment is classified in:

- **Internal alignment**: reducing memory footprint, optimizing memory bandwidth, and minimizing cache-line misses
- **External alignment**: minimizing cache-line misses, vectorization (SIMD instructions)

Internal Structure Alignment

```
struct A1 {
   char   x1; // offset 0
   double y1; // offset 8!! (not 1)
   char   x2; // offset 16
   double y2; // offset 24
   char   x3; // offset 32
   double y3; // offset 40
   char   x4; // offset 48
   double y4; // offset 56
   char   x5; // offset 64 (65 bytes)
}
```

```
struct A2 {   // internal alignment
   char   x1; // offset 0
   char   x2; // offset 1
   char   x3; // offset 2
   char   x4; // offset 3
   char   x5; // offset 4
   double y1; // offset 8
   double y2; // offset 16
   double y3; // offset 24
   double y4; // offset 32 (40 bytes)
}
```

Considering an *array of structures* (*AoS*), there are two problems:

- We are wasting 40% of memory in the first case (`A1`)
- In common x64 processors the cache line is 64 bytes. For the first structure `A1` , every access involves two cache line operations (2x slower)

see also `#pragma pack(1)`

External Structure Alignment and Padding

Considering the previous example for the structure `A2`, random loads from an array of structures `A2` leads to one or two cache line operations depending on the alignment at a specific index, e.g.

`index 0` $\rightarrow$ one cache line load

`index 1` $\rightarrow$ two cache line loads

It is possible to fix the structure alignment in two ways:

- The **memory padding** refers to introduce extra bytes at the end of the data structure to enforce the memory alignment
  e.g. add a `char` array of size 24 to the structure `A2`
- **Align keyword or attribute** allows specifying the alignment requirement of a type or an object (next slide)

C++ allows specifying the alignment requirement in different ways:

- Explicit padding for variable / struct declaration → affects `sizeof(T)`
  - C++11 `alignas(N)`
  - GCC/Clang: `__attribute__((aligned(N)))`
  - MSVC: `__declspec(align(N))`
- Explicit alignment for pointers
  - C++17 aligned `new` (e.g. `new int[2, N]`)
  - GCC/Clang: `__builtin_assume_aligned(x)`
  - Intel: `__assume_aligned(x)`

```cpp
struct alignas(16) A1 { // C++11
    int x, y;
};

struct __attribute__((aligned(16))) A2 { // compiler-specific attribute
    int x, y;
};

auto ptr1 = new int[100, 16]; // 16B alignment, C++17
auto ptr2 = new int[100];     // 4B alignment guarantee
auto ptr3 = __builtin_assume_aligned(ptr2, 16); // compiler-specific attribute
auto ptr4 = new A1[10];       // no aligment guarantee
```

Memory Prefetch

`__builtin_prefetch` is used to *minimize cache-miss latency* by moving data into a cache before it is accessed. It can be used not only for improving *spatial locality*, but also *temporal locality*

```
for (int i = 0; i < size; i++) {
    auto data = array[i];
    __builtin_prefetch(array + i + 1, 0, 1); // 2nd argument, '0' means read-only
                                             // 3th argument, '1' means
                                             // temporal locality=1, default=3
    // do some computation on 'data', e.g. CRC
}
```

The **CPU/threads affinity** controls how a process is mapped and executed over multiple cores (including sockets). It affects the process performance due to core-to-core communication and cache line invalidation overhead

Maximizing threads *"clustering"* on a single core can potentially lead to higher cache hits rate and faster communication. On the other hand, if the threads work independently/almost independently, namely they show high locality on their working set, mapping them to different cores can improve the performance

Arithmetic Types

Hardware Notes

- Instruction throughput greatly depends on processor model and characteristics, e.g., there is no hardware support for integer division on GPUs. This operation is translated to 100 instructions for 64-bit operands
- Modern processors provide separated units for floating-point computation (`FPU`)
- *Addition*, *subtraction*, and *bitwise operations* are computed by the `ALU`, and they have very similar throughput
- In modern processors, *multiplication* and *addition* are computed by the same hardware component for decreasing circuit area → multiplication and addition can be fused in a single operation `fma` (floating-point) and `mad` (integer)

Data Types

- **32-bit integral vs. floating-point**: in general, integral types are faster, but it depends on the processor characteristics

- **32-bit types are faster than 64-bit types**
  - 64-bit integral types are slightly slower than 32-bit integral types. Modern processors widely support native 64-bit instructions for most operations, otherwise they require multiple operations
  - Single precision floating-points are up to three times faster than double precision floating-points

- **Small integral types are slower than 32-bit integer**, but they require less memory $\rightarrow$ cache/memory efficiency

- Arithmetic **increment/decrement** `x++` / `x--` has the same performance of `x + 1` / `x - 1`
- Arithmetic **compound operators** (`a *= b`) has the same performance of assignment + operation (`a = a * b`) *
- **Prefer prefix increment/decrement** (`++var`) instead of the postfix operator (`var++`) *

* the compiler automatically applies such optimization whenever possible. This is not ensured for object types

- **Keep near constant values/variables** → the compiler can merge their values
- Some operations on **`unsigned types`** are faster than on **`signed types`** because they don't have to deal with negative numbers, e.g. `x / 2` → `x >> 1`
- Some operations on **`signed types`** are faster than on **`unsigned types`** because they can exploit *undefined behavior*, see next slide
- Prefer **logic operations** `||` to **bitwise operations** `|` to take advantage of short-circuiting

```
bool mainGuT(uint32_t i1, uint32_t i2, // if i1, i2 are int32_t, the code
             uint8_t *block) {         // uses half of the instructions!!
    uint8_t c1, c2;
    // 1                               // why? if i1, i2 are uint32_t the compiler
    c1 = block[i1], c2 = block[i2];    // must copy them into 32-bit registers to
    if (c1 != c2) return (c1 > c2);    // ensure wrap-around behavior before passing
    i1++, i2++;                        // them to the subscript operator (size_t)

    // 2                               // On the other hand, int32_t overflow is
    c1 = block[i1], c2 = block[i2];    // undefined behavior and the compiler can
    if (c1 != c2) return (c1 > c2);    // assume it never happens
    i1++, i2++;
    // ... continue repeating the     // the code is also optimal with size_t on 64-bit
}   //     code multiple times        // arch because block cannot be larger than it
```

Garbage In, Garbage Out: Arguing about Undefined Behavior with Nasal Daemons, Chandler Carruth, CppCon 2016

Arithmetic Operations - Integer Multiplication

Integer multiplication requires double the number of bits of the operands

```
// 32-bit platforms

int f1(int x, int y) {
    return x * y;                       // efficient but can overflow
}

int64_t f2(int64_t x, int64_t y) {      // same for f2(int x, int64_t y)
    return x * y;                       // always correct but slow
}

int64_t f3(int x, int y) {
    return x * static_cast<int64_t>(y); // correct and efficient!!
}
```

Arithmetic Operations - Power-of-Two Multiplication/Division/Modulo

- Prefer shift for **power-of-two multiplications** (`a << b`) and **divisions** (`a >> b`) only for run-time values *

- Prefer bitwise AND (`a % b → a & (b - 1)`) for **power-of-two modulo** operations only for run-time values *

- **Constant multiplication and division** can be heavily optimized by the compiler, even for non-trivial values

---

* the compiler automatically applies such optimizations if `b` is known at compile-time. Bitwise operations make the code harder to read

`Ideal divisors:  when a division compiles down to just a multiplication`

Conversion

From	To	Cost
`Signed`	`Unsigned`	no cost, bit representation is the same
`Unsigned`	`Larger Unsigned`	no cost, register extended
`Signed`	`Larger Signed`	1 clock-cycle, register + sign extended
`Integer`	`Floating-point`	4-16 clock-cycles Signed → Floating-point is faster than Unsigned → Floating-point (except `AVX512` instruction set is enabled)
`Floating-point`	`Integer`	fast if `SSE2`, slow otherwise (50-100 clock-cycles)

Floating-Point Division

Multiplication is much faster than division*

not optimized:

```
// "value" is floating-point (dynamic)
for (int i = 0; i < N; i++)
    A[i] = B[i] / value;
```

optimized:

```
div = 1.0 / value;    // div is floating-point
for (int i = 0; i < N; i++)
    A[i] = B[i] * div;
```

* Multiplying by the inverse is not the same as the division see lemire.me/blog/2019/03/12

Floating-Point FMA

Modern processors allow performing `a * b + c` in a single operation, called **fused multiply-add** (`std::fma` in C++11). This implies better performance and accuracy

CPU processors perform computations with a larger register size than the original data type (e.g. 48-bit for 32-bit floating-point) for performing this operation

Compiler behavior:

- GCC 9 and ICC 19 produce a single instruction for `std::fma` and for `a * b + c` with `-O3 -march=native`
- Clang 9 and MSVC 19.* produce a single instruction for `std::fma` but not for `a * b + c`

`FMA: solve quadratic equation`

`FMA: extended precision addition and multiplication by constant`

Compiler intrinsics are highly optimized functions directly provided by the compiler instead of external libraries

Advantages:

- Directly mapped to hardware functionalities if available
- Inline expansion
- Do not inhibit high-level optimizations, and they are portable contrary to `asm` code

Drawbacks:

- Portability is limited to a specific compiler
- Some intrinsics do not work on all platforms
- The same instricics can be mapped to a non-optimal instruction sequence depending on the compiler

Most compilers provide intrinsics **bit-manipulation functions** for `SSE4.2` or `ABM` (Advanced Bit Manipulation) instruction sets for Intel and AMD processors

GCC examples:

`__builtin_popcount(x)` count the number of one bits

`__builtin_clz(x)` (`count leading zeros`) counts the number of zero bits following the most significant one bit

`__builtin_ctz(x)` (`count trailing zeros`) counts the number of zero bits preceding the least significant one bit

`__builtin_ffs(x)` (`find first set`) index of the least significant one bit

gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

- Compute integer log2

```
inline unsigned log2(unsigned x) {
    return 31 - __builtin_clz(x);
}
```

- Check if a number is a power of 2

```
inline bool is_power2(unsigned x) {
    return __builtin_popcount(x) == 1;
}
```

- Bit search and clear

```
inline int bit_search_clear(unsigned x) {
    int pos = __builtin_ffs(x); // range [0, 31]
    x       &= ~(1u << pos);
    return pos;
}
```

Example of intrinsic portability issue:

`__builtin_popcount()` GCC produces `__popcountdi2` instruction while Intel Compiler (ICC) produces 13 instructions

`_mm_popcnt_u32` GCC and ICC produce `popcnt` instruction, but it is available only for processor with support for `SSE4.2` instruction set

More advanced usage

- Compute CRC: `_mm_crc32_u32`
- AES cryptography: `_mm256_aesenclast_epi128`
- Hash function: `_mm_sha256msg1_epu32`

`software.intel.com/sites/landingpage/IntrinsicsGuide/`

Using intrinsic instructions is extremely dangerous if the target processor does not natively support such instructions

Example:

> *"If you run code that uses the intrinsic on hardware that doesn't support the* `lzcnt` *instruction, the results are unpredictable" - MSVC*

on the contrary, GNU and clang `__builtin_*` instructions are always well-defined. The instruction is translated to a non-optimal operation sequence in the worst case

The instruction set support should be checked at *run-time* (e.g. with `__cpuid` function on MSVC), or, when available, by using compiler-time macro (e.g. `__AVX__`)

Automatic Compiler Function Transformation

`std::abs` can be recognized by the compiler and transformed to a hardware instruction

In a similar way, C++20 provides a portable and efficient way to express bit operations `<bit>`

rotate left :	`std::rotl`
rotate right :	`std::rotr`
count leading zero :	`std::countl_zero`
count leading one :	`std::countl_one`
count trailing zero :	`std::countr_zero`
count trailing one :	`std::countr_one`
population count :	`std::popcount`

Why is the standard "abs" function faster than mine?

Value in a Range

Checking if a non-negative value *x* is within a range *[A, B]* can be optimized if *B* > *A* (useful when the condition is repeated multiple times)

```
if (x >= A && x <= B)

// STEP 1: subtract A
if (x - A >= A - A && x - A <= B - A)
// -->
if (x - A >= 0 && x - A <= B - A) // B - A is precomputed

// STEP 2
//   - convert "x - A >= 0" --> (unsigned) (x - A)
//   - "B - A" is always positive
if ((unsigned) (x - A) <= (unsigned) (B - A))
```

Value in a Range Examples

Check if a value is an uppercase letter:

```
uint8_t x = ...
if (x >= 'A' && x <= 'Z')
    ...
```

→

```
uint8_t x = ...
if (x - 'A' <= 'Z')
    ...
```

A more general case:

```
int x = ...
if (x >= -10 && x <= 30)
    ...
```

→

```
int x = ...
if ((unsigned) (x + 10) <= 40)
    ...
```

The compiler applies this optimization only in some cases (tested with GCC/Clang 9 -O3)

Lookup Table

Lookup table (LUT) is a *memoization* technique which allows replacing *runtime* computation with precomputed values

Example: a function that computes the logarithm base 10 of a number in the range [1-100]

```
template<int SIZE, typename Lambda>
constexpr std::array<float, SIZE> build(Lambda lambda) {
    std::array<float, SIZE> array{};
    for (int i = 0; i < SIZE; i++)
        array[i] = lambda(i);
    return array;
}
float log10(int value) {
    constexpr auto lamba = [](int i) { return std::log10f((float) i); };
    static constexpr auto table = build<100>(lambda);
    return table[value];
}
```

Make your lookup table do more

Low-Level Optimizations

Collection of low-level implementations/optimization of common operations:

- **Bit Twiddling Hacks**
  `graphics.stanford.edu/~seander/bithacks.html`
- **The Aggregate Magic Algorithms**
  `aggregate.org/MAGIC`
- **Hackers Delight Book**
  `www.hackersdelight.org`

Low-Level Information

The same instruction/operation may take different clock-cycles on different architectures/CPU type

- **Agner Fog - Instruction tables** (latencies, throughputs)
  `www.agner.org/optimize/instruction_tables.pdf`
- **Latency, Throughput, and Port Usage Information**
  `uops.info/table.html`

Control Flow

Computation is faster than decision

Pipelines are an essential element in modern processors. Some processors have up to 20 pipeline stages (14/16 typically)

The downside to long pipelines includes the danger of **pipeline stalls** that waste CPU time, and the time it takes to reload the pipeline on **conditional branch** operations (`if`, `while`, `for`)

- Prefer `switch` statements to multiple `if`
  - If the compiler does not use a jump-table, the cases are evaluated in order of appearance → the most frequent cases should be placed before
  - Some compilers (e.g. `clang`) are able to translate a sequence of `if` into a `switch`
- In general, `if` statements affect performance when the branch is taken
- Not all control flow instructions (or branches) are translated into `jump` instructions. If the code in the branch is small, the compiler could optimize it in a conditional instruction, e.g. `ccmovl`
  Small code section can be optimized in different ways [2] (see next slides)

[2] `Is this a branch?`

Minimize Branch Overhead

- **Branch prediction**: technique to guess which way a branch takes. It requires hardware support, and it is generically based on dynamic history of code executing
- **Branch predication**: a conditional branch is substituted by a sequence of instructions from both paths of the branch. Only the instructions associated to a *predicate* (boolean value), that represents the direction of the branch, are actually executed

```
int x = (condition) ? A[i] : B[i];
P = (condition) // P: predicate
@P  x = A[i];
@!P x = B[i];
```

- **Speculative execution**: execute both sides of the conditional branch to better utilize the computer resources and commit the results associated to the branch taken

Branch Hints - [[likely]] / [[unlikely]]

C++20 `[[likely]]` and `[[unlikely]]` provide a hint to the compiler to optimize a conditional statement, such as `while`, `for`, `if`

```
for (i = 0; i < 300; i++) {
    [[unlikely]] if (rand() < 10)
        return false;
}
```

```
switch (value) {
  [[likely]]   case 'A': return 2;
  [[unlikely]] case 'B': return 4;
}
```

Signed/Unsigned Integers

- Prefer **signed integer** for **loop indexing**. The compiler optimizes more aggressively such loops because integer overflow is not defined. Unsigned loop indexing generates complex intermediate expressions, especially for nested loops, that the compiler could not solve

- Prefer **32-bit signed integer** or **64-bit integer** for **any operation that is translated to 64-bit**. The most common is *array indexing*. The subscript operator implicitly defines its parameter as `size_t`. Any indexing operation with 32-bit unsigned integer requires the compiler to enforce wrap-around behavior, e.g. by moving the variable to a 32-bit register

```
unsigned v = ...;
// some operations on v
array[v];
```

Loops

- Prefer **square brackets** syntax `[]` over pointer arithmetic operations for array access to facilitate compiler loop optimizations (e.g. polyhedral loop transformations)
- Prefer range-based loop for iterating over a container [1]

[1] Branch predictor: How many 'if's are too many?
The Little Things: Everyday efficiencies

Loop Hoisting

Loop Hoisting, also called *loop-invariant code motion*, consists of moving statements or expressions outside the body of a loop *without affecting the semantics* of the program

Base case:

```
for (int i = 0; i < 100; i++)
    a[i] = x + y;
```

Better:

```
v = x + y;
for (int i = 0; i < 100; i++)
    a[i] = v;
```

Loop hoisting is also important in the evaluation of loop conditions

Base case:

```
// "x" never changes
for (int i = 0; i < f(x); i++)
   a[i] = y;
```

Better:

```
int limit = f(x);
for (int i = 0; i < limit; i++)
   a[i] = y;
```

In the worst case, `f(x)` is evaluated at every iteration (especially when it belongs to another translation unit)

Loop unrolling (or **unwinding**) is a loop transformation technique which optimizes the code by removing (or reducing) loop iterations

The optimization produces better code at the expense of binary size

Example:

```
for (int i = 0; i < N; i++)
    sum += A[i];
```

can be rewritten as:

```
for (int i = 0; i < N; i += 8) {
    sum += A[i];
    sum += A[i + 1];
    sum += A[i + 2];
    sum += A[i + 3];
    ...
} // we suppose N is a multiple of 8
```

Loop unrolling can make your code better/faster:

+ Improve instruction-level parallelism (ILP)
+ Allow vector (SIMD) instructions
+ Reduce control instructions and branches

Loop unrolling can make your code worse/slower:

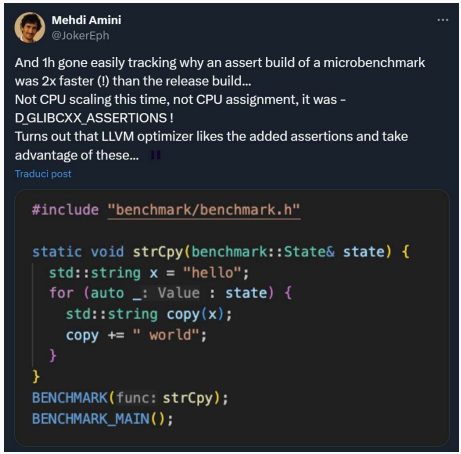
- Increase compile-time/binary size
- Require more instruction decoding
- Use more memory and instruction cache

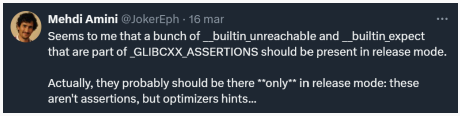
Unroll directive The `Intel`, `IBM`, and `clang` compilers (but not `GCC`) provide the preprocessing directive `#pragma unroll` (to insert above the loop) to force loop unrolling. The compiler already applies the optimization in most cases

`Why are unrolled loops faster?`

Assertions

Some compilers (e.g. `clang`) use assertions for optimization purposes: most likely code path, not possible values, etc. [3]





Compiler Hints - `[[assume]]`

C++23 allows defining an *assumption* in the code that is always true

```
int x = ...;
[[assume(x > 0)]]; // the compiler assume that 'x' is positive

int y = x / 2;     // the operation is translated in a single shift as for
                   // the unsigned case
```

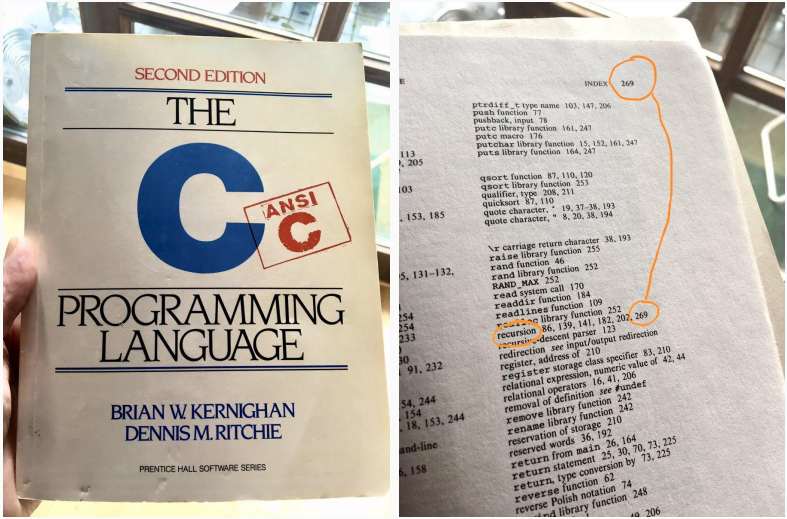
Compilers provide non-portable instructions for previous C++ standards: `__builtin_assume()` (`clang`), `__builtin_unreachable()` (`gcc`), `__assume()` (`msvc`, `icc`)

C++23 also provides `std::unreachable()` (`<utility>`) for marking unreachable code

Avoid run-time recursion (very expensive). Prefer *iterative* algorithms instead

Recursion cost: The program must store all variables (snapshot) at each recursion iteration on the stack, and remove them when the control return to the caller instance

The **tail recursion** optimization avoids maintaining caller stack and pass the control to the next iteration. The optimization is possible only if all computation can be executed before the recursive call



Via Twitter - Jan Wildeboer

Functions

Function Call Cost

Function call methods:

Direct Function address is known at compile-time

Indirect Function address is known only at run-time

Inline The function code is fused in the caller code (same translation unit or Link-time-optimization)

Direct/Indirect function call cost:

- The caller pushes the arguments on the stack in reverse order
- Jump to function address
- The caller clears (pop) the stack
- The function pushes the return value on the stack
- Jump to the caller address

The **optimal way** to pass and return arguments (*by-value*) to/from functions is in *registers*. It also avoid the pointer aliasing performance issue. The following conditions must be satisfied:

- The object is **trivially copyable**: No user-provided copy/move/default constructors, destructor, and copy/move assignment operators, no virtual functions, apply recursively to base classes and non-static data members
- Linux/Unix (`SystemV x86-64 ABI`): data types $\leq$ **16 bytes** (8B $\times$ 2), max **6 arguments**
- Windows (`x64 ABI`): data types $\leq$ **8 bytes**, max **4 arguments**

- `when are structs/classes passed and returned in registers?`
- `System V ABI - X86-64 Calling Convention`
- `x64 calling convention - Parameter Passing`

- If the previous conditions are not satisfied, the object is passed **by-reference**. In addition, objects that are not *trivially-copyable* could be expensive to pass *by-value* (copied).

- Pass **by-reference** and **by-pointer** introduce one level of indirection

- Pass **by-reference** is more efficient than pass **by-pointer** because it facilitates variable elimination by the compiler, and the function code does not require checking for `NULL` pointer

`const` modifier applied to values, pointers, references *does not produce better code* in most cases, but it is useful for ensuring read-only accesses

In some cases, pass `by-const` is beneficial for performance because `const` member function overloading could be cheaper than their counterparts

inline

`inline` specifier for optimization purposes is just a hint for the compiler that increases the heuristic threshold for **inlining**, namely copying the function body where it is called

```
inline void f() { ... }
```

- the compiler can ignore the hint
- *inlined* functions increase the binary size because they are expanded in-place for every function call

Compilers have different heuristics for function inlining

- Number of lines (even comments: `How new-lines affect the Linux kernel performance`)
- Number of assembly instructions
- Inlining depth (recursive)

GCC/Clang extensions allow to *force* inline/non-inline functions:

```
__attribute__((always_inline)) void f() { ... }
__attribute__((noinline))      void f() { ... }
```

- An Inline Function is As Fast As a Macro
- Inlining Decisions in Visual Studio

Inlining and Linkage

The compiler can *inline* a function only if it is independent from external references

- A function with *internal linkage* is not visible outside the current translation unit, so it can be aggressively *inlined*
- On the other hand, *external linkage* doesn't prevent function inlining if the function body is visible in a translation unit. In this situation, the compiler can duplicate the function code if it determines that there are no external references

Symbol Visibility

All compilers, except MSVC, export all function symbols → the symbols can be used in other translation units and this can prevent inlining

Alternatives:

- Use `static` functions
- Use `anonymous namespace` (functions and classes)
- Use GNU extension (also `clang`) `__attribute__((visibility("hidden")))`

gcc.gnu.org/wiki/Visibility

Function Attributes

Some compilers, including Clang, GCC, provide additional attributes to optimize function calls:

- `__attribute__((pure))` / `[[gnu::pure]]` *no side effects* on its parameters and no external global references (program state)
  → subject to data flow analysis and might be eliminated
- `__attribute__((const))` / `[[gnu::const]]` *depends only* on its parameters, no read from global references
  → subject to common sub-expression elimination and loop optimizations

note: the compiler is able to deduce such properties in most cases

---

`Implications of pure and constant functions`
`__attribute__((pure))` `function attribute`

Consider the following example:

```
// suppose f() is not inline
void f(int* input, int size, int* output) {
    for (int i = 0; i < size; i++)
        output[i] = input[i];
}
```

- The compiler cannot *unroll* the loop (sequential execution, no ILP) because `output` and `input` pointers can be **aliased**, e.g. `output = input + 1`

- The aliasing problem is even worse for more complex code and *inhibits all kinds of optimization* including code re-ordering, vectorization, common sub-expression elimination, etc.

Most compilers (included `GCC/Clang/MSVC`) provide **restricted pointers** (`__restrict`) so that the programmer asserts that the pointers are not aliased

```
void f(int* __restrict input,
       int             size,
       int* __restrict output) {
    for (int i = 0; i < size; i++)
        output[i] = input[i];
}
```

Potential benefits:

- Instruction-level parallelism
- Less instructions executed
- Merge common sub-expressions

Benchmarking matrix multiplication

```
void matrix_mul_v1(const int* A,
                   const int* B,
                   int        N,
                   int*       C) {
```

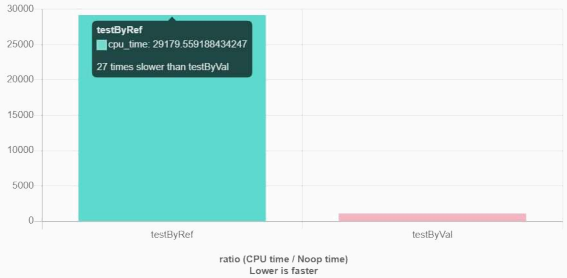
```
void matrix_mul_v2(const int* __restrict A,
                   const int* __restrict B,
                   int                  N,
                   int*       __restrict C) {
```

Optimization	-O1	-O2	-O3
v1	1,030 ms	777 ms	777 ms
v2	513 ms	510 ms	761 ms
Speedup	2.0x	1.5x	1.02x

```
void foo(std::vector<double>& v, const double& coeff) {
    for (auto& item : v) item *= std::sinh(coeff);
}
```

vs.

```
void foo(std::vector<double>& v, double coeff) {
    for (auto& item : v) item *= std::sinh(coeff);
}
```



Object-Oriented Programming

Variable/Object Scope

Declare local variable in the innermost scope

- the compiler can more likely fit them into registers instead of stack
- it improves readability

Wrong:

```
int i, x;
for (i = 0; i < N; i++) {
    x    = value * 5;
    sum += x;
}
```

Correct:

```
for (int i = 0; i < N; i++) {
    int x  = value * 5;
    sum   += x;
}
```

- C++17 allows local variable initialization in `if` and `while` statements, while C++20 introduces them for in *range-based loops*

Variable/Object Scope

Exception! Built-in type variables and passive structures should be placed in the innermost loop, while objects with constructors should be placed outside loops

```
for (int i = 0; i < N; i++) {
    std::string str("prefix_");
    std::cout << str + value[i];
} // str call CTOR/DTOR N times
```

```
std::string str("prefix_");
for (int i = 0; i < N; i++) {
    std::cout << str + value[i];
}
```

Object Optimizations

- Prefer **direct initialization** and *full object constructor* instead of two-step initialization (also for variables)
- Prefer **move semantic** instead of *copy constructor*. Mark *copy constructor* as `=delete` (sometimes it is hard to see, e.g. implicit)
- Use `static` for all members that do not use instance member (avoid passing `this` pointer)
- If the object semantic is *trivially copyable*, ensure **defaulted** `= default` *default/copy constructors* and *assignment operators* to enable vectorization

Object Dynamic Behavior Optimizations

- **Virtual calls** are slower than standard functions
  - Virtual calls prevent any kind of optimizations as function lookup is at runtime (loop transformation, vectorization, etc.)
  - Virtual call overhead is up to 20%-50% for function that can be inlined

- Mark `final` all `virtual` functions that are not overridden

- Avoid dynamic operations, e.g. `dynamic_cast`

- The Hidden Performance Price of Virtual Functions
- Investigating the Performance Overhead of C++ Exceptions

Object Operation Optimizations

- Minimize multiple `+` operations between objects to avoid temporary storage
- Prefer `x += obj`, instead of `x = x + obj` → avoid object copy and temporary storage
- Prefer `++obj` / `--obj` (return `&obj`), instead of `obj++`, `obj--` (copy and return old `obj`)

Object Implicit Conversion

```
struct A {   // big object
    int array[10000];
};
struct B {
    int array[10000];

    B() = default;

    B(const A& a) { // user-defined constructor
        std::copy(a.array, a.array + 10000, array);
    }
};
//------------------------------------------------------------------------
void f(const B& b) {}

A a;
B b;
f(b); // no cost
f(a); // very costly!! implicit conversion
```

`Std` Library and Other Language Aspects

- Avoid old `C` library routines such as `qsort`, `bsearch`, etc. Prefer **`std::sort`**, **`std::binary_search`** instead
  - `std::sort` is based on a hybrid sorting algorithm. Quick-sort / head-sort (introsort), merge-sort / insertion, etc. depending on the `std` implementation
  - Prefer `std::find()` for small array, `std::lower_bound`, `std::upper_bound`, `std::binary_search` for large sorted array

Function Optimizations

- `std::fill` applies `memset` and `std::copy` applies `memcpy` if the input/output are continuous in memory

- Use the same type for initialization in functions like `std::accumulate()`, `std::fill`

```
auto array = new int[size];
...
auto sum = std::accumulate(array, array + size, 0u);
// 0u != 0 → conversion at each step

std::fill(array, array + size, 0u);
// it is not translated into memset
```

Containers

- Use `std` container member functions (e.g. `obj.find()`) instead of external ones (e.g. `std::find()`). Example: `std::set` $O(log(n))$ vs. $O(n)$
- Be aware of container properties, e.g. `vector.push_vector(v)`, instead of `vector.insert(vector.begin(), value)` $\rightarrow$ entire copy of all vector elements
- Set `std::vector` size during the object construction (or use the `reserve()` method) if the number of elements to insert is known in advance $\rightarrow$ every implicit resize is equivalent to a copy of all vector elements
- Consider *unordered* containers instead of the standard one, e.g. `unorder_map` vs. `map`
- Prefer `std::array` instead of dynamic heap allocation

Critics to Standard Template Library (STL)

- Platform/Compiler-dependent implementation
- Execution order and results across platforms
- Debugging is hard
- Complex interaction with custom memory allocators
- Error handling based on exceptions is non-transparent
- Binary bloat
- Compile time (see `C++ Compile Health Watchdog`, and `STL Explorer`)

`STL isn't for *anyone*`

Other Language Aspects

- Prefer `lambda` expression (or `function object`) instead of `std::function` or function pointers

- Avoid dynamic operations: **exceptions** (and use `noexcept`), **smart pointer** (e.g. `std::unique_ptr`)

- Use `noexcept` decorator → program is aborted if an error occurred instead of raising an exception. see
  `Bitcoin: 9% less memory: make SaltedOutpointHasher noexcept`

Modern C++ Programming

22. Performance Optimization III

Non-Coding Optimizations and Benchmarking

Federico Busato

2024-03-29

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Compiler Optimizations

*"I always say the purpose of optimizing compilers is not to make code run faster, but to prevent programmers from writing utter **** in the pursuit of making it run faster"*

Rich Felker, `musl-libc` (`libc` alternative)

```
bool isEven(int number) {
    int  numberCompare = 0;
    bool even          = true;
    while (number != numberCompare) {
        even = !even;
        numberCompare++;
    }
    return even;
}
```

```
bool isEven(int number) {
    return number & 1u;
}
```

On the other hand, having a good compiler does not mean that it can fully optimize any code:

- The compiler does not *"understand"* the code, as opposed to human
- The compiler is *conservative* and applies optimizations only if they are safe and do not affect the correctness of computation
- The compiler is full of *models and heuristics* that could not match a specific situation
- The compiler *cannot spend large amount of time* in code optimization
- The compiler could consider *other targets* outside performance, e.g. binary size

Important advise: **Use an updated version of the compiler**

- Newer compiler produces **better/faster code**
  - Effective optimizations
  - Support for newer CPU architectures
- **New warnings** to avoid common errors and better support for existing error/warnings (e.g. code highlights)
- **Faster compiling, less memory usage**
- **Less compiler bugs**: compilers are very complex and they have many bugs

Use an updated version of the linker: e.g. for *Link Time Optimization*, `gold linker` or LLVM linker `lld`

Which compiler?

Answer: It dependents on the code and on the processor
example: `GCC 9 vs. Clang 8`

Some compilers can produce optimized code for specific architectures:

- **Intel Compiler** (commercial): Intel processors
- **IBM XL Compiler** (commercial): IBM processors/system
- **Nvidia NVC++ Compiler** (free/commercial): Multi-core processors/GPUs

---

- `gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
- `Intel Blog: gcc-x86-performance-hints`
- `Advanced Optimization and New Capa-bilities of GCC 10`

`-O0`, `/Od` Disables any optimization
- default behavior
- fast compile time

`-O1`, `/O1` Enables basic optimizations

`-O2`, `/O2` Enables advanced optimizations
- some optimization steps are expensive
- can increase the binary size

`-O3` Enable aggressive optimizations. Turns on all optimizations specified by -O2, plus some more
- `-O3` does not guarantee to produce faster code than `-O2`
- it could break floating-point IEEE754 rules in some non-traditional compilers (`nvc++`, `IBM xlc`)

`-O4` / `-O5`	It is an alias of `-O3` in some compilers, or it can refer to `-O3` + inter-procedural optimizations (basic, full) and high-order transformation (HOT) optimizer for specialized loop transformations
`-Ofast`	Provides other aggressive optimizations that may violate strict compliance with language standards. It includes `-O3 -ffast-math`
`-Os`, `/Os`	Optimize for size. It enables all `-O2` optimizations that do not typically increase code size (e.g. loop unrolling)
`-Oz`	Aggressively optimize for size
`-funroll-loops`	Enables loop unrolling (not included in `-O3`)
`-fopt-info`	Describes optimization passes and missed optimizations `-fopt-info-missed`

In general, enabling the following flags implies less floating-point accuracy, breaking the IEEE754 standard, and it is implementation dependent (not included in `-O3`)

`-fno-signaling-nans`	
`-fno-trapping-math`	Disable floating-point exceptions
`-mfma -ffp-contract=fast`	Force floating-point expression contraction such as forming of fused multiply-add operations
`-ffinite-math-only`	Disable special conditions for handling `inf` and `NaN`
`-fassociative-math`	Assume floating-point associative behavior

`-funsafe-math-optimizations` Allows breaking floating-point associativity and enables reciprocal optimization

`-ffast-math` Enables aggressive floating-point optimizations. All the previous, flush-to-zero denormal number, plus others

Beware of fast-math
Semantics of Floating Point Math in GCC

Linker Optimization Flags

`-flto` Enables *Link Time Optimizations* (Interprocedural Optimization). The linker merges all modules into a single combined module for optimization

- the linker must support this feature: `GNU ld v2.21++` or `gold` version, to check with `ld --version`
- it can significantly improve the performance
- in general, it is a very expensive step, even longer than the object compilations

`-fwhole-program` Assume that the current compilation unit represents the whole program being compiled → Assume that all non-extern functions and variables belong only to their compilation unit

Architecture-oriented optimizations are not included in other flags (`-O3`)

`-m64` In 64-bit mode the number of available registers increases from 6 to 14 general and from 8 to 16 `XMM`. Also, all 64-bits x86 architectures have `SSE2` extension by default. 64-bit applications can use more than 4GB address space

`-m32` 32-bit mode. It should be combined with `-mfpmath=sse` to enable using of `XMM` registers in floating point instructions (instead of stack in x87 mode). 32-bit applications can use less than 4GB address space

It is recommended to use 64-bits for High-Performance Computing applications and 32-bits for phone and tablets applications

`-march=<arch>`	Generates instructions for a specific processor to exploit exclusive hardware features. `<arch>` represents the minimum hardware supported by the binaries (not portable)
`-mtune=<tune_arch>`	Specifies the target microarchitecture. Generates optimized code for a class of processors without exploiting specific hardware features. Binaries are still compatibles with other processors, e.g. earlier CPUs in the architecture family (maybe slower than `-march`)
`-mcpu=<tune_arch>`	Deprecated synonym for `-mtune` for x86-64 processors, optimizes for both a particular architecture and microarchitecture on `Arm`
`-mfpu<fp_hw>`	(Arm) Optimize for a specific floating-point hardware
`-m<instr_set>`	(x86-64) Optimize for a specific instruction set

`<arch>`	`armv9-a`, `armv7-a+neon-vfpv4`, `znver4`, `core2`, `skylake`
`<tune_arch>`	`cortex-a9`, `neoverse-n2`, `generic`, `intel`
`<instr_set>`	`see2`, `avx512`
`<fp_hw>`	`neon`, `neon-fp-armv8`

- `<tune_arch>` should be always greater than `<arch>`
- In general, `-mtune` is set to `generic` if not specified
- `-march=native`, `-mtune=native`, `-mcpu=native`: Allows the compiler to determine the processor type (not always accurate)
- Especially with new compilers, prefer **auto-vectorization** to explicit vector intrinsics

---

- GCC Arm options, GCC X86-64 options
- Compiler flags across architectures: -march, -mtune, and -mcpu
- NVIDIA Grace CPU Benchmarking Guide, Arm Vector Instructions: SVE and NEON

Help the Compiler to Produce Better Code

- Grouping variables and functions related to each other in the same translation unit
- Define *global variables* and *functions* in the translation unit in which they are used more often
- *Global variables* and functions that are not used by other translation units should have *internal linkage* (*anonymous namespace*/ `static` function)

Static library linking helps the linker to optimize the code across different modules (link-time optimizations). Dynamic linking prevents these kinds of optimizations

Profile Guided Optimization (PGO) is a compiler technique aims at improving the application performance by reducing instruction-cache problems, reducing branch mispredictions, etc. *PGO provides information to the compiler about areas of an application that are most frequently executed*

It consists in the following steps:

(1) Compile and *instrument* the code

(2) *Run* the program by exercising the most used/critical paths

(3) *Compile again* the code and exploit the information produced in the previous step

The particular options to instrument and compile the code are compiler specific

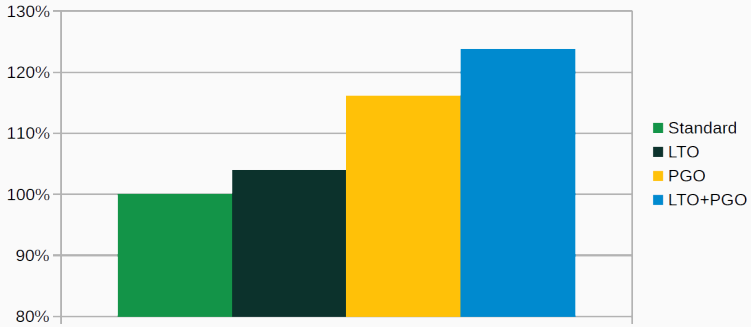
GCC

```
$ gcc -fprofile-generate my_prog.c my_prog # program instrumentation
$ ./my_prog # run the program (most critial/common path)
$ gcc -fprofile-use -O3 my_prog.c my_prog  # use instrumentation info
```

Clang

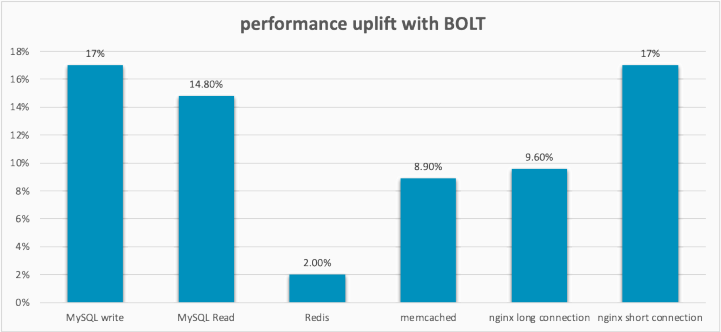
```
$ clang++ -fprofile-instr-generate my_prog.c my_prog
$ ./my_prog
$ xcrun llvm-profdata merge -output default.profdata default.profraw
$ clang++ -fprofile-instr-use=default.profdata -O3 my_prog.c my_prog
```

PGO, LTO Performance



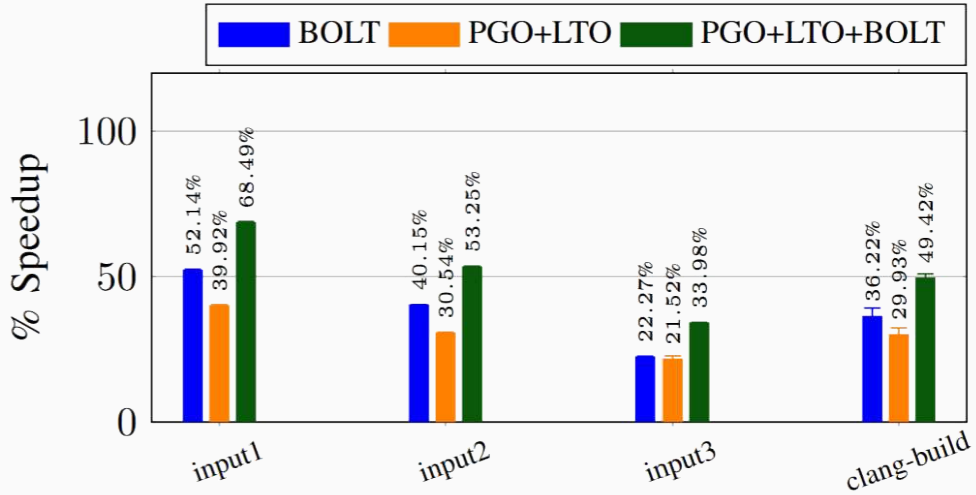
SPEC 2017 built with GCC 10.2 and -O2

The code layout in the final binary can be further optimized with a **post-link binary optimizer** and **layout optimization** like `BOLT` or `Propeller` (sampling or instrumentation profile)

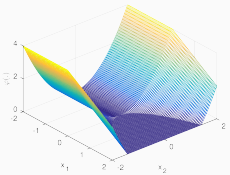


`BOLT: A Practical Binary Optimizer for Data Centers and Beyond`

`BOLT optimization technology could bring obvious performance uplift on arm server`


BOLT
PGO+LTO
PGO+LTO+BOLT
% Speedup
100
50
0
52.14%
39.92%
68.49%
40.15%
30.54%
53.25%
22.27%
21.52%
33.98%
36.22%
29.93%
49.42%
input1
input2
input3
clang-build

Polyhedral Optimizations



Polyhedral optimization is a compilation technique that rely on the representation of programs, especially those involving nested loops and arrays, in *parametric polyhedra*. Thanks to combinatorial and geometrical optimizations on these objects, the compiler is able to analyze and optimize the programs including *automatic parallelization*, *data locality*, *memory management*, *SIMD instructions*, and *code generation for hardware accelerators*

`Polly` is a high-level loop and data-locality optimizer and optimization infrastructure for LLVM

`PLUTO` is an automatic parallelization tool based on the polyhedral model

see also `Using Polly with Clang`

Compiler Transformation Techniques

Help the Compiler to Produce Better Code

Overview on compiler code generation and transformation:

- `Optimizations in C++ Compilers`
  Matt Godbolt, *ACM Queue*
- `Compiler Optimizations`

- **Constant folding**. Direct evaluation constant expressions at compile-time

```
const int K = 100 * 1234 / 2;
```

- **Constant propagation**. Substituting the values of known constants in expressions at compile-time

```
const int K = 100 * 1234 / 2;
const int J = K * 25;
```

- **Common subexpression elimination**. Avoid computing identical and redundant expressions

```
int x = y * z + v;
int y = y * z + k; // y * z is redundant
```

- **Induction variable elimination**. Eliminate variables whose values are dependent (induction)

```
for (int i = 0; i < 10; i++)
    x = i * 8;
// "x" can be derived by knowing the value of "i"
```

- **Dead code elimination**. Elimination of code which is executed but whose result is never used, e.g. dead store

```
int a = b * c;
... // "a" is never used, "b * c" is not computed
```

Unreachable code elimination instead involves removing code that is never executed

- **Use-define chain**. Avoid computations related to a variable that happen before its definition

```
x = i * k + l;
x = 32; // "i * k + l" is not needed
```

- **Peephole optimization**. Replace a small set of low-level instructions with a faster sequence of instructions with better performance and the same semantic. The optimization can involve pattern matching

```
imul    eax, eax, 8 // a * 8
sal     eax, 3      // a << 3 (shift)
```

Loop Unswitching

- **Loop Unswitching**. Split the loop to improve data locality, reduce loop instructions (especially branches), and allow additional optimizations

```
for (i = 0; i < N; i++) {
    if (x)
        a[i] = 0;
    else
        b[i] = 0;
}
```

```
if (x) {
    for (i = 0; i < N; i++)
        a[i] = 0; // use memset
}
else {
    for (i = 0; i < N; i++)
        b[i] = 0; // use memset
}
```

Loop Fusion

- **Loop Fusion** (jamming). Merge multiple loops to improve data locality and perform additional optimizations

```
for (i = 0; i < 300; i++)
    a[i] = a[i] + sqrt(i);
for (i = 0; i < 300; i++)
    b[i] = b[i] + sqrt(i);
```

```
for (i = 0; i < 300; i++) {
    a[i] = a[i] + sqrt(i); // sqrt(i) is computed only
    b[i] = b[i] + sqrt(i); // one time
}
```

Loop Fission

- **Loop Fission** (distribution). Split a loop in multiple loops to

```
for (i = 0; i < size; i++) {
    a[i] = b[rand()]; // cache pollution
    c[i] = d[rand()];
}
```

```
for (i = 0; i < size; i++)
    a[i] = b[rand()]; // better cache utilization
for (i = 0; i < size; i++)
    c[i] = d[rand()];
```

Loop Interchange

- **Loop Interchange**. Exchange the order of loop iterations to improve data locality and perform additional optimizations (e.g. vectorization)

```
for (i = 0; i < 1000000; i++) {
    for (j = 0; j < 100; j++)
        a[j * x + i] = ...; // low locality
}
```

```
for (j = 0; j < 100; j++) {
    for (i = 0; i < 1000000; i++)
        a[j * x + i] = ...; // high locality
}
```

Loop Tiling

- **Loop Tiling** (blocking, nest optimization). Partition the iterations of multiple loops to exploit data locality

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++)
        a[j * N + i] = ...; // low locality
}
```

```
for (i = 0; i < N; i += TILE_SIZE) {
    for (j = 0; j < M; j += TILE_SIZE) {
        for (k = 0; k < TILE_SIZE; k++) {
            for (l = 0; l < TILE_SIZE; l++) {
```

Libraries and Data Structures

Consider using optimized *external* libraries for critical program operations

- **Compressed Bitmask:** set algebraic operations
  - `BitMagic Library`
  - `Roaring Bitmaps`
- **Ordered Map/Set:** `B+Tree` as replacement for red-black tree
  - `STX B+Tree`
  - `Abseil B-Tree`
- **Hash Table:** (replace for `std::unsorted_set/map`)
  - `Google Sparse/Dense Hash Table`
  - `bytell hashmap`
  - `Facebook F14 memory efficient hash table`
  - `Abseil Hashmap` (2x-3x faster)
  - `Robin Hood Hashing`
  - `Comprehensive C++ Hashmap Benchmarks 2022`

- **Probabilistic Set Query:** Bloom filter, 'XOR filter, Facebook's Ribbon Filter, Binary Fuse filter
- **Scan, print, and formatting:** fmt library, scn library instead of iostream or printf/scanf
- **Random generator:** PCG random generator instead of Mersenne Twister or Linear Congruent
- **Non-cryptographic hash algorithm:** xxHash instead of CRC
- **Cryptographic hash algorithm:** BLAKE3 instead of MD5 or SHA

- **Linear Algebra:** `Eigen`, `Armadillo`, `Blaze`
- **Sort:**
  - `Beating Up on Qsort`. Radix-sort for non-comparative elements (e.g. `int`, `float`)
  - `Vectorized and performance-portable Quicksort`
- `malloc` **replacement:**
  - `tcmalloc` (Google)
  - `mimalloc` (Microsoft)

Libraries and `Std` replacements

- **`Folly`**: Performance-oriented `std` library (Facebook)
- **`Abseil`**: Open source collection of C++ libraries drawn from the most fundamental pieces of Google's internal codebase
- **`Frozen`**: Zero-cost initialization for immutable containers, fixed-size containers, and various algorithms.

A curated list of awesome header-only C++ libraries

Performance Benchmarking

Performance benchmarking is a non-functional test focused on measuring the efficiency of a given task or program under a particular load

Performance benchmarking is hard!!

Main reasons:

- What to test?
- Workload/Dataset quality
- Cache behavior
- Stable CPU performance
- Program memory layout
- Measurement overhead
- Compiler optimizations
- Metric evaluation

What to Test?

1. **Identify performance metrics**: The metric(s) should be strongly related to the specific problem and that allows a comparison across different systems, e.g. elapsed time is not a good metric in general for measuring the throughput
   - Matrix multiplication: FLoating-point Operation Per Second (FLOP/S)
   - Graph traversing: Edge per Second (EPS)

2. **Plan performance tests**: Determine what part of the problem is relevant for solving the given problem, e.g. excluding initialization process
   - Suppose a routine that requires different steps and ask a memory buffer for each of them. Memory allocations should be excluded as a user could use a memory pool

Workload/Dataset Quality

1. **Stress the most important cases**: Rare or edge cases that are not used in real-world applications or far from common usage are less important, e.g. a graph problem where all vertices are not connected

2. **Use datasets that are well-known in the literature and reproducible**. Don't use "self-made" dataset and, if possible, use public available resources

3. **Use a reproducible test methodology**. Trying to remove sources of "noise", e.g. if the procedure is randomized, the test should be use with the same seed. It is not always possible, e.g. OS scheduler, atomic operations in parallel computing, etc.

see also `Reproducibility in artificial intelligence`

- *Cache behavior is not deterministic*. Different executions lead to different hit rates
- After a data is loaded from the main memory, it remains in the cache until it expires or is evicted to make room for new content
- Executing the same routine multiple times, the first run is much slower than the other ones due to the cache effect (warmup run)

There is no a systematic way to flush the cache. Some techniques to ensure more reliable performance results are

- overwrite all data involved in the computation between each runs
- read/write between two buffers of size at least the size of the largest cache
- some processors, such as ARM, provide specific instructions to *invalidate* the cache `__builtin___clear_cache()`, `__clear_cache()`

Note: manual cache invalidation must consider cache locality (e.g. L1 per CPU core) and compiler optimizations that can remove useless code (solution: use global variables and `volatile`)

see: `Is there a way to flush the entire CPU cache related to a program?`

One of the first source of fluctuation in performance measurement is due to unstable CPU frequency

Dynamic frequency scaling, also known as *CPU throttling*, automatically decreases the CPU frequency for:

- Power saving, extending battery life
- Decrease fan noise and chip heat
- Prevent high frequency damage

Modern processors also comprise advanced technologies to automatically **raise CPU operating frequency when demanding tasks are running** (e.g. Intel® Turbo Boost). Such technologies allow processors to run with the *highest possible frequency* for limited amount of time depending on different factors like *type of workload*, *number of active cores*, *power consumption*, *temperature*, etc.

Get CPU info:

- *CPU characteristics*:
  `lscpu`
- *Monitor CPU clocks in real-time*:
  `cpupower monitor -m Mperf`
- *Get CPU clocks info*:
  `cpupower frequency-info`
  see “`cpufreq governors`”

- *Disable Turbo Boost*
  `echo 1 >> /sys/devices/system/cpu/intel_pstate/no_turbo`
- *Disable hyper threading*
  `echo 0 > /sys/devices/system/cpu/cpuX/online`
  or through BIOS
- *Use* "`performance`" *scaling governor*
  `sudo cpupower frequency-set -g performance`
- *Set CPU affinity* (CPU-Program binding) `taskset -c <cpu_id> <program>`
- *Set process priority* `sudo nice -n -5 taskset -c <cpu_id> <process>`

- *Disable address space randomization*
  `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`
- *Drop file system cache* (if the benchmark involves IO ops)
  `echo 3 | sudo tee /proc/sys/vm/drop_caches; sync`
- *CPU isolation*
  don't schedule process and don't run kernels code on the selected CPUs. GRUB options: `isolcpus=<cpu_ids>,rcu_nocbs=<cpu_ids>`

- How to get consistent results when benchmarking on Linux?
- How to run stable benchmarks
- Best Practices When Benchmarking CUDA Applications

Multi-Threads Considerations

- `numactl --interleave=all`
  NUMA: Non-Uniform Memory Access (e.g. multi-socket system)
  The default behavior is to allocate memory in the same node as a thread is scheduled to run on, and this works well for small amounts of memory. However, when you want to allocate more than a single node memory, it is no longer possible. This option sets interleaved memory allocations among NUMA nodes

- `export OMP_NUM_THREADS=96` Set the number of threads in an OpenMP program

Program Memory Layout

A small code change modifies the memory program layout
$\rightarrow$ large impact on cache (up to 40%)

- **Linking**
  - link order $\rightarrow$ changes function addresses
  - upgrade a library

- **Environment Variable Size**: moves the program stack
  - run in a new directory
  - change username

- `Performance Matters`, *E. Berger*, CppCon20
- `Producing Wrong Data Without Doing Anything Obviously Wrong!`, *Mytkowicz et al.*, ASPLOS'09

Measurement Overhead

Time-measuring functions could introduce significant overhead for small computation

`std::chrono::high_resolution_clock::now()` / `std::chrono::system_clock::now()` rely on library/OS-provided functions to retrieve timestamps (e.g. `clock_gettime`) and their execution can take several clock cycles

Consider using a **benchmarking framework**, such as `Google Benchmark` or `nanobench` (`std::chrono` based), to retrieve hardware counters and get basic profiling info

Compiler Optimizations

Compiler optimizations could distort the actual benchmark

- *Dead code elimination:* the compiler discards code that does not perform "useful" computation
- *Constant propagation/Loop optimization:* the compiler is able to pre-compute the result of simple codes
- *Instruction order:* the compiler can even move the time-measuring functions

Other Considerations

The actual values for a benchmark could significantly affect the results. For instance, a GEMM operation could show 2X performance between matrices filled with zeros and random values due to the effect on power consumption

After extracting and collecting performance results, it is fundamental to report/summarize them in a way to fully understand the experiment, provide interpretable insights, ensure reliability, and compare different observations, e.g. codes, algorithms, systems, etc.

Metric	Formula	Description
Arithmetic mean	$\bar{x} = \sum_{i=1}^{n} x_i$	For summarizing costs, e.g. exec. times, floating point ops, etc.
Harmonic mean	$\frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$	For summarizing rates, e.g. flop/s
Geometric mean	$\sqrt[n]{\prod_{i=1}^{n} x_i}$	For summarizing rates. Harmonic mean should be preferred. Commonly used for comparing speedup
Standard deviation	$\sigma = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}$	Measure of the spread of normally distributed samples
Coefficient of Variation	$\frac{std.dev}{arith.mean}$	Represents the stability of a set of normally distributed measurement results. Normalized standard deviation

Metric	Formula	Description
Confidence intervals of the mean	$z = t\left(n-1, \frac{\alpha}{2}\right)$ $CI = \left[\bar{x} - \frac{z\sigma}{\sqrt{n}}, \bar{x} + \frac{z\sigma}{\sqrt{n}}\right]$	Measure of reliability of the experiment. The concept is interpreted as the probability (e.g. $\alpha = 95\%$) that the observed confidential interval contains the true mean
Median	value at position $n/2$ after sorting all data	Rank measures are more robust with regard to outliers but do not consider all measured values
Quantile: Percentile/Quartile	value at a given position after sorting all data	The percentiles/quartiles provide information about the spread of the data and the skew. It indicates the value below which a given percentage of data falls
Minumum/ Maximum	$\min$ / $\max_{i=1}^{n}(x_i)$	Provide the lower/upper bounds of the data, namely the range of the values

Confidence Interval	Z
80%	1.282
85%	1.440
90%	1.645
95%	1.960
99%	2.576
99.5%	2.807
99.9%	3.291

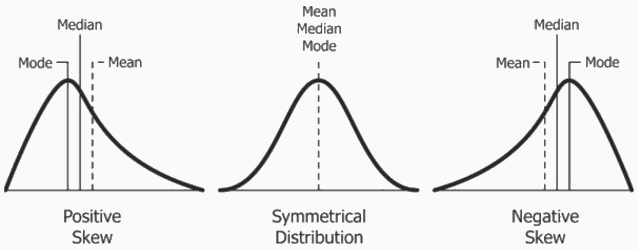
Some metrics assume a normal distribution $\rightarrow$ the arithmetic mean, median and mode are all equal

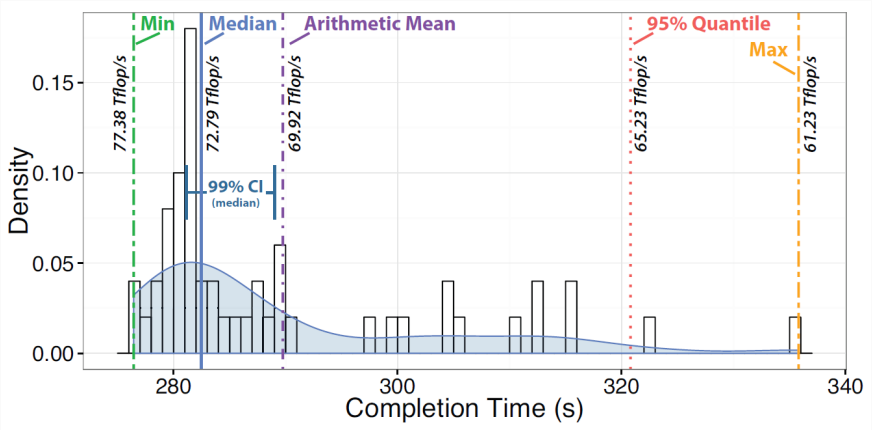
$$\frac{|\bar{x} - median|}{\max(\bar{x}, median)}$$

If the *relative difference between the mean and median* is larger than 1%, values are probably not normally distributed

Minimum/Maximum vs. Arithmetic mean. The minimum/maximum could be used to get the best outcome of an experiment, namely the measure with the least noise. On the other hand, the arithmetic mean considers all values and could better represent the behavior of the experiment.

If the *skewness* of the distribution is *symmetrical* (e.g. normal, binomial) then the arithmetic mean is a superior statistic, while the minimum/maximum could be useful in the opposite case (e.g. log-normal distribution)





- Benchmarking: minimum vs average
- Scientific Benchmarking of Parallel Computing Systems
- Benchmarking C++ Code

Profiling

Overview

A **code profiler** is a form of *dynamic program analysis* which aims at investigating the program behavior to find performance bottleneck. A profiler is crucial in saving time and effort during the development and optimization process of an application

Code profilers are generally based on the following methodologies:

- **Instrumentation** Instrumenting profilers insert special code at the beginning and end of each routine to record when the routine starts and when it exits. With this information, the profiler aims to measure the actual time taken by the routine on each call.
  `Problem:` The timer calls take some time themselves
- **Sampling** The operating system interrupts the CPU at regular intervals (time slices) to execute process switches. At that point, a sampling profiler will record the currently-executed instruction

gprof

gprof is a profiling program which collects and arranges timing statistics on a given program. It uses a hybrid of instrumentation and sampling programs to monitor *function calls*

Website: `sourceware.org/binutils/docs/gprof/`

Usage:

- Code Instrumentation

```
$ g++ -pg [flags] <source_files>
```

Important: `-pg` is required also for linking and it is not supported by `clang`

- Run the program (it produces the file `gmon.out`)
- Run `gprof` on `gmon.out`

```
$ gprof <executable> gmon.out
```

- Inspect `gprof` output

gprof output

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 84.04      0.85     0.85        1   848.84   848.84  yet_another_test
  6.00      0.91     0.06        1    60.63   909.47  test
  1.00      0.92     0.01        1    10.11    10.11  some_other_test
  0.00      0.92     0.00        1     0.00   848.84  another_test
```

gprof can be also used for showing the call graph statistics

```
$ gprof -q <executable> gmon.out
```

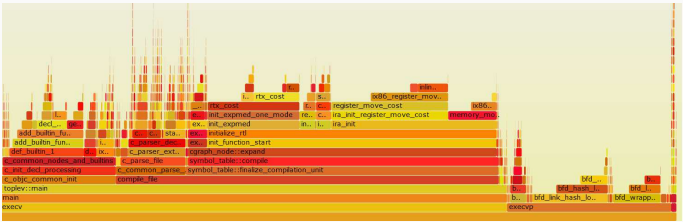
uftrace

The uftrace tool is to trace and analyze execution of a program written in C/C++

Website: github.com/namhyung/uftrace

```
$ gcc -pg <program>.cpp
$ uftrace record <executable>
$ uftrace replay
```

Flame graph output in html and svg



callgrind

callgrind is a profiling tool that records the call history among functions in a program's run as a call-graph. By default, the collected data consists of the number of instructions executed

Website: `valgrind.org/docs/manual/cl-manual.html`

Usage:

- Profile the application with `callgrind`

```
$ valgrind --tool callgrind <executable> <args>
```

- Inspect `callgrind.out.XXX` file, where `XXX` will be the process identifier

cachegrind

cachegrind simulates how your program interacts with a machine's cache hierarchy and (optionally) branch predictor

Website: `valgrind.org/docs/manual/cg-manual.html`

Usage:

- Profile the application with `cachegrind`

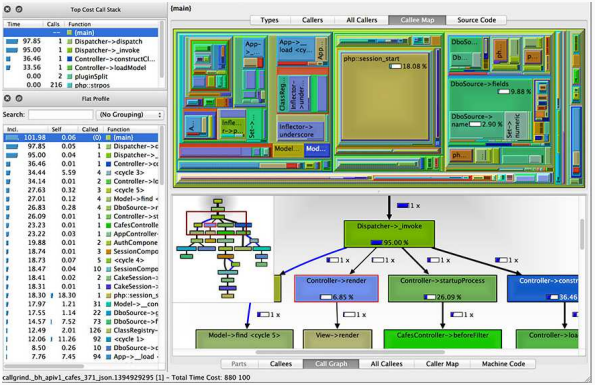
```
$ valgrind --tool cachegrind --branch-sim=yes <executable> <args>
```

- Inspect the output (cache misses and rate)
  - `I1` L1 instruction cache
  - `D1` L1 data cache
  - `LL` Last level cache

kcachegrind and qcachegrindwin (View)

KCachegrind (linux) and **Qcachegrind** (windows) provide a graphical interface for browsing the performance results of callgraph

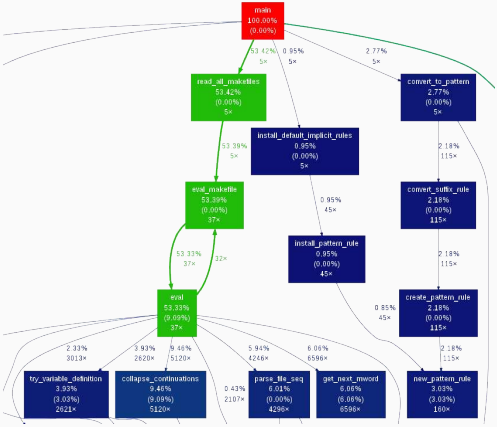
- kcachegrind.sourceforge.net/html/Home.html
- sourceforge.net/projects/qcachegrindwin



gprof2dot (View)

gprof2dot is a Python script to convert the output from many profilers into a dot graph

Website: `github.com/jrfonseca/gprof2dot`



Perf is performance monitoring and analysis tool for Linux. It uses statistical profiling, where it polls the program and sees what function is working

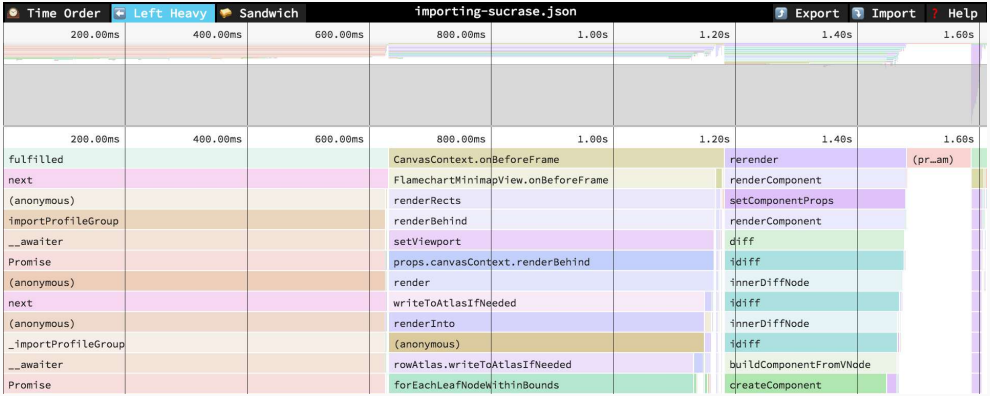
Website: perf.wiki.kernel.org/index.php/Main_Page

```
$ perf record -g <executable> <args> // or
$ perf record --call-graph dwarf <executable>
$ perf report // or
$ perf report -g graph --no-children
```

```
# Overhead  Command      Shared Object                  Symbol
# ........  .......  .................  ......................
#
    86.79%       dd  [kernel.kallsyms]  [k] common_file_perm
    11.41%       dd  perf_3.2.0-23      [.] memcpy
     1.80%       dd  [kernel.kallsyms]  [k] native_write_msr_safe
```

Data collected by **perf** can be visualized by using flame graphs, see:

`Speedscope: visualize what your program is doing and where it is spending time`



Other Profilers

Free profiler:

- `Hotspot`

Proprietary profiler:

- `Intel VTune`
- `AMD CodeAnalyst`

Parallel Computing

Concurrency vs. Parallelism

Concurrency

A system is said to be **concurrent** if it can support two or more actions in progress at the same time. Multiple processing units work on different tasks independently

Parallelism

A system is said to be **parallel** if it can support two or more actions executing simultaneously. Multiple processing units work on the same problem and their interaction can effect the final result

Note: parallel computation requires rethinking original sequential algorithms (e.g. avoid race conditions)

Performance Scaling

Strong Scaling

The **strong scaling** defined how the compute time decreases increasing the number of processors for a fixed total problem size

Weak Scaling

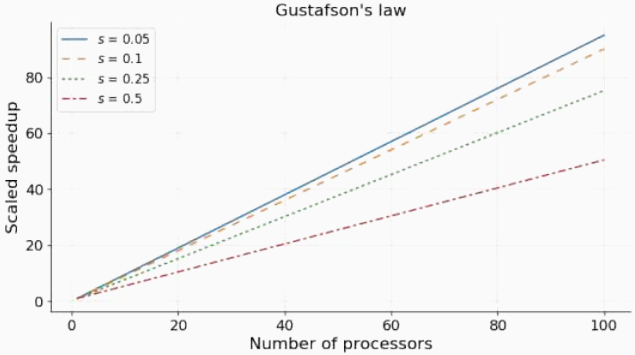
The **weak scaling** defined how the compute time decrease increasing the number of processors for a fixed total problem size per processor

Strong scaling is hard to achieve because of computation units communication. *Strong scaling* is in contrast to the Amdahl's Law

Gustafson's Law

Increasing number of processor units allow solving larger problems in the same time (the computation time is constant)

Multiple problem instances can run concurrently with more computational resources



C++11 Threads (+ Parallel STL) free, multi-core CPUs

OpenMP free, directive-based, multi-core CPUs and GPUs (last versions)

OpenACC free, directive-based, multi-core CPUs and GPUs

Khronos OpenCL free, multi-core CPUs, GPUs, FPGA

Nvidia CUDA free, Nvidia GPUs

AMD ROCm free, AMD GPUs

HIP free, heterogeneous-compute Interface for AMD/Nvidia GPUs

Khronos SyCL free, abstraction layer for OpenCL, OpenMP, C/C++ libraries, multi-core CPUs and GPUs

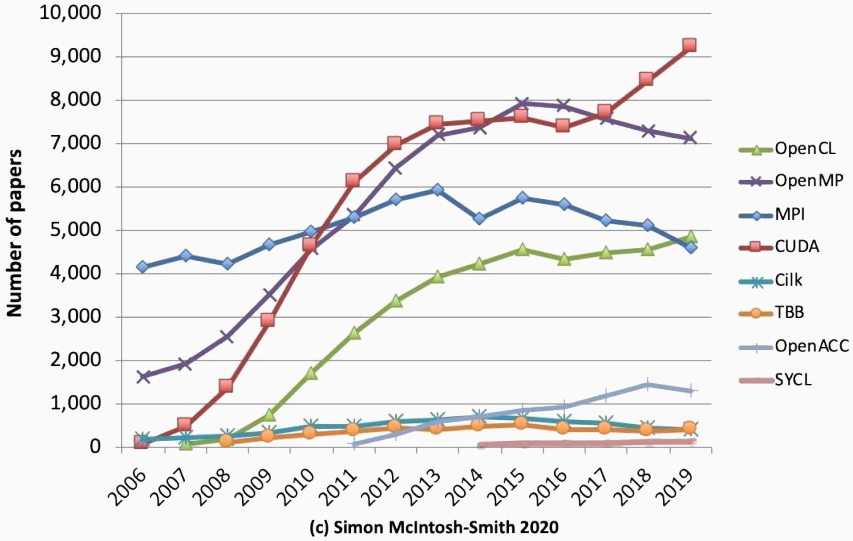
KoKKos (Sandia) free, abstraction layer for multi-core CPUs and GPUs

Raja (LLNL) free, abstraction layer for multi-core CPUs and GPUs

Intel TBB commercial, multi-core CPUs

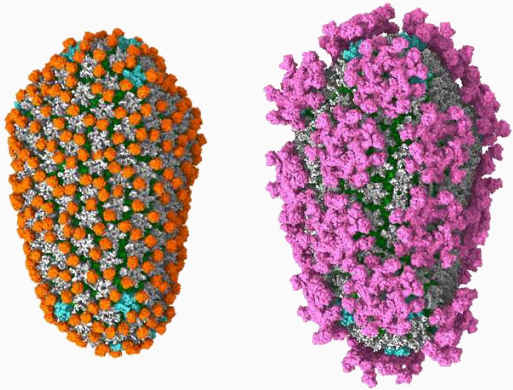
OneAPI free, Data Parallel C++ (DPC++) built upon C++ and SYCL, CPUs, GPUs, FPGA, accelerators

MPI free, de-facto standard for distributed system


10,000
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
Number of papers
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
OpenCL
OpenMP
MPI
CUDA
Cilk
TBB
OpenACC
SYCL
(c) Simon McIntosh-Smith 2020

A Nice Example

Accelerates computational chemistry simulations from 14 hours to 47 seconds with OpenACC on GPUs ($\sim 1,000x$ Speedup)



Modern C++ Programming

23. Software Design I [DRAFT]

Basic Concepts

Federico Busato

2024-03-29

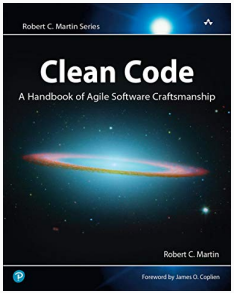
Table of Contents

Table of Contents

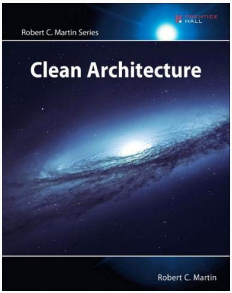
Table of Contents

Table of Contents

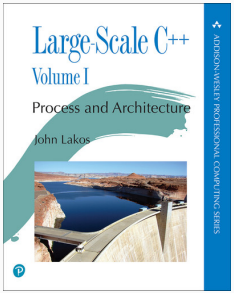
Books and References



Clean Code: A Handbook of Agile Software Craftsmanship
Robert C. Martin, 2008

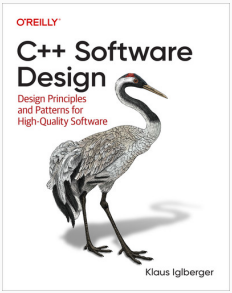


Clean Architecture
Robert C. Martin, 2017



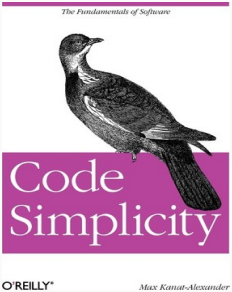
Large-Scale C++ Volume I: Process and Architecture

J. Lakos, 2021

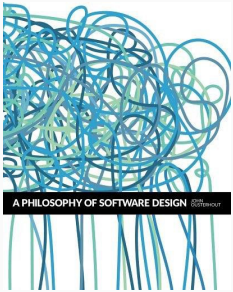


C++ Software Design

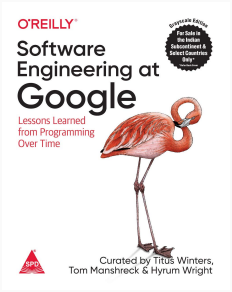
K. Iglberger, 2022



Code Simplicity
M. Kanat-Alexander, 2012



A Philosophy of Software Design (2nd)
J. Ousterhout, 2021



Software Engineering at Google: Lessons Learned from Programming over Time
T. Winters, 2020
`(download link)`

Basic Concepts

Abstraction, Interface, Module, and Class Invariant

An **abstraction** is the process of *generalizing relevant information and behavior* (semantics) from concrete details

An **interface** is a communication point that allows iterations between users and the system. It aims to *standardize* and *simplify* the use of programs

A **module** is a software component that provides a specific functionality. Common examples are classes, files, and libraries

> *"In modular programming, each **module** provides an **abstraction** in form of its **interface**"*
>
> – **John Ousterhout**, *A Philosophy of Software Design*

Quotes

*"Most modules have more users than developers, so it is better for the developers to suffer than the users... **it is more important for a module to have a simple interface than a simple implementation**"*

– **John Ousterhout**, *A Philosophy of Software Design*

*"The key to **designing abstractions** is to understand what is important, and to look for designs that **minimize the amount of information that is important**"*

– **John Ousterhout**, *A Philosophy of Software Design*

Class Invariant

A **class invariant** (or **type invariant**) is a *property* of an object which remains unchanged after operations or transformations. In other words, *a set of conditions that hold throughout its life*. A *class invariant* constrains the object state and describes its behavior

Software Design Principles

"Separation of concern" suggests to organize software in **modules**, each of which address a separate "concern" or functionality

Benefits of a modular design includes

- *Decrease cognitive load*. Small consistent parts are easier to understand than the whole system in its entirety
- *Help code maintainability*. Fewer or no dependencies allow to focus on smaller pieces of code, isolate potential bugs, and minimize the impact of changes
- *Independent development*

Modular design can be achieved both with *vertical* and *horizontal* organization, i.e. layers of abstractions or functionalities at the same level

*"The most fundamental problem in computer science is **problem decomposition**: how to take a complex problem and divide it up into pieces that can be solved independently"*

– **John Ousterhout**, *A Philosophy of Software Design*

"We want to design components that are self-contained: independent, and with a single, well-defined purpose"

– **Andy Hunt**, *The Pragmatic Programmer*

Low Coupling, High Cohesion

Cohesion *refers to the degree to which the elements inside a module belong together.* In other words, the code that changes together, stays together.

See also the *Single Responsibility Principle*

Coupling refers to the degree of interdependence between software modules. In other words, how a modification in one module affects changes in other modules

The **Low Coupling, High Cohesion** principle suggests to minimize dependencies and keep together code that is part of the same functionality

Encapsulation and Information Hiding

Encapsulation *refers to grouping together related data and methods that operate on the data*. It allows to present a consistent interface that is independent of its internal implementation

Encapsulation is usually associated with the concept of **information hiding** that prevents

- Exposing implementation details
- Violating *class invariant* maintained by the methods

It also provides freedom for the internal implementations

Encapsulation and information hiding are common paradigms to achieve *software modularity*

Problem Decomposition

> *"Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces"*
>
> – **James C. Dehnert and Alexander Stepanov**
> *Fundamentals of Generic Programming*

Code reuse

"Code reuse is the Holy Grail of Software Engineering"
– **Douglas Crockford**, *Developer of the JavaScript language*

Software Complexity

Technical Debt

"Technical debt is most often caused not so much be developers taking shortcuts, but rather by management who pushes velocity over quality, features over simplicity"

– **Grady Booch**, *UML/Design Pattern*

The SOLID Design Principles

Member Functions vs. Free Functions

Member Functions vs. Free Functions

> *"If you're writing a function that can be implemented as either a member or as a non-friend non-member, you should prefer to implement it as a non-member function. That decision increases class encapsulation. When you think encapsulation, you should think non-member functions"*
>
> – **Scott Meyers**, *Effective C++*

- https://workat.tech/machine-coding/tutorial/design-good-functions-classes-clean-code-86h68awn9c7q
- Prefer nonmember, nonfriends?
- Monoliths "Unstrung",
- How Non-Member Functions Improve Encapsulation
- C++ Core Guidelines - C.4: Make a function a member only if it needs direct access to the representation of a class
- Functions Want To Be Free, David Stone, CppNow15
- Free your functions!, *Klaus Iglberger*, Meeting C++ 2017

Member Functions

Functions that must be member (C++ standard):

- Constructors, destructor, e.g. `A()`, `~A()`
- Assignment operators, e.g. `operator=(const A&)`
- Subscript operators, `operator[]()`
- Arrow operators, `operator->()`
- Conversion operators, `operator B()`
- Function call operator, `operator()`
- Virtual functions, `virtual f()`

Member Functions

Functions strongly suggested being member:

- **Unary operators** because they don't interact with other entities
  - Member access operators: dereferencing `*a`, address-of `&a`
  - Increment, decrement operators: `a++` `--a`
- Any **method that preserves**
  - **const correctness**, e.g. pointer access
  - **object initialization state**, e.g. a variable that cannot be changed externally after initialization (invariant)

Functions suggested being member:

- In general, **compound operators** are expressed by updating private data members

Non-Member Functions

Functions that must be non-member (C++ standard):

- Stream extraction and insertion `<<`, `>>`

Functions that are strongly suggested being non-member:

- Binary operators to maintain symmetry, see "Implicit conversion and overloading"
- `Template functions` within a `class template`
  Otherwise, it requires an additional `template` keyword when calling the function (see *dependent typename*) → verbose, error-prone

---

`Effective C++ item 24: Declare Non-member Functions When Type Conversions Should Apply to All Parameters`

Non-Member Functions

More in general, member functions should be used only to **preserve the invariant properties** of a class and **cannot be efficiency implemented in terms of other public methods**

All other functions are **suggested** to be **free-functions**

Why Prefer Non-Member Functions

Encapsulation: *Non-member functions guarantee to preserve the class invariant* as they can only call public methods, protecting the class state by definition.

Non-member functions helps to keep the class smaller and simpler $\rightarrow$ easier to maintain and safer

Member functions induce **coupling** forcing the dependency from the `this` pointer.

Member functions can be split or organized in several other functions, worsening the problem. Such methods are forced to perform actions that are only specific to such class. On the contrary, non-member function favor generic code and can be potentially reused across the program

Why Prefer Non-Member Functions

Cohesion/Single Responsibility Principle Member functions can perform actions that are not strictly required by the class, bloating its semantics

Open-Close Principle *Non-member functions improve the flexibility and extensibility of classes by extending its functionality but without*

BLAS GEMM Case Study

BLAS GEMM

GEneralized **M**atrix-**M**atrix product API provided by **B**asic **L**inear **A**lgebra **S**ubroutine standard is one of the most used function in scientific computing and artifical intelligence

The API is defined in C as follow: $C = \alpha op(A) * op(B) + \beta C$

```
ErrorEnum sgemm(int m, int n, int k,
                OperationEnum opA,
                OperationEnum opB,
                float  alpha,
                float* a,
                int    lda,
                float* b,
                int    ldb,
                float  beta,
                float* c,
                int    ldc);
```

BLAS GEMM - Comprehension Problems

- `m`, `n`, `k` **describe the shapes of** `A`, `B`, `C` in a non-intuitive way. Except domain-expert, users prefer providing the number of rows and columns as matrix properties, not GEMM problem properties
- **Privatization of the return channel** for providing errors
- **Errors expressed with enumerators**. Need additional API to get a description of the error meaning
- **Domain-specific cryptic name**. e.g. `zgemm`: generalized matrix-matrix multiplication with double-precision complex type
- **The data type on which the function operates is encoded in the name itself** `zgemm` $\rightarrow$ any new combination of data types requires a new name.

- `A`, `B`, `C` matrices could have different types
- The compute type, namely the type of intermediate operations, could be different from the matrices. This is also known as *mixed-precision* computation
- Batched computation, namely having multiple input/output matrices, is not supported
- The API is **state-less** $\rightarrow$ preprocessing steps for optimization or additional properties (e.g. different algorithms) cannot be expressed
- Matrix sizes can be greater than `int` $(2^{31} - 1)$, specially on distributed systems
- Even if we perform computations with relative small matrices, the strides, e.g. `row * lda` could be larger than `int` $(2^{31} - 1)$

- `alpha/beta` could have a different type from matrix types
- `alpha/beta` are typically pointers on accelerators (e.g. GPU) to allow asynchronous computation
- The underline memory layout is implicit (column-major). Row-major and other layouts are not supported
- `C` is both input and output. It is more flexible to decouple `C` and add another parameter for the output `D`
- Doesn't have an *execution policy* which describes *where* (host, device) and *how* (sequential, parallel, vectorized, etc.)

- Doesn't have a *memory resource* which provides a mechanism to manage internal memory
- *Memory alignment* is known only at run-time
- It is not possible to optimize the execution with compile-time matrix sizes

Most of all these points have been addressed by the `std::linalg` *proposal*

Owning Objects and Views

Objects vs. View

Object

An **object** is a representation of a *concrete entity* as a *value in memory*

Resource-owning object

Resource-owning object refers to RAII paradigm which ties resources to object lifetime

example: `std::vector`, `std::string`

View

A **view** acts as a *non-owning reference* and does not manage the storage that it refers to. Lifetime management is up to the user

example: `std::span`, `std::mdspan`, `std::string_view`

Objects vs. View

- lack ownership
- short-lived
- generally appear only in function parameters
- generally cannot be stored in data structures
- generally cannot be returned safely from functions (no ownership semantics)

Objects vs. View

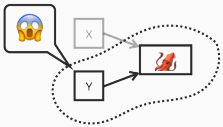
```cpp
#include <string>
#include <string_view>

std::string f() { return "abc"; }

void g(std::string_view sv) {}

std::string_view x = f(); // memory leak
g(f());                   // memory leak
```

Regular, Revisited, *Victor Ciura*, CppCon23

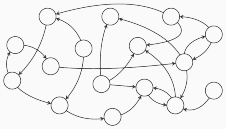
Value vs. Reference Semantic

Technical Debt: engineering cost: more coupled, more rigid, fragile (multiple references)

Spooky action: different references see an implicitly shared object. Modification to a reference affects the other ones



Incidental algorithms: emerges from a composition of locally defined behaviors and with no explicit encoding in the program. References are connection between dynamic objects



Visibility broken invariant: a modification to a reference can have a chain of actions that reflects to the original object, breaking the visibility of an action

Race conditions: spooky action between different threads

`Values - Safety, Regularity, Independence, and the Future of Programming`, *Dave Abrahams*, CppCon22

Surprise mutation: invisible coupling introduced by involuntary dependencies

```
void offset(int& x, const int& delta) { x += delta;}

int a = 3;
offset(a, a); // x=6, delta=6
offset(a, a); // x=12, delta=12
```

Unsafe operations mutation: A safe operation cannot cause undefined behavior

```
int a  = 3;
int b& = a;
a = b++;
```

~~see also, strict aliasing violation~~

Property Models: From Incidental Algorithms to Reusable Components, *Jarvi et al*, GPCE'08

Regularity: $x = x; x == y \rightarrow y == x; x == \text{copy}(x); x = y \iff x = \text{copy}(x)$

regular data type properties: copying, equality, hashing, comparison, assignment, serialization, differentiation

composition of value type is a value type

Independence: local and thread-safe

value semantic in C++

- pass-by-value gives callee an independent value
- a return value is independent in the caller
- a rvalue is independent

Global Variables

Global Variables

The Problems with Global Variables

Modern C++ Programming

24. Software Design II [DRAFT]

Design Patterns and Idioms

Federico Busato

2024-03-29

Table of Contents

C++ Idioms

Rule of Zero

The **Rule of Zero** is a rule of thumb for C++

> Utilize the *value semantics* of existing types to avoid having to implement *custom* copy and move operations

Note: many classes (such as `std` classes) manage resources themselves and should not implement copy/move constructor and assignment operator

```
class X {
public:
    X(...); // constructor
            // NO need to define copy/move semantic
private:
    std::vector<int>     v; // instead raw allocation
    std::unique_ptr<int> p; // instead raw allocation
};                          // see smart pointer
```

Rule of Three

The **Rule of Three** is a rule of thumb for C++(03)

If your class needs any of

- a copy constructor `X(const X&)`
- an assignment operator `X& operator=(const X&)`
- or a destructor `~X()`

defined explicitly, then it is likely to need all three of them

Some resources cannot or should not be copied. In this case, they should be declared as deleted

```
X(const X&) = delete
X& operator=(const X&) = delete
```

Rule of Five

The **Rule of Five** is a rule of thumb for C++11

If your class needs any of

- a copy constructor `X(const X&)`
- a move constructor `X(X&&)`
- an assignment operator `X& operator=(const X&)`
- an assignment operator `X& operator=(X&&)`
- or a destructor `~X()`

defined explicitly, then it is likely to need all five of them

Design Pattern

Singleton

Singleton is a software design pattern that restricts the instantiation of a class to one and only one object (a common application is for logging)

```
class Singleton {
public:
    static Singleton& get_instance() {  // note "static"
        static Singleton instance { ..init.. } ;
        return instance;   // destroyed at the end of the program
    }                      // initiliazed at first use

    Singleton(const& Singleton)      = delete;
    void operator=(const& Singleton) = delete;

    void f() {}
private:
    T _data;

    Singleton( ..args.. ) { ... }  // used in the initialization
}
```

PIMPL - Compilation Firewalls

Pointer to IMPLementation (PIMPL) idiom allows decoupling the interface from the implementation in a clear way

header.hpp

```
class A {
public:
    A();
    ~A();
    void f();
private:
    class Impl;  // forward declaration
    Impl* ptr;   // opaque pointer
};
```

NOTE: The class does not expose internal data members or methods

PIMPL - Implementation

source.cpp (Impl actual implementation)

```
class A::Impl { // could be a class with a complex logic
public:
    void internal_f() {
        ..do something..
    }
private:
    int   _data1;
    float _data2;
};

A::A()    : ptr{new Impl()} {}
A::~A()    { delete ptr; }
void A::f() { ptr->internal_f(); }
```

PIMPL - Advantages, Disadvantages

Advantages:

- ABI stability
- Hide private data members and methods
- Reduce compile type and dependencies

Disadvantages:

- Manual resource management
  - `Impl* ptr` can be replaced by `unique_ptr<impl> ptr` in C++11
- Performance: pointer indirection + dynamic memory
  - dynamic memory could be avoided by using a reserved space in the interface e.g. `uint8_t data[1024]`

PIMPL - Implementation Alternatives

What parts of the class should go into the `Impl` object?

- *Put all private and protected members into* `Impl`:
  Error prone. Inheritance is hard for opaque objects

- *Put all private members (but not functions) into* `Impl`:
  Good. Do we need to expose all functions?

- *Put everything into* `Impl`*, and write the public class itself as only the public interface, each implemented as a simple forwarding function*:
  Good

`https://herbsutter.com/gotw/_100/`

The **Curiously Recurring Template Pattern (CRTP)** is an idiom in which a class `X` derives from a class template instantiation using `X` itself as template argument

A common application is *static polymorphism*

```
template <class T>
struct Base {
    void my_method() {
        static_cast<T*>(this)->my_method_impl();
    }
};

class Derived : public Base<Derived> {
//  void my_method() is inherited
    void my_method_impl() { ... } // private method
};
```

```cpp
#include <iostream>
template <typename T>
struct Writer {
    void write(const char* str) {
        static_cast<const T*>(this)->write_impl(str);
    }
};
class CerrWriter : public Writer<CerrWriter> {
    void write_impl(const char* str) {  std::cerr << str;  }
};
class CoutWriter : public Writer<CoutWriter> {
    void write_impl(const char* str) {  std::cout << str;  }
};
CoutWriter x;
CerrWriter y;
x.write("abc");
y.write("abc");
```

```
template <typename T>
void f(Writer<T>& writer) {
    writer.write("abc);
}

CoutWriter x;
CerrWriter y;
f(x);
f(y);
```

Virtual functions cannot have template arguments, but they can be emulated by using the following pattern

```
class Base {
public:
    template<typename T>
    void method(T t) {
        v_method(t);    // call the actual implementation
    }
protected:
    virtual void v_method(int t)    = 0; // v_method is valid only
    virtual void v_method(double t) = 0; // for "int" and "double"
};
```

Actual implementations for derived class `A` and `B`

```
class AImpl : public Base {
protected:
    template<typename T>
    void t_method(T t) { // template "method()" implementation for A
        std::cout << "A " << t << std::endl;
    }
};

class BImpl : public Base {
protected:
    template<typename T>
    void t_method(T t) { // template "method()" implementation for B
        std::cout << "B " << t << std::endl;
    }
};
```

```cpp
template<class Impl>
class DerivedWrapper : public Impl {
private:
    void v_method(int t) override {
        Impl::t_method(t);
    }
    void v_method(double t) override {
        Impl::t_method(t);
    } // call the base method
};

using A = DerivedWrapper<AImpl>;
using B = DerivedWrapper<BImpl>;
```

```cpp
int main(int argc, char* argv[]) {
    A a;
    B b;
    Base* base = nullptr;

    base = &a;
    base->method(1);   // print "A 1"
    base->method(2.0); // print "A 2.0"

    base = &b;
    base->method(1);   // print "B 1"
    base->method(2.0); // print "B 2.0"
}
```

`method()` calls `v_method()` (pure virtual method of `Base`)
`v_method()` calls `t_method()` (actual implementation)