Modern C++ Programming

3. Basic Concepts II

Federico Busato

University of Verona, Dept. of Computer Science

2018, v1.5

Agenda

- **Memory Management: Heap and Stack**
  - Heap allocation and memory leak
  - Stack memory
  - Stack 2D allocation
  - Initialization
  - Data/Bss memory segment
- **Storage Class Specifiers**
- **Pointers and References**
  - Pointers
  - Void Pointer
  - Address-of Operator
  - Pointer Arithmetic
  - Reference
- `sizeof` **Operator**
- `const`, `constexpr`
- **Explicit Type Conversion**
- **Declaration and Definition**
- **Functions**
  - Call-by-value/pointer/reference
  - `inline`
  - Default parameters
  - Overloading
- **Preprocessing**
  - Macro
  - Pragma

Memory Management: Heap and Stack

Process Address Space

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

Dynamic Heap 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**

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

Dynamic Heap 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;
```

Fundamental rules:

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

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
- Undefined values
- Additional memory consumption

```
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

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
}
```

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
}
```

Stack Allocation

Unless it is allocated in heap memory (i.e. `new`), then it is either in stack memory or CPU registers

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

```
int* wrongFunction() {
    int A[3] = {1, 2, 3};
    return A;
}

int main() {
   int* ptr = wrongFunction();
   cout << ptr[0]; // Illegal memory access!!
}
```

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

```
int* ptr[10];   // array of ten integer pointers
                // read as (int*) ptr[10]
int (*ptr)[10]; // pointer to an array of ten integers
                // equal to:
                //      int a[10];
                //      int* ptr = a;
```

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

```c
int data[] = {1, 2, 3, 4};  // 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) are larger than stack memory (max $\approx$ 1GB in general) but slower

Initialization

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  C[3] = {1, 2};    // C[2] = 0 -> default value

int  D[4] = {0}; // all values of D are initialized to 0
int  E[3] = {};  // all values of E are initialized to 0 (C++11)
```

Two dimensions:

```
// int F[][]    = ...; // compile error!!
// int G[2][]   = ...; // compile error!!
int G[][2]   = { {1,2}, {3,4}, {5,6} }; // ok
int H[2][2] = { 1, 2, 3, 4 }; // ok
```

Default Initialization

Rules:

- An object with **dynamic** storage duration (heap) has indeterminate value
- An object whose initializer is an **empty set of parentheses** is zero or default initialized

Initialization

```
int a1;                  // indeterminate
int* a2 = new int;       // indeterminate
int* a3 = new int();     // indeterminate
int* a4 = new int(4);    // allocate a single value equal to 4!!

int* b1 = new int[4]();       // allocate 4 elements zero-initiliazed
int* b2 = new int[4]{};       // indeterminate
int* b3 = new int[4]{1, 2};   // set first, second, indeterminate
                              // other values

int c1(4);               // c1 = 4;
int c2 = int();          // zero-initiliazed
int c4 { 0 };            // zero-initiliazed
int c5 = { 0 };          // zero-initiliazed
int c6 {};               // zero-initiliazed

// int d3();              // d3 is a function
```

Pointers and References

Pointers and Pointer Dereferencing

Pointer

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

Pointer Dereferencing

Pointer **dereferencing** means obtaining the value stored in at the location refereed to the pointer

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

Common error:

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

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

- A `void*` can be assigned to another `void*`
- `void*` can be compared for equality and inequality
- A `void*` can be explicitly converted to another type
- Other operations would be unsafe because the compiler cannot know what kind of object is really pointed to. Consequently, other operations result in compile-time errors

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

int array[] = { 2, 3, 4 };
void* ptr = array;
cout << *array;            // print 2
// cout << *ptr;           // compile error!!
cout << *((int*) ptr);     // print 2
// void* ptr2 = ptr + 2; // compile error!!
```

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 = ...`

```
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
```

$1+1 \neq 2$: Pointer Arithmetic

Pointer syntax:

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

Pointer arithmetic rule:

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

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

Example:

```
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!!
```

`char arr[3] = "abc"`

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

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

value	address	
4	0x0	←arr[0]
	0x1	
	0x2	
	0x3	
5	4	←arr[1]
	0x5	
	0x6	
	0x7	
	0x8	←arr[2]

Reference

A variable **reference** 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) but also they take space on the stack
- References are internally implemented as *pointer*, but the compiler treats them in a very different way

References are safer then pointers:

- References **cannot have NULL** value. You must always be able to assume that a reference is connected to a legitimate piece of 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& d;   // reference. compile error!! no initilization
int  c = 2;
int& e = c; // reference. ok valid initialization
e++;        // increment
cout << c;  // print 3
```

```
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'

int& c = a;  // reference
c++;         // change the value of 'a'
```

Reference (Function Arguments)

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

References can be use to indicate fixed size arrays:

```
f(int (&array)[3]) {} // accepts only arrays of size 3
                      // f(int array[]) accepts any size
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
```

Reference (Arrays)

```
int A[4];
int (&B)[4] = A;      // ok
int C[10][3];
int (&D)[10][3] = C; // ok

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;    // complie error!!
int (*f)[4] = c;          // ok
```

Reference:

[1] www3.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html

Reference and struct

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

```
#include <iostream>
struct A {
   int x = 3;
};

int main() {
   A obj;

   A* p = &obj;  // pointer
   p->x;         // arrow syntax

   A& ref = obj; // reference
   std::cout << obj.x;  // dot syntax
   std::cout << ref.x;  // dot syntax
}
```

`sizeof` Operator

`sizeof` Built-in operator 1/3

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
- `sizeof(bitfield)` 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` and memory allocation:

```
int A[10];
int* B = new int[10];
cout << sizeof(A); // print sizeof(int) * 10 = 40
cout << sizeof(B); // print sizeof(int*) = 8  (64-bit)
```

```
sizeof(int);    // 4
sizeof(int*);   // 8 in a 64-bit OS
sizeof(void*)   // 8 in a 64-bit OS
sizeof(size_t)  // 8 in a 64-bit OS

char  a;
char& b = a;
sizeof(&a);    // 8 in a 64-bit OS (pointer)
sizeof(b);     // 1 sizeof(char)

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

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

int array2[0];
sizeof(array2); // 0
```

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

struct C : B { // C extends B
    short z;
};

sizeof(B);      // 8 : 4 + 1 (+ 3) (padding)
sizeof(C);      // 12 : sizeof(B) + 2 (+ 2) (padding)

int array[4]
sizeof(array)  // 16: 4 elements of 4 bytes
sizeof(array) / sizeof(int);  // 4 elements
```

`const` and `constexpr`

`const` keyword

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

Compile-time value if the right expression is 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!!
int B[SIZE] = {1, 2, 3}; // ok

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

Constness rules:

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

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

int f2(int* array) {}

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

`constexpr`

C++11/C++14 guarantees compile-time evaluation of an expression as long as **all** its arguments are constant

- `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`***

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

```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 c1 = v1;    // ok
// constexpr c2 = v3; // compile error!!
```

```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
```

`if constexpr`

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)
        std::cout << "compile!";
    else
        THIS STRING NEVER COMPILE // 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
```

Explicit Type Conversion

Cast

Old style cast `(type) value`

C++11 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.
- `reinterpret_cast`
  `reinterpret_cast<T*>(v)` equal to `(T*) v`
  `reinterpret_cast<T&>(v)` equal to `*((T*) &v)`
- `const_cast` may be used to cast away (remove) constness or volatility.

Type punning

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

```
bool is_negative(float x) {
        return x < 0.0;
}
bool is_negative(float x) {
        unsigned int* ui = (unsigned int *) &x; // gcc warning:
        return (*ui) & 0x80000000;              // -Wstrict-aliasing
}
```

https://blog.qt.io/blog/2011/06/10/type-punning-and-strict-aliasing/

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
```

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
int a[3][4];   // array reshaping example
int (&b)[2][6] = reinterpret_cast<int (&)[2][6]>(a);
int (*c)[6]    = reinterpret_cast<int (*)[6]>(a);
```

Narrowing Conversion

C++11 provides protection against **narrowing**, i.e. assigning a numeric value to a numeric type not capable of holding that value

```
int main() {
    int a1 = 36.6;          // ok
//  int a2 = { 36.6 };      // compile error!!
    int a3 { 36.6 };        // ok!! (constructor)

    float b1 = 36.6;        // ok
//  float b2 = { 36.6 };    // compile error!!
    int a3 { 36.6 };        // ok!! (constructor)

    char c1 = 512;          // ok
//  char c2 = { 512 };      // compile error!!
    char c3 = { 512 };      // ok!! (constructor)
}
```

Declaration and Definition

Declaration/Definition

Declaration/Prototype

A **declaration** (or prototype) of an entity is an identifier describing its type

A declaration is what the compiler and the linker needs to accept references to that identifier

Definition/Implementation

An entity **definition** is the implementation of a declaration

Declaration/Definition (Incomplete Type)

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

C++ Entities (class, functions, etc.) can be declared multiple times (with the same signature)

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

struct B {  // declaration and definition
    int b;
//  A   x;    // incomplete type
    A*  y;    // ok
};

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

Functions

Signature

Type signature defines the *inputs* and *outputs** for a function. A type signature includes the number of arguments, the types of arguments and the order of the arguments contained by a function

Function Parameter [formal]

A parameter is the variable which is part of the method's signature

Function Argument [actual]

An argument is the actual value (instance) of the variable that gets passed to the function

* (return type) if the function is generated from a function template

`https://stackoverflow.com/a/292390`

```
int f(int a, char* b);  // function declaration
                        // signature:  (int, char*)
                        // parameters: int a, char* b

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

// char f(int a, char* b); // compile error!! same signature

// int f(const int a, char* b); // invalid declaration!
                                // const int == int

int f(int a, const char* b);    // ok

int main() {
    f(3, "abc");  // function arguments: 3, "abc"
                  // "f" call f(int, const char*)
}
```

Call-by-Value

Call-by-value

The object is copied and assigned to input arguments of the method

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 a large array)

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

Call-by-Pointer

Call-by-pointer

The address of a variable is copied and assigned to input arguments of the method

Advantages:

- Allows a function to change the value of the argument
- Copy of the argument is not made (fast)

Disadvantages:

- The argument may be `nullptr`
- Dereferencing a pointer is slower than accessing a value directly

When to use:

- When passing *raw* arrays (use `const *` if read-only)

When not to use:

- Small objects

Call-by-Reference

Call-by-reference

The reference of a variable is copied and assigned to input arguments of the method

Advantages:

- Allows a function to change the value of the argument
- Copy of the argument is not made (fast)
- References must be initialized (no null pointer)
- Avoid implicit conversion

When to use:

- Structs or Classes (use `const &` if read-only)

Examples

```
struct MyStruct {
    int field;
};

void f1(int a);         // call by value
void f2(int& a);        // call by reference
void f3(const int& a); // call by const reference
void f4(MyStruct& a);  // call by reference
                        // note: requires a.field to access

void f5(int* a);        // call by pointer
void f6(const int* a); // call by const pointer
void f7(MyStruct* a);  // call by pointer
                        // requires a->field to access
//-----------------------------------------------------
char c = 'a';
f1('a');    // ok, pass by value
// f2('a'); // compile error!! pass by reference
```

`inline` Function Declaration

inline

`inline` specifier is a hint for the compiler. The code of the function can be copied where it is called (inlining)

```
inline void f(int a) { ... }
```

- It is just a hint. The compiler can ignore the hint (`inline` increases the compiler heuristic threshold)
- The compiled code is larger because the `inline` function is expanded in-place for every function call

GCC/Clang extensions allow to *force* inline/non-inline functions:

```
__attribute__((always_inline)) void f(int a) { ... }
__attribute__((noinline))      void f(int a) { ... }
```

Function Default Parameters

Default/Optional parameter

A **default parameter** is a function parameter that has a default value provided to it

If the user does not supply a value for this parameter, the default value will be used. If the user does supply a value for the default parameter, the user-supplied value is used instead of the default value

- All default parameters must be the rightmost parameters
- Default parameters can only be declared once
- Default parameters can improve compile time because they avoid defining other overloaded functions

```
void f(int a, int b = 20);
// void g(int a = 10, int b); // compile error!!

void f(int a, int b) { ... }  // default value of "b" already set

f(5);  // b is 20
```

Function Overloading (+ Ambiguous Matches)

Overloading

An **overloaded declaration** is a declaration with the same name as a previously declared identifier (in the same scope), which have different number of arguments and types

Overload resolution rules:

- An exact match
- A promotion (e.g. `char` to `int`)
- A standard type conversion (e.g. between `float` and `int`)
- A constructor or user-defined type conversion

```
void f(int a);
void f(float value);

void g(int a);

void h(int a);
void h(int a, int b = 0);
```

```
   f(0);    // ok
// f('a'); // ambiguous matches, compile error
   f(2.3f); // ok
// f(2.3); // ambiguous matches, compile error

   g(2.3);  // ok, standard type conversion

// h(3);    // ambiguous matches, compile error
```

Functor (Function as Argument)

Functor

Functors, or **function object**, are objects that can be treated as parameters*

```
int eval(int a, int b, int (*f)(int, int)) {
    return f(a, b);
}
int add(int a, int b) { // type: int (*)(int, int)
    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
```

*C++11 provides a more efficient and convenience way to pass *"procedure"* to other function called `lambda expression`

Function Attributes

C++ allows marking functions with standard properties to better express their intent:

- C++11 `[[noreturn]]` indicates that the function does not return
- C++14 `[[deprecated]]`, `[[deprecated("reason")]]` indicates the use of a function is discouraged (for some reason). It issues a warning if used
- C++17 `[[nodiscard]]` issues a warning if the return value is discarded
- C++17 `[[maybe_unused]]` suppresses compiler warnings on unused functions, if any (it applies also to other entities)

Function Attributes 2/2

```cpp
[[noreturn]] void f() {
    std::exit(0);
}

[[deprecated]] void my_rand() {
    rand();
}

[[nodiscard]] int g() {
    return 3;
}

[[maybe_unused]] void h() {}

//-------------------------------
my_rand();   // warning "deprecated"
g();         // warning "discard return value"
int x = g(); // no warning
```

Preprocessing

Preprocessor directives are lines preceded by a hash sign (#) which tell the compiler how to interprets the source code *before* compiling

Macro are preprocessor directives which replace 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 debugged
- Macro expansions can have strange side effects
- Macro have no namespace or scope

Preprocessing Syntax

- `#if <condition>`
  `#elif <condition>`
  `#else`
  `#endif`
- `#if defined(...)` equal to `#ifdef ...`
- `#if !defined(...)` equal to `#ifndef ...`
- `#define <macro>`
- `#undef <macro>` (every macro should be undefined for safety reasons)

Multi-line Preprocessing: `\` at the end of the line

Indent: `#    define`

Useful Macro

Commonly used macros:

`__LINE__` Integer value representing the current line in the source code file being compiled

`__FILE__` A string literal containing the presumed name of the source file being compiled

`__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

main.cpp:

```cpp
#include <iostream>
int main() {
  std::cout << __FILE__ << ":" << __LINE__;  // print main.cpp:2
}
```

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++`

Select code depending on the operation 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

Very Comprehensive Macro list:
`https://sourceforge.net/p/predef/wiki/Home/`

Macro (Common Error 1)

Do not define macro in header file and before includes!!

Example:

```cpp
#include <iostream>

#define value    // very dangerous!!
#include <big_lib>

int main() {
    std::cout << f(4); // should print 7, but it prints always 3
}
```

big_lib.hpp:

```cpp
int f(int value) {    // 'value' disapear
    return value + 3;
}
```

Macro (Common Error 2)

Use parenthesis in macro definition!!

Example:

```
#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
}
```

Macro (Common Error 3)

Macros make hard to find compile errors!!

Example:

```
 1: #include <iostream>
 2:
 3: #define F(a) {        \
 4:         ...          \
 5:         ...          \
 6:         return v;
 7:
 8: int main() {
 9:     F(3);      // compile error at line 10!!
10: }
```

- **In which line is the error??!**

Macro (Common Error 4)

Use curly brackets in multi-lines macros!!

Example:

```
#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!!

Variadic Macro

In C++11, a **variadic macro** is a special macro accepting a varying 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__);

int main() {
    PRINT(1, 2)
    PRINT(1, 2, 3)
}
```

Macro Use Cases

When macros 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`, specially for constant values and functions

```
#define SIZE 3 // replaced with
const int SIZE = 3;

#define SUB(a, b) ((a) - (b)) // replaced with
constexpr int sub(int a, int b) {
    return a - b;
}
```

- `#pragma once` It indicates that a (header) file is only to be parsed once, even if it is (directly or indirectly) included multiple times in the same source file
  It is an alternative (less portable) of the standard include guard (e.g. `myfile.h`):

```
#ifndef MYFILE_H       // (first line of the file)
#define MYFILE_H
...code...
#endif // MYFILE_H     // (last line of the file)
```

- `#pragma unroll` Applied immediately before a for loop, it replicates his body to eliminates branches. Unrolling enables aggressive instruction scheduling (supported by `Intel/Ibm/Clang` compilers)

- `#pragma message "text"` Display informational messages at compile time (every time this instruction is parsed)

- `_Pragma(<command>)` (C++11)
  It is an operator (like `sizeof`), and can be embedded in a macro (ex. `#define`)

```
#define MY_LOOP                \
    _Pragma(unroll)            \
    for(i = 0; i < 10; i++)    \
        cout << "c";
```

- `#error "text"` The directive emits a user-specified error message at compile time when the compiler parse the related instruction.

Macro Tricks

Find the size offset of a field inside a structure:

```
#define FIELD_OFFSET(structure, field)                                  \
    reinterpret_cast<size_t>(                                          \
        &((reinterpret_cast<structure*>(0))->field) )
```

Get the size of an arbitrary type without using `sizeof`

```
#define MY_SIZE(type, ret)                                              \
    { type x; ret = reinterpret_cast<char*>(&x + 1) -                  \
                    reinterpret_cast<char*>(&x); }
```

```
struct A {
    int   a;
    float b;
};

std::cout << FIELD_OFFSET(A, b); // print 4
int size;
MY_SIZE(A, size); // size = 8
```