Modern C++ Programming

5. Basic Concepts IV

Memory Concepts

Federico Busato

2024-02-09

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

higher memory addresses 0x00FFFFFF	**Stack** ↓	**stack memory**	`int data[10]`
	↑ **Heap**	**dynamic memory**	`new int[10]` `malloc(40)`
	BSS and Data Segment .bss/.data	**Static/Global data**	`int data[10]` (global scope)
lower memory addresses 0x00FF0000	**Code** .text		

Data and BSS Segment

```c
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

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 and 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 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
```

Uniform Initialization

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

```
int b1{2};          // direct list (or value) initialization
int b2{};           // direct list (or value) initialization (zero-initialization)

int b3 = int{};     // copy initialization (zero-initialization)
int b4 = int{4};    // copy initialization

int b5 = {};        // copy list initialization (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
```

Fixed-Size Array 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
```

```
struct S {
    unsigned x;
    unsigned y;
};

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

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

Structure Initialization - C++11


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

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

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

C++20 introduces *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

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
```

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 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`[†] is not allowed for 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
```

$1 + 1 \neq 2$: Pointer Arithmetic

`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	

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 used 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`
- **String literal** of type `const char[]`, e.g `"literal"`
- `nullptr`
- User-defined literals

const keyword

The const keyword indicates objects never changing value after their initialization (they must be initialized when declared)

const variables are evaluated at compile-time value if the right expression is also evaluated at compile-time

```
int size = 3;
int A[size] = {1, 2, 3}; // Technically possible (size is dynamic)
                         // But NOT approved by the C++ standard
const int SIZE = 3;
// SIZE = 4;             // compile error (SIZE is const)
int B[SIZE] = {1, 2, 3}; // ok

const int size2 = size;
int C[size2] = {1, 2, 3}; // BAD programming!! size2 is not const
// (some compilers allow variable size stack array -> dangerous!!)
```

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

```
void f1(const int* array) {} // the values of the array cannot
                             //  be  modified
void f2(int* array) {}

int*       ptr  = new int[3];
const int* cptr = new int[3];
f1(ptr);        // ok
f2(ptr);        // ok
f1(cptr);       // ok
// f2(cptr);    // compile error
```

```
void g(const int) {  // pass-by-value combined with 'const'
    ...              // note: it is not useful because the value
}                    //       is copied
```

`const` Keyword and Pointers

- `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) {
//  ptr[0] = 0;       // not allowed: pointer to const objects
    ptr    = nullptr; // allowed
}
void f3(const_ptr_t ptr) { // same as before
//  ptr[0] = 0;       // not allowed: pointer to const objects
    ptr    = nullptr; // allowed
}
void f2(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 that the expressions can be evaluated at compile time

- `const` guarantees the value of a variable to be fixed overall the execution of the program
- `constexpr` implies `const`
- `constexpr` helps for performance and memory usage
- `constexpr` could potentially impact on 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

- Cannot contain run-time functions, namely non-`constexpr` functions
- 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
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 `static` variables
- cannot contain `assert()` until C++14
- must not be `virtual` until C++20
- undefined behavior code is not allowed, e.g. `reinterpret_cast`, unsafe usage of `union`, signed integer overflow, etc.

`constexpr` *non-static member functions* of run-time objects cannot be used even if all constrains are respected.

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

```
struct A {
    constexpr int        f() const { return 3; }
    static constexpr int g()       { return 4; }
};

A a1;
// constexpr int x = a1.f(); // compile error
constexpr    int y = a1.g(); // ok, also 'A::g()' is fine

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

`consteval` Keyword

`consteval` (C++20)

`consteval`, or *immediate functions*, guarantees compile-time evaluation of a function. A non-constant value always produces a compilation 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 non-constant value always produces a compilation error

- The value of a variable can change during the execution
- `const constinit` does not imply `constexpr`, while the opposite is true
- `constexpr` requires compile-time evaluation during his entire lifetime

```
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` C++17 feature allows to *conditionally* compile code based on a *compile-time* value

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

```cpp
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):

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

Other example: `C++17 Compile-time Quick-Sort`

`if constexpr` Pitfalls

`if constexpr` works only 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

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

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

int x = f(3); // x = 0

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

`std::is_constant_evaluated()` has two problems that C++23 `if consteval` 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())  // if consteval works fine
        return g(n);
    return 4;
}

// f(3);   compiler error
```

(2) `if constexpr (std::is_constant_evaluated())` is a bug as 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;
}
```

`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 - Pointer


```
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)
```

sizeof - struct


```
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)
```

```cpp
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

//-------------------------------------------------------------
// SPECIAL CASES
struct A {};
sizeof(A);       // 1 : sizeof never return 0

A array1[10];
sizeof(array1); // 1 : array of empty structures

int array2[0];  // only gcc
sizeof(array2); // 0 : special case
```

`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`)