Modern C++ Programming

5. Basic Concepts IV

Federico Busato

University of Verona, Dept. of Computer Science
2020, v3.0

Table of Context

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

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

Overview

A **function** (**procedure** or **routine**) is a piece of code that performs a *specific task*

Purpose:

- **Avoiding Code Duplication** less code for the same functionality $\rightarrow$ less bugs
- **Readability** better express what the code does
- **Organization** break the code in separate modules

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 <u>reference</u> of a variable is copied and assigned to input arguments of the method

Advantages:

- Allows a function to change the value of the argument (better readability compared with pointers)
- Copy of the argument is not made (fast)
- References must be initialized (no null pointer)
- Avoid implicit conversion (if not `const`)

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(c);     // ok, pass by value
// f2(c); // compile error!! pass by reference
```

inline

`inline` specifier allows a function to be defined identically (not only declared) in multiple translation units (source file + headers) [see "Translation Units" slides]

- `inline` is one of the most misunderstood features of C++
- `inline` is a hint for the linker. Without it, the linker can emit "multiple definitions" error
- It can be applied for optimization purposes only if the function has *internal* linkage (`static` or inside an `anonymous namespace`)
- C++17 `inline` can be also applied to variables

```cpp
inline        void f() { ... }
static        void g1() { ... }
static inline void g2() { ... }
namespace { // anonymous namespace -> same as static
    inline void g3() { ... }
}
```

`f()` :

- Can be defined in a header and included in multiple source files
- The linker removes all definitions except one
- Declaring `void f();` in a file that does not include the header is still valid because the function has *external* linkage

`g1(), g2(), g3()` :

- Can be defined in a header included in multiple source files
- The compiler replicates the code in each translation unit (the linker does not see these functions)
- Declaring `void g1();` in a file that does not include the header is no more valid because the function has *internal* linkage

inline (internal linkage)

`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 for the compiler that can ignore it (`inline` increases the compiler heuristic threshold)
- `inline` functions increase the binary size because they are 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);    // declaration
// 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 1/2

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* symbol (#) 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

Preprocessors

All statements starting with `#`

- `#include "my_file.h"`
  Inject the code in the current file

- `#define MACRO <expression>`
  Define a new macro

- `#undef MACRO`
  Undefine a macro
  (a macro should be undefined as early as possible for safety reasons)

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

Indent: `#   define`

Conditional Compiling

```
#if <condition>
    code
#elif <condition>
    code
#else
    code
#endif
```

- `#if defined(MACRO)` equal to `#ifdef MACRO`
  Check if a macro is defined

- `#if !defined(MACRO)` equal to `#ifndef MACRO`
  Check if a macro is not defined

Macro (Common Error 1)

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

```
#include <iostream>

#define value     // very dangerous!!
#include "big_lib.hpp"

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

`big_lib.hpp`:

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

Macro (Common Error 2)

Use parenthesis in macro definition!!

```cpp
#include <iostream>

#define SUB1(a, b)  a - b        // wrong
#define SUB2(a, b) (a - b)       // wrong
#define SUB3(a, b) ((a) - (b))  // correct

int main() {
    std::cout << (5 * SUB1(2, 1));   // print 9 not 5!!
    std::cout << SUB2(3 + 3, 2 + 2); // print 6 not 2!!
    std::cout << SUB3(3 + 3, 2 + 2); // print 2
}
```

Macro (Common Error 3)

Macros make hard to find compile errors!!

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

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

Macro (Common Error 4)

Use curly brackets in multi-lines macros!!

```cpp
#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!!

Macro (Common Error 5)

Macros do not have scope!!

```cpp
#include <iostream>

void f() {
    #define value 4
    std::cout << value;
}

int main() {
    f();                  // 4
    std::cout << value;   // 4
    #define value 3
    f();                  // 4
    std::cout << value;   // 3
}
```

Macro (Common Error 6)

Macros can have side effect!!

```c
#define MIN(a, b) ((a) < (b) ? (a) : (b))

int main() {
    int array1[] = { 1, 5, 2 };
    int array2[] = { 6, 2, 4 };
    int i = 0;
    int j = 0;
    int v1 = MIN(array1[i++], array2[j++]); // v1 = 5!!
    int v2 = MIN(array1[i++], array2[j++]); // segfault ☠
}
```

Reference: `arne-mertz.de/2019/03/macro-evil`

When Preprocessors are Necessary

- **Conditional compiling**: different architectures, compiler features, etc.
- **Mixing different languages**: code generation (example: `asm assembly`)
- **Complex name replacing**: see template programming

Otherwise, prefer `const` and `constexpr` for constant values and functions

```cpp
#define SIZE 3      // replaced with
const int SIZE = 3; // only C++11 at global scope

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

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++`

---

* MSVC defines `__cplusplus == 199711L` even for C++11/14. Link: `MSVC now correctly reports __cplusplus Avatar`

† `__GNUC__` is defined by many compilers outside. Link: `GCC __GNUC__Meaning`

Useful Macro

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:

- `sourceforge.net/p/predef/wiki/Home/`
- `Compiler predefined macros`
- `Abseil platform macros`

Macro (Common Error 7)

Macros depend on compilers and environment!!

```
struct A {
    int x;  // enable C++11 code
#if __cplusplus >= 201103
    A() = default;
#else
    A() {}
#endif
};
```

```
// should return ≈ 10.0f
float safe_function() {
    A a{}; // zero-initialization
    for (int i = 0; i < 10; i++)
        a.x += 1.0f;
    return a.x;
}
```

The code works fine on Linux, but not under Windows MSVC. MSVC sets `__cplusplus` to `199711` even if C++11/14/17 flag is set!! in this case the code can return `NaN`

---

see Lecture "Object-Oriented Programming II - Zero Initialization" and `MSVC now correctly reports __cplusplus`

Stringizing Operator (#)

The **stringizing macro operator** (`#`) causes the corresponding actual argument to be enclosed in double quotation marks `"`

```
#define STRING_MACRO(string) #string

std::cout << STRING_MACRO(hello);  // equivalent to "hello"
```

```
#define INFO_MACRO(my_func)                          \
    my_func                                          \
    std::cout << "call " << #my_func << " at " \
              << __FILE__ << ":" __LINE__;

void g(int) {}

INFO_MACRO( g(3) ) // print: call g(3) at my_file.cpp:7
```

Token-Pasting Operator (##)

The **token-concatenation (or pasting) macro operator** (`##`) allows combining two tokens (without leaving no blank spaces)

```
#define FUNC_GEN_A(tokenA, tokenB)  \
    void tokenA##tokenB() {}

#define FUNC_GEN_B(tokenA, tokenB)  \
    void tokenA##_##tokenB() {}

FUNC_GEN_A(my, function)
FUNC_GEN_B(my, function)

int main() {
    myfunction();  // ok, from FUNC_GEN_A
    my_function(); // ok, from FUNC_GEN_B
}
```

Macro (Common Error 8)

Code injection

```cpp
#include <cstdio>

#define CHECK_ERROR(condition) \
{
   if (condition) {
      std::printf("expr: " #condition " failed at line %d\n",
                  __LINE__);
   }
}

int main() {
    int t = 5, s = 3;
    CHECK_ERROR(t > s) // print "expr: t > s failed at line 13"
    CHECK_ERROR(t % s == 0) // segmentation fault!!!
}      // printf interprets "% s" as a format specifier
```

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)
}
```

- `#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 and #error

- `_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 Trick

Convert a number literal to a string literal

```
#define TO_LITERAL_AUX(x) #x
#define TO_LITERAL(x)     TO_LITERAL_AUX(x)
```

Motivation: avoid integer to string conversion (performance)

```
int main() {
   int  x1   = 3 * 10;
   int  y1   = __LINE__ + 4;
   char x2[] = TO_LITERAL(3);
   char y2[] = TO_LITERAL(__LINE__);
}
```