Modern C++ Programming

2. Basic Concepts I

Federico Busato

University of Verona, Dept. of Computer Science
2018, v1.0

Agenda

- **Before Start**
    - What compiler?
    - What editor/IDE?
    - How to compile?

- **Hello World**

- **I/O Stream**
    - cout/cin
    - Filestream
      (ifstream/ofstream)

- **C++ Primitive Types**
    - Built-in types
    - size_t, void, auto, nullptr
    - Conversion rules

- **Floating Point**
    - Floating point representation
    - Floating point issues
    - Floating point comparison
    - Overflow/Underflow

- **Strongly Typed Enumerators**

- **Math Operators**

- **Statement and Control Flow**
    - Loop
    - Range Loop
    - Undefined behavior
    - goto

What C++ compiler should I use?

Popular (free) compilers:

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

Suggested compiler: **Clang**

- Faster compiles, low memory use, and in general faster code (compared to GCC/MSVC). [compiler comparison link]
- Expressive diagnostics (examples and propose corrections)
- Strict C++ compliance. GCC/MSVC compatibility (inverse direction is not ensured)
- Includes many very useful tools: memory sanitizer, static code analyzed, automatic formatting, linter (clang-tidy), etc.
- Easy to install: releases.llvm.org

C++ compiler features support: en.cppreference.com/w/cpp/compiler_support

What editor/IDE compiler should I use?

Popular C++ IDE (Integrated Development Environment) and editors:

- **Microsoft Visual C++**
- **QT-Creator** (link). Fast (written in C++), simple
- **Clion** (link). (free for student). Most powerful IDE, but may be slow (written in java) and a lot of options may make it not intuitive
- **Atom** (link). Standalone editor oriented for programming. A lot of useful plugins and modular
- **Sublime Text editor** (link). Standalone editor oriented for programming. Faster than Atom, but less complete

Not suggested:

- `Notepad, Gedit, and related editors`. Lack of support for programming

How to compile?

Compile C++ programs:

```
g++  <program.cpp> -o program
```

Compile C++11 programs:

```
g++ -std=c++11 <program.cpp> -o program
```

- requires g++ version $\geq$ 4.8.1
- requires clang++ version $\geq$ 3.3

Compile C++14 programs:

```
clang++ -std=c++14 <program.cpp> -o program
```

- requires g++ version $\geq$ 5
- requires clang++ version $\geq$ 3.4

C code with `printf`:   C++ code with streams:

```
#include <stdio.h>

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

```
#include <iostream>

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

`printf` prints on standard output   `cout` : represent 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 we adopt implicit std namespace in the next
slides

I/O Stream

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 pass to I/O stream is known <u>statically</u> by the compiler. In contrast, `printf` uses "%" fields to figure out the types dynamically

- **Less error prone**: With IO Stream, there are no redundant "%" tokens that have to be consistent with the actual objects pass to I/O stream. Removing redundancy removes a class of errors

- **Extensible**: The C++ IO Stream mechanism allows new user-defined types to be pass 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 whitespace:

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

some usefull details about outout stream (I/O Stream manipulator):

- `flush` flushes the output stream `cout ≪ flush`;

- `endl` shortcut for `cout ≪ "\n" ≪ flush`;

- `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

```
#include <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)

`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

```cpp
#include <iostream>
using namespace std;

int main() {
    int a;
    cout << "Please enter an integer value:" << endl;
    cin >> a;

    int b;
    float c;
    cout << "Please enter an integer value followed "
        << "by a float value:" << endl;
    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

```
#include <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>
using namespace std;

int main() {
    ifstream fin("example.txt");
    string str;
    while (fin.good()) {
        getline(fin, str);
        cout << str << endl;
    }
    fin.close();
}
```

An alternative version with redirection:

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ifstream fin("example.txt");
    cout << fin.rdbuf();
    fin.close();
}
```

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:

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ifstream fin("example.txt");
    char c = fin.peek(); // c = '2'
    while (fin.good()) {
        int var;
        fin >> var;
        cout << var; // out: 2370445789
    }
    fin.seekg(4);
    c = fin.peek();  // c = '0'
    fin.close();
}
```

Check if a file is a **regular file** and can be read/written (it exists, it is not
a directory, it is not a device, you have read/write permissions, etc.)

```cpp
#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
    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;
}
```

Get the **file size** in bytes in a **portable** way:

```
long long int fileSize(const char* file_path) {
   ifstream fin(file_path);
   fin.seekg(0, ios::beg);
   std::istream::pos_type start_pos = fin.tellg();
   fin.seekg(0, ios::end);
   std::istream::pos_type end_pos = fin.tellg();
   return end_pos - start_pos;
}
```

C++ Primitive Types

Type	Size (bytes)	Range	Fixed width types
`bool`	1	true, false	
`char` [†]	1	-127 to 127	
`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*		
`long unsigned int`	4/8*		
`long long int`	8	-2^{63} to 2^{63}-1	int64_t
`long long unsigned int`	8	0 to 2^{64}-1	uint64_t
`float` (IEEE 754)	4	$\pm1.18 \times 10^{-38}$ to $\pm3.4 \times 10^{+38}$	
`double` (IEEE 754)	8	$\pm2.23 \times 10^{-308}$ to $\pm1.8 \times 10^{+308}$	

* 4 bytes instead 8 bytes in Win64 systems, [†] one-complement

C++ provides also **long double** (no IEEE-754) of size 8/12/16.

Signed types:

Signed Type	short name
signed int	int
signed long int	long
signed long long int	long long

For other details see:
en.cppreference.com/w/cpp/language/types
en.cppreference.com/w/cpp/types/integer

$C++$ provides fixed width integer types. They have the same size
on any architecture (#include <cstdint>)

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

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

```
int8_t var;
std::cin >> var;        // read '2'
std::cout << var;       // print '2'
std::cout << var * 2;   // print 100 !!
```

Builtin types suffix:

Type	short name	Suffix	example
unsigned int	unsigned	u	3u
long int	long	l	8l
long unsigned	long unsigned	ul	2ul
long long int	long long	ll	4ll
long long unsigned int	long long unsigned	ull	7ull
float		f	3.0f
double			3.0

Builtin types representation prefix:

Representation	Prefix	example
Binary C++14	0b	0b010101
Octal	0	0308
Hexadecimal	0x or 0X	0xFFA010

Definition (size_t)

size_t is a data type capable of storing the biggest representable value on the current architecture

- Defined in `<cstddef>`

- It is an <u>unsigned integer</u> type (of at least 16 bit)

- In common linux implementations it is 4-byte on 32-bit architectures and 8-byte on 64-bit architectures

- It is commonly used for array indexing and loop counting

Note: `void` is an incomplete type (not defined) without a values
In C `sizeof(void) == 1` (GCC), while in C++ `sizeof(void)`
does not compile!!

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

C++11 introduces the new keyword `nullptr` to represent null pointers

```cpp
int* p1 = NULL;   // ok, equal to int* p1 = 0
int* p2 = nullptr; // ok

int n1 = NULL;    // ok, we are assigning 0 to n1
// int n2 = nullptr;  // error! we are assigning a null pointer
//                    //        to an integer variable
// int* p2 = true ? 0 : nullptr; // incompatible types
```

Remember: `nullptr` is not a pointer, but an object of type `nullptr_t` → safer

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

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

`auto` keyword may be very useful for maintainability.

```cpp
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

Implicit type conversion rules (applied <u>in order</u>) :

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

(a) Floating point promotion

floating_type $\otimes$ integer_type $=$ floating_type

(b) Size promotion

small_type $\otimes$ large_type $=$ large_type

(c) Sign promotion

signed_type $\otimes$ unsigned_type $=$ unsigned_type

Common errors:

Integers are not floating points!

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

Implicit conversion can be expensive!

```
int b = 5;
int a = 3.5 * b;   // 3.5 is double --> useless overhead!!
//equal to: int a = (int) ( 3.5 * (double) b )
```

Integer type are more accurate than floating types for large numbers!!

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

float numbers are different from double numbers!

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

Detect overflow/underflow for <u>floating point</u> types is easy ($\pm$inf).

Detect overflow/underflow for <u>unsigned integral</u> types is **not trivial** !!

```cpp
bool isAddOverflow(unsigned a, unsigned b) {
    return (a + b) < a || (a + b) < b;
}
bool isMulOverflow(unsigned a, unsigned b) {
    unsigned x = a * b;
    return a != 0 && (x / a) != b;
}
```

Overflow/underflow for <u>signed integral</u> types is **not defined** !!

```cpp
#include <limits>
unsigned a = std::numeric_limits<unsigned>::max(); // maximum value
unsigned b = b + 1; // b = 0
int c     = std::numeric_limits<int>::max();       // maximum value
int d     = c + 1; // d can be any int value!!
```

Floating Point

Floating Point

In general, C/C++ adopt IEEE754 floating-point standard.

- <u>Single</u> precision (32-bit) (`float`)

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

- <u>Double</u> precision (64-bit) (`double`)

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

Check if the actual C++11 implementation adopts IEEE754 standard:

```cpp
#include <limits>
std::numeric_limits<float>::is_iec559;  // should return true
std::numeric_limits<double>::is_iec559; // should return true
```

Floating point (Exponent Bias)

Definition (Exponent Bias)

In IEEE 754 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 <u>lexicographic ordered</u>
- For a single-precision number, the exponent is stored in the range $[1, 254]$ (0 and 255 have special meanings), and is <u>biased</u> by subtracting 127 to get an exponent value in the range $[-126, +127]$

- Example

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

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

Definition (Normal number)

A **normal** number is a floating point number that can be
represented without leading zeros in its significant
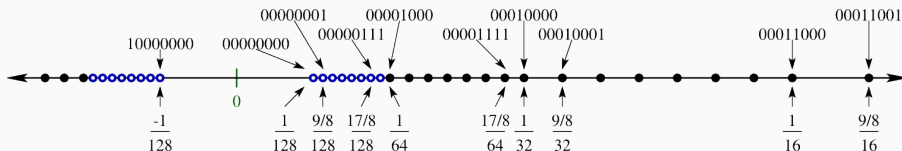
Definition (Denormal number)

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 subnormal

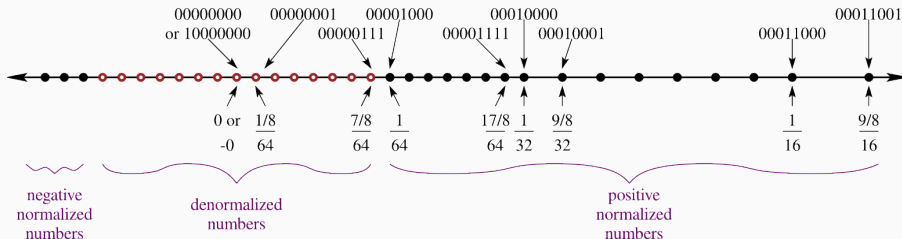If the exponent is all 0s, but the fraction is non-zero (else it would
be interpreted as zero), then the value is a denormal number

Check if a floating-point number is normal/denormal in C++11:

```cpp
#include <cmath>
isnormal(T value); // true if normal, false otherwise
```

Floating point online tool: www.h-schmidt.net/FloatConverter/IEEE754.html

Why denormal number make sense:

The problem: distance values from zero

cit: www.toves.org/books/float/

Floating point (special values)

- $\pm$ infinity

*	11111111	00000000000000000000000

- NaN (mantissa $\neq$ 0)

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

- ± 0

*	00000000	00000000000000000000000

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

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

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

*	00000001	00000000000000000000000

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

*	11111110	11111111111111111111111

Floating point issues

The floating point precision is finite!

```cpp
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
```
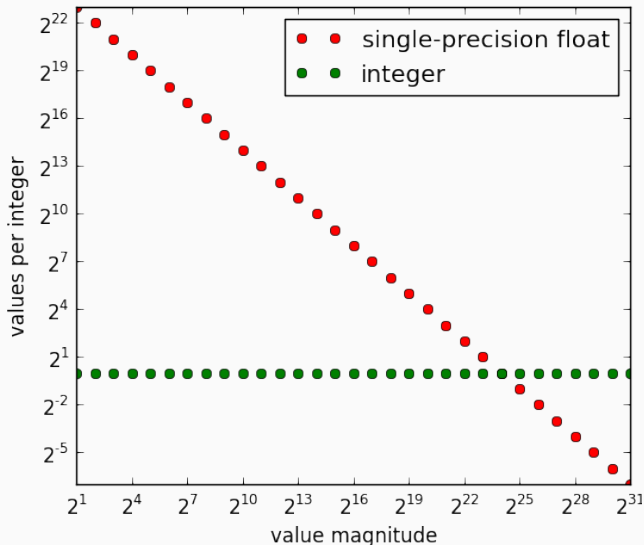
Floating point arithmetic is commutative, but not associative and not reflexive (see NaN) !!

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

Floating point type has special values:

```cpp
cout << 0 / 0;      // undefined behavior
cout << 0.0 / 0.0;  // print `nan'
cout << 5.0 / 0.0;  // print `inf'
cout << -5.0 / 0.0; // print `-inf'
cout << ((5.0 / 0.0) == ((5.0 / 0.0) + 9999999.0)) // print true
```

Intersection $\approx 16,777,217$

NaN properties

Definition (NaN)

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

Operations generating NaN:

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

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

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

Floating Point - Useful Functions

where T is a numeric type C++11

```cpp
#include <cmath>

bool isnormal(T value); // true if normal, false otherwise

bool isnan(T value) // returns true if value is nan, false otherwise

bool isinf(T value) // returns true if value is ±inf, false otherwise

bool isfinite(T value) // returns true if value is not nan or infinite,
                       // false otherwise

T ldexp(T x, p) // multiplies a number by 2 raised to a power.
                // returns x * 2^p

int ilogb(T value) // extracts exponent of the number
```

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 $\quad \frac{|a-b|}{b} < \varepsilon$

```cpp
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: $\quad \frac{|a-b|}{\max(|a|,|b|)} < \varepsilon$

```cpp
bool areFloatNearlyEqual(float a, float b) {
    const float epsilon = <user_defined>
    float abs_a = std::abs(a);
    float abs_b = std::abs(b);

    if (a == b) // a=0,b=0 and a = ±∞, b = ±∞
        return true;

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

References:

[1] floating-point-gui.de/errors/comparison

[2] www.cygnus-software.com/papers/comparingfloats

Floating Point (In)Accuracy

Definition (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 : $\varepsilon = 1.17549435 * 10^{-38}$

```cpp
#include <limits>
T std::numeric_limits<T>:: epsilon() // returns the  machine epsilon
```

Definition (Truncation error)

A number x that is **Truncated** (or *Chopped*) at the mth digit means that all $n - m$ digits after the nth digit are removed.

- Machine floating-point representation of x is denoted **fl(x)**

The relative error as a result of truncation is

$$\left| \frac{fl(x) - x}{x} \right| \le \frac{1}{2}\varepsilon \qquad fl(x) = x(1 + \delta) \qquad |\delta| \le \frac{1}{2}\varepsilon$$
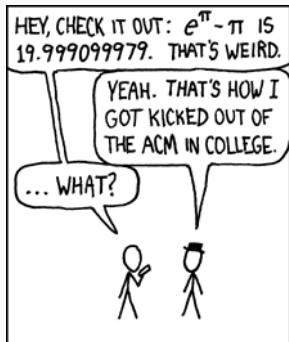
Minimize Error Propagation

- Prefer **multiplication/division** than addition/subtraction

- Scale by a **power of two** is safe

- Try to reorganize the computation to **keep near** numbers with the same scale (maybe sorting numbers)

- Consider to **put a zero** very small number (under a threshold). Common application: iterative algorithms

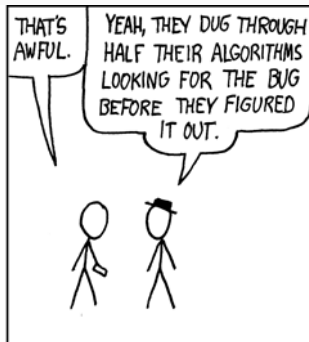- **Switch to log scale**. Multiplication becomes Add, and Division becomes Subtraction

Suggest reading:
D. Golberg, *"What Every Computer Scientist Should Know About Floating-Point Arithmetic*, 1991, link

Enumerators

Definition (Enumerator)

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

```
enum color_t { BLACK, BLUE, GREEN = 2 };

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 = BLUE;
fruit_t fruit = APPLE;
cout << (color == fruit); // generally true, but undefined !!
// and, most importantly, does the match between a color and
// a fruit makes any sense?
```

C++11 introduces the `enum class` (scoped enum) data type
that are not <u>implicitly</u> convertible to int

Type safe enumerator: `enum class`

Syntax: `<enum_class>::<enum_value>`

```cpp
enum class color_t { BLACK, BLUE, GREEN = 2 };
enum class fruit_t { APPLE, CHERRY };

color_t color = color_t::BLUE;
fruit_t fruit = fruit_t::APPLE;
// cout << (color == fruit); // compile error!!
//     we are trying to match colors with fruits
//     BUT, they are different things entirely

// int a = color_t::GREEN;  // compile error!!
```

- Strongly typed enumerators can be compared:

```
enum class Colors { RED = 1, GREEN = 2, BLUE = 3 };

cout << (Colors::RED < Colors::GREEN); // print true
```

- Strongly typed enumerators do not support other operations:

```
enum        WColors { RED = 1, GREEN = 2, BLUE = 3 };
enum class SColors { RED = 1, GREEN = 2, BLUE = 3 };

int v = RED + GREEN; // ok
// int v = SColors::RED + SColors::GREEN; // compile error!
```

- The size of `enum class` can be set:

```
#include <cstdint>
enum class Colors : int8_t { RED = 1, GREEN = 2, BLUE = 3 };
```

- Strongly typed enumerators can be converted:

```
int a = (int) color_t::GREEN; → ok
```

- Enum class objects should be always initialized:

```
enum class SColors { RED = 1, GREEN = 2, BLUE = 3 };

int main() {
    SColors my_color; // my_color maybe 0!!
}
```

Math Operators

Precedence	Operator	Description	Associativity
1	a++ a--	Suffix/postfix increment and decrement	Left-to-right
2	++a --a	Prefix increment and decrement	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	&&	Logical AND	Left-to-right
12	||	Logical OR	Left-to-right

In general:

- **Unary** operators have <u>higher</u> precedence than **binary operators**

- **Standard math operators** (+, *, etc.) have <u>higher</u> precedence than **comparison**, **bitwise**, and **logic** operators

- **Comparison** operators have <u>higher</u> precedence than **bitwise** and **logic operators**

- **Bitwise** operators have <u>higher</u> precedence than **logic** operators

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

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

Statements and Control Flow

Statements and Control Flow

- Assignment operations and control flow (special cases):

```cpp
int a;
int b = a = 3;   // (a = 3) return value 3
if (b = 4)       // it is not an error, but BAD programming
    ...
if (<true expression> || array[-1] == 0)
    ... // no error!! even though index is -1
        // left-to-right short-circuiting evaluation
```

- C++ allows "in loop" definitions:

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

- Jump statements:

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

- Infinite loop:

```cpp
for (;;)
    ...
```

- Range loop: C++11

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

char letters[] = "abcd";
for (auto c : letters)
    cout << c << " ";    // print: a b c d
```

- Ternary operator: `<cond> ? <expression1> : <expression2>`
$<expression1>$ and $<expression2>$ must return a value of the same type

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

Expressions with undefined (implementation-defined) behavior:

```cpp
int i = 0;
i = ++i + 2;        // undefined behavior until C++11,
                    // otherwise i = 3
i = 0;
i = i++ + 2;        // undefined behavior until C++17,
                    // modern compilers (clang, gcc): i = 3

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

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

n = ++i + i;        // undefined behavior
```

When it is useful:

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

become:

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