CS100 Recitation 11

FlashHu

May 26, 2022

Contents

Contents

operator()

The function call operator provides function semantics for any object.
A class that overloads operator() is called **functor**.

```
#include <functional> // for std::greater<int>
class Modulo {
   public:
    Modulo(int x): _x(x) {}
    inline bool operator()(int& a, int& b) const {
        return a % _x < b % _x;
    }
   private:
    const int _x;
};
std::priority_queue<int, std::vector<int>, std::greater<int>> q;
std::sort(a, a + 10, std::greater<int>());
std::sort(a, a + 10, Modulo(5));
```

Contents

Prefix and postfix operator++

```
struct X
{
    // prefix increment
    X& operator++()
    {
        // actual increment takes place here
        return *this; // return new value by reference
    }

    // postfix increment
    X operator++(int) // the argument is ignored
    {
        X old = *this; // copy old value
        operator++();  // prefix increment
        return old;    // return old value
    }
}
```

It appears to be a binary operator `class->member`.
However, it can only be overloaded as a unary operator.

- `class->member` includes 2 steps: `(*class).member`.
- The fact is that `operator.` cannot be overloaded.
- Therefore the only modifiable step is the first step: specify the address of its content.

Unlike other operators that have no restrictions on the type of return value, `operator->` must return a pointer or another class with overloaded `operator->` to be realistically usable.

```
template <typename T>
class iterator_like {
   private:
    T* m_content;
   public:
    T& operator*() const {
        // implementation-specific
    }
    T* operator->() const {
        return &operator*(); // &(*(*this))
    }
};
```

However, what if class T overloads its operator&? A tricky solution here:

```
#include <memory>
        return std::addressof(operator*());
```

Contents

Conditional Directives

```
#if expression
#ifdef identifier
#ifndef identifier
#elif expression
#else
#endif
```

Difference between `#if` and `#ifdef`:

- `expression` is a constant expression. `expression` can also be an identifier. The result is 1 if the identifier was defined as a macro name, otherwise the result is 0.
- `identifier` can only be an identifier.

Examples:

- Include guard
- Debug mode control

Contents

Macros

```
#define identifier
#define identifier replacement-list
#define identifier(parameters) replacement-list
#define identifier(parameters, ...) replacement-list
#undef identifier
g++ -o ... -Didentifier
```

- Remember it only replace the text where `identifier` appears with `replacement-list`! The `parameters` and `replacement-list` doesn't need to be a single object, so pay attention to the potential problem caused by `()` `,` `;`!
- Unlike functions which can be overloaded, macros defined by the same identifier makes the program ill-formed unless the definitions are identical.

Object-like macros

```
#define identifier replacement-list
```

Examples:

```
#define PI 3.14

// A terrible example widely used by OIers
#define int long long // replace int by long long below
typedef long long int; // Error
using int = long long; // Error
```

Function-like macros

```
#define identifier(parameters) replacement-list
#define identifier(parameters, ...) replacement-list
#define identifier(...) replacement-list
```

The later two are with variable number of arguments.

- The additional arguments (...) can be accessed using `__VA_ARGS__` identifier.
- replacement-list may contain the token sequence `__VA_OPT__(content)`, which is replaced by `content` if `__VA_ARGS__` is non-empty, and nothing otherwise.

Examples:

```
#define LEN(x, y) sqrt((x)*(x)+(y)*(y)) // OK
#define LEN(x, y) sqrt(x*x+y*y) // Potential problems!
#define eprintf(format, ...) \ // define in multiline
    fprintf(stderr, format, __VA_ARGS__)
```

Operators in #define

- #: replace the following identifier by its string literal
- ##: concatenate two successive identifiers

Examples:

```
#define PRINT(v) std::cout << (#v ":") << v
int tmp = 1;
PRINT(v);
// output: tmp:1

#define DECLARE(fun, id) void fun_##id(int x)
DECLARE(test, 5);
// void test_5(int x);
```

Preprocessor metaprogramming

Like template, preprocessor is able to realize complex functions during compile time, although it is not designed to do that.
In C++ we prefer template metaprogramming, and preprocessor metaprogramming is out of fashion.
Example:

- Some compiler doesn't support binary integer literal such as `0b0100110`
- Binary integer literal encoding (https://paste.ubuntu.com/p/h875zMBKJX/)

A tricky example: FOR_EACH

https://www.scs.stanford.edu/~dm/blog/va-opt.html

```
#define PARENS ()
#define EXPAND(...) EXPAND1(EXPAND1(EXPAND1(EXPAND1(__VA_ARGS__))))
#define EXPAND4(...) EXPAND3(EXPAND3(EXPAND3(EXPAND3(__VA_ARGS__))))
#define EXPAND3(...) EXPAND2(EXPAND2(EXPAND2(EXPAND2(__VA_ARGS__))))
#define EXPAND2(...) EXPAND1(EXPAND1(EXPAND1(EXPAND1(__VA_ARGS__))))
#define EXPAND1(...) __VA_ARGS__
#define FOR_EACH(macro, ...) \
    __VA_OPT__(EXPAND(FOR_EACH_HELPER(macro, __VA_ARGS__)))
#define FOR_EACH_HELPER(macro, a1, ...) \
    macro(a1)                           \
        __VA_OPT__(FOR_EACH_AGAIN PARENS(macro, __VA_ARGS__))
#define FOR_EACH_AGAIN() FOR_EACH_HELPER
```

A tricky example: debug info printing by FOR_EACH

```
#include <iostream>
#define DEBUG(...)                              \
    do {                                        \
        std::cout << __LINE__ << ": "; \
        FOR_EACH(PRINT, __VA_ARGS__);  \
        std::cout << std::endl;                 \
    } while (0)
#define PRINT(v) std::cout << (#v "=") << v << ' ';
int main() {
    int a = 1;
    double b = 2.3;
    char c[] = "hello";
    DEBUG(a, b, c);
} // output: 25: a=1 b=2.3 c=hello
```

Contents

Other Directives

```
#include <...> // usually for standard libraries
#include "..." // usually for local header files
#error message // stop compilation
#pragma parameters // implementation-specific behavior of the
    compiler
```

Examples:

```
#include <iostream>
#include "matrix.hpp"
#pragma GCC optimize(2)
#pragma omp parallel
```

CS100 Introduction to Programming

Recitation 11-CMake

Wei Jiaxin

What is program?

- A program in Linux is a file with execute permission, which can be a script or binary file
- Commonly used "`cd,ls`" and other commands are executable files located in the /bin directory
- When programming in C++, a compiler can be used to compile a text file into an executable program

Compiler g++

- `g++ filename.cpp` generates .out executable file in the current directory by default
- `g++ -o name.out filename.cpp` generates an executable file with a custom filename in the current directory

Why we need CMake?

- A large project usually contains many folders and source files, and there may be complex dependencies between classes
- A large number of compilation commands need to be entered, and the entire compilation process becomes extremely cumbersome
- So we need CMake to manage source code

What is CMake?

- CMake is a meta build system that uses scripts called ”CMakeLists” to generate build files for a specific environment
- CMake is distributed as open-source software under permissive BSD-3-Clause license
- CMake only needs to maintain several “CMakeLists.txt” files, which greatly reduces the difficulty of maintaining the entire project

How to use CMake?

- In a cmake project, use the `cmake` command to generate a Makefile file
- And then use the `make` command to compile the entire project according to the contents of the Makefile file

Step 1

- Write the source code
- The source file with the main function will be compiled to generate an executable file
- Other source files will be packaged into libraries for other programs to call

Step 2

- Create a new “CMakeLists.txt” file to tell CMake what to do with the files in the directory
- The following contents are needed:

```
#Declare the required minimum version of cmake

cmake_minimum_required(VERSION 2.8)

#Add C++11 standard support

set( CMAKE_CXX_FLAGS "-std=c++11" )

#Open debug mode

set(CMAKE_BUILD_TYPE "Debug")

#Declare a cmake project

project( HelloWorld )

#Add compile parameters

add definitions("-g")
```

```
#Add a static library
add_library( hello libHelloWorld.cpp )
#Add a shared library
add_library( hello SHARED libHelloWorld.cpp )
#Add an executable program
add_executable( helloWorld helloWorld.cpp )
#Link the executable to the library
target_link_libraries( helloWorld hello )
```

Step 3

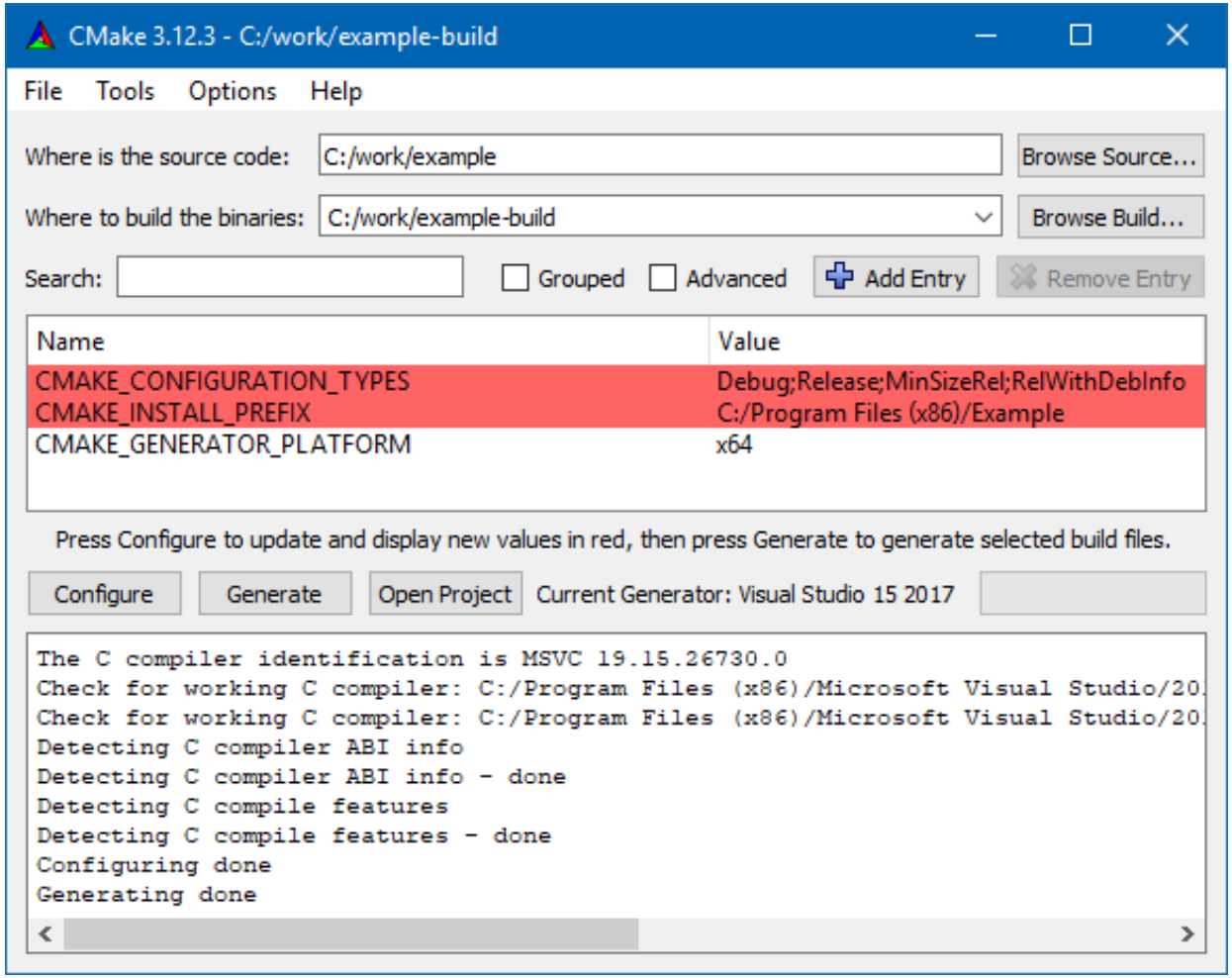
- Create a new intermediate folder “build” in the current directory, and then enter the build folder (separate the intermediate files generated by cmake from the source code)
- Compile the upper-level directory (the directory where the source code is located) through the `cmake ..` command

- Compile the automatically generated Makefile through the `make` command to get the executable program helloWorld
- When you need to release the source code, just delete the build folder

CMake on Windows

- Run cmake-gui.exe
- The top two entries are the source code and binary directories
- They allow you to specify where the source code is for what you want to compile and where the resulting binaries should be placed.

CMake GUI


CMake 3.12.3 - C:/work/example-build
File
Tools
Options
Help
Where is the source code:
C:/work/example
Browse Source...
Where to build the binaries:
C:/work/example-build
Browse Build...
Search:
Grouped
Advanced
Add Entry
Remove Entry
Name
Value
CMAKE_CONFIGURATION_TYPES
Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_INSTALL_PREFIX
C:/Program Files (x86)/Example
CMAKE_GENERATOR_PLATFORM
x64
Press Configure to update and display new values in red, then press Generate to generate selected build files.
Configure
Generate
Open Project
Current Generator: Visual Studio 15 2017
The C compiler identification is MSVC 19.15.26730.0
Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio/20.
Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio/20.
Detecting C compiler ABI info
Detecting C compiler ABI info - done
Detecting C compile features
Detecting C compile features - done
Configuring done
Generating done

Useful links

- CMake tutorial https://cmake.org/cmake/help/latest/guide/tutorial/index.html
- CLion CMake tutorial https://www.jetbrains.com/help/clion/quick-cmake-tutorial.html