Modern C++ Programming

4. Basic Concepts III

- Memory Management

Federico Busato

University of Verona, Dept. of Computer Science

2021, v3.09

Heap and Stack

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		

Stack and Heap Memory Overview

	Stack	**Heap**
Memory Organization	Contiguous	(block) Fragmented
Max size	Small (~8MB)	Whole system memory
If exceed	Program crash	Exception or `nullptr`
Allocation	Compile-time	Run-time
Locality	High	Low
Thread View	Each thread in a program has its own stack	Shared among threads

Stack Memory

A local variable is either in stack memory or CPU registers

```
int x = 3; // not on stack

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)
    int* ptr = new int; // variable "ptr" is on stack
}
```

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

It is $\approx 8MB$ on linux by default

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; // Illegal memory access!! ☠
}
```

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 allocate and free *memory blocks* (expressed in bytes)

Example:

```
int* array = new int[10];  // C: (int*) malloc(10 * sizeof(int))
delete[] array;            // C: free(array)
```

`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*
- **Allocated bytes**: The size of the allocated memory is calculated by the compiler for `new`, while the user must take care of manually calculate the size for `malloc()`
- **Initialization**: `new` can be used to initialize an object or a set of objects

Dynamic 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 and zero-initialize N elements**

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

- **Allocate N structures**

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

Dynamic 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;
```

Dynamic Memory Notes

Fundamental rules:

- Each object allocated with `new` must be deallocated with `delete`
- Each object allocated with `new[]` must be deallocated with `delete[]`

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

`delete` and `delete[]` applied to `NULL`/ `nullptr` pointers do not produce errors (same as `free`)

Memory Leak

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

Problems:

- Illegal memory accesses → segmentation fault
- Undefined values → segmentation fault
- Additional memory consumption

```cpp
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

2D Memory Allocation

Easy on stack:

```
int A[3][4];
```

Dynamic Memory 2D allocation/free:

```
int** A = new int*[3];
for (int i = 0; i < 3; i++)
    A[i] = new int[4];

for (int i = 0; i < 3; i++)
    delete[] A[i];
delete[] A;
```

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

```
auto A = new int[3][4];     // allocate 3 objects of size 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
```

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 ≈ 1GB in general) but slower

Initialization

Variable Initialization

C++03:

```
int a1;           // default initialization (undefined value)
int a2(2);        // direct (or value) initialization
int a3 = 2;       // copy initialization
int a4 = 2u;      // copy initialization (implicit conversion)

int();            // direct (or value) initialization (zero-init)
int(2)            // direct (or value) initialization
int a5 = int(2);  // copy initialization
int a6 = int();   // copy initialization (zero-initialization)

int a7 = {2};     // copy list initialization
// int a8();      // a8 is a function
```

Uniform Initialization

C++11 provides the **Uniform Initialization** syntax, also called *brace-initialization* or *braced-init-list*, to initialize 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 = {};       // copy list initialization (zero-initialization)

int{};             // direct list (or value) initialization (zero-init)
int{4};            // direct list (or value) initialization
int b4 = int{};    // copy initialization
int b5 = int{4};   // copy initialization
```

Brace Initialization Advantages

The C++11 **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
```

```cpp
struct S {
    unsigned x;
    unsigned y;
};
//-------------------------------------------------------------

// C++03
S s1;            // default initialization (undefined values)
S s2 = {};       // copy list initialization (zero-init)
S s3 = {1, 2};   // copy list initialization

// C++11
S s4{};          // direct list (or value) initialization (zero-init)
S s5{1, 2};      // direct list (or value) initialization
// S s6{1, -2}; // compile error
```

Non-Static Data Member Initialization (NSDMI):

```
struct S {
    unsigned x = 3; // equal initialization
    unsigned y = 2; // equal initialization
};                  // also functions are allowed
struct S1 {
    unsigned x {3}; // brace initialization
    unsigned y {2}; // 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

S f() {
    return {3, 2};
}
```

Stack 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 of D are initialized to 0
int  f[3] = {};  // all values of E are initialized to 0 (C++11)
int  g[3] {};    // all values of F 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
```

Dynamic Initialization

C++03:

```
int* a1 = new int;       // undefined
int* a2 = new int();     // zero-initialization
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
// int* a6 = new int[4](3); // not valid
```

C++11:

```
int* b1 = new int[4]{};      // allocate 4 elements zero-initialized
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 refereed to the pointer

Subscript Operator []

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

Deferencing:

```
int* ptr1 = new int;
*ptr1     = 4;     // deferencing (assignment)
int a     = *ptr1; // deferencing (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	

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
}
```

Solution:

```
int main() {
    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
- A `void*` can be implicitly converted to another pointer
- 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 initilization
//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
```

see also www3.ntu.edu.sg/home/

Reference and `struct`

- 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 = 3;
};

A a;
A* ptr = &a; // pointer
ptr->x;      // arrow syntax

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

`const, constexpr, consteval, constinit`

`const` Keyword

`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!!)
```

Constness rules:

- `int* → const int*`
- `const int*` $\not\rightarrow$ `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 refereed 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 refereed by the pointer cannot be modified
- `int *const` const pointer to `int`
  - The value of the pointer cannot be modified
  - The elements refereed by the pointer can be modified
- `const int *const` const pointer to `const int`
  - The value of the pointer cannot be modified
  - The elements refereed by the pointer cannot be modified

Note: `const int*` is equal to `int const*`

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!!
    ptr[0] = 0;       // allowed
//  ptr    = nullptr; // not allowed: const pointer to
}                     // modifiable objects
```

`constexpr` (variable)

C++11 `constexpr` variables are evaluated at compile-time

- `const` guarantees the value of a variable to be fixed overall the execution of the program
- `constexpr` tells the compiler that the expression results is at compile-time. `constexpr` value implies `const`

`constexpr` (function)

C++11 `constexpr` guarantees compile-time evaluation of a function as long as **all** its arguments are constant

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

`constexpr` Keyword

```cpp
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
```

```cpp
constexpr int square(int value) {
    return value * value;
}

square(4); // compile-time evaluation

int a = 4; // "a" is dynamic
square(a); // run-time evaluation
```

consteval Keyword

consteval

C++20 consteval, or *immediate functions*, guarantees compile-time evaluation of a function. A non-constant value 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 produces a compilation error

- The value of the 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
```

C++17 introduces `if constexpr` feature which allows *conditionally* compiling code based on a *compile-time* value

It is an `if` statement where the branch is chosen at compile-time (similarly to the `#if` preprocessor)

```
void f() {
    if constexpr (true)
        cout << "compile!";
    else
        cout << "error!"; // never compiled
}
```

`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
```

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

std::is_constant_evaluated

C++20 provides `std::is_constant_evaluated()` utility for evaluating 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
```

Explicit Type Conversion

Old style cast `(type) value`

New style cast:

- `static_cast` does compile-time (not run-time) checking of the types involved In many situations, this can make it the safest type of cast, as it provides the least room for accidental/unsafe conversions between various 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 instructions

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 |\uline{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*

```
// slow without optimizations. The branch breaks the 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?
- Type Punning In C++17

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(incomplete type)` produces compile error, e.g. `void`
- `sizeof(bitfield member)` produces compile error
- `sizeof(anything)` never returns 0, except for array of size 0
- `sizeof(char)` always returns 1
- When applied to structures, it also takes into account padding
- When applied to a reference, the result is the size of the referenced type

```
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); // print sizeof(int) * 10 = 40 bytes
cout << sizeof(array2); // print sizeof(int*) = 8 bytes
f(array1);              // print 8 bytes (64-bit OS)
```

```c
struct A { // "A" is aligned to the largest type: int (4 bytes)
    int  x;
    char y; // offset 4
};
sizeof(A); // 8 bytes : 4 + 1 (+ 3 padding)

struct B {
    int   x; // offset 0
    char  y; // offset 4
    short z; // offset 6 -> 2 bytes aligned
};
sizeof(B); // 8 bytes : 4 + 1 (+ 1 padding) + 2

struct C {
    short z; // offset 0 -> 2 bytes aligned
    int   x; // offset 4 -> 4 bytes aligned
    char  y; // offset 8
};
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 (except for arrays)

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

int array2[0];
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`)