Lab Report for Object-oriented Programming course
Lab 2: C/C++ Preprocessor

Wang, Chen
16307110064
School of Software
Fudan University

April 18, 2019

Contents

Chapter 1

Background Knowledge & Concepts Required for This Lab

1.1 C/C++ Compiling Process

1.1.1 Overall process of compiling

Compiling a source code file in C++ is a four-step process.[1] For example, if you have a C++ source code file named *prog1.cpp* and you execute the compile command

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```

the compilation process looks like this:

1. The C++ preprocessor copies the contents of the included header files into the source code file, generates macro code, and replaces symbolic constants defined using *#define* with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

By using appropriate compiler options, we can stop this process at any stage.

1. To stop the process after the preprocessor step, you can use the *-E* option:

   ```
   g++ -Wall -std=c++11 -E prog1.cpp
   ```

[1] `http://faculty.cs.niu.edu/mcmahon/CS241/Notes/compile.html`

The expanded source code file will be printed on standard output (the screen by default); you can redirect the output to a file if you wish. Note that the expanded source code file is often incredibly large - a 20 line source code file can easily produce an expanded file of 20,000 lines or more, depending on which header files were included.

2. To stop the process after the compile step, you can use the *-S* option:

   ```
   g++ -Wall -std=c++11 -S prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.s*.

3. To stop the process after the assembly step, you can use the *-c* option:

   ```
   g++ -Wall -std=c++11 -c prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.o*. The entire process for compiling is shown in the figure 1.1

1.1.2 Preprocessing

To program in C and C++, you need to understand the steps and tools in the compilation process. Some languages (C and C++, in particular) start compilation by running a *preprocessor* on the source code. The preprocessor is a simple program that replaces patterns in the source code with other patterns the programmer has defined (using *preprocessor directives*). Preprocessor directives are used to save typing and to increase the readability of the code. However, from the author of the book TIC, the design of C++ is meant to discourage much of the use of the preprocessor, since it can cause subtle bugs. The preprocessed code is often written to an intermediate file.

1.1.3 Parsing

Compilers usually do their work in two passes. The first pass *parses* the preprocessed code. The compiler breaks the source code into small units and organizes it into a structure called a *tree*. In the expression "**A + B**" the elements '**A**', '+,' and '**B**' are leaves on the parse tree.

1.1.4 Global optimization

A *global optimizer* is sometimes used between the first and second passes to produce smaller, faster code.

1.1.5 Code generation

In the second pass, the *code generator* walks through the parse tree and generates either assembly language code or machine code for the nodes of the tree. If the code generator creates assembly code, the assembler must then be run. The end result in both cases is an object module (a file that typically has an extension of **.o** or **.obj**).

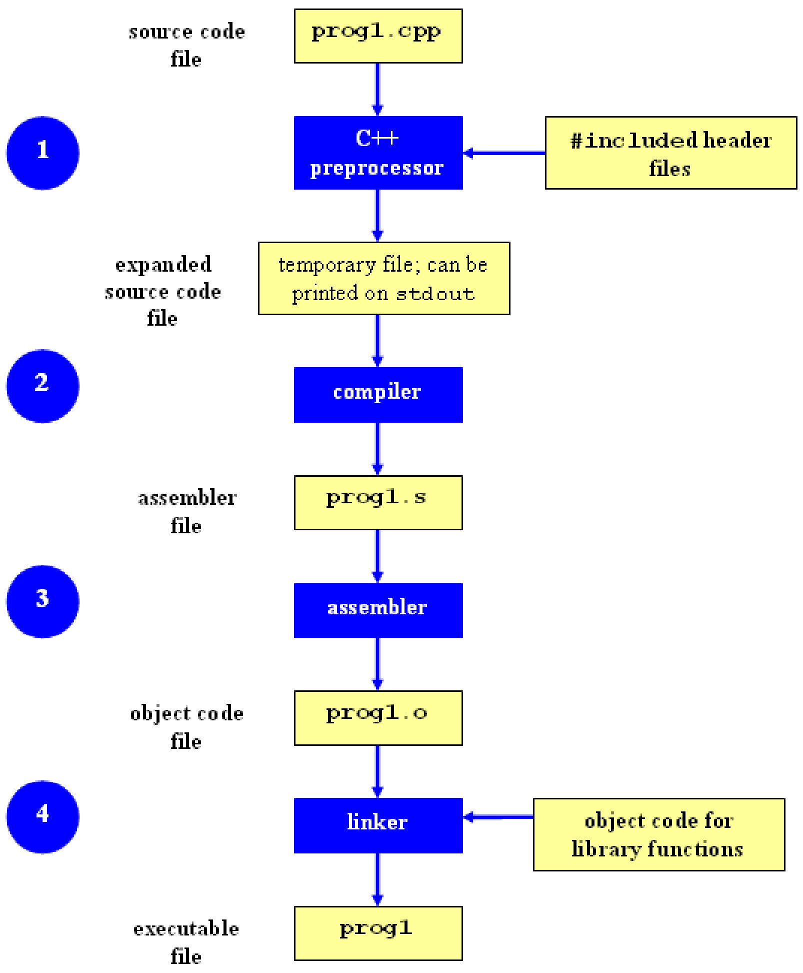


Figure 1.1: Overall compiling process

1.1.6 Peehole optimization

A *peephole optimizer* is sometimes used in the second pass to look for pieces of code containing redundant assembly-language statements.

1.1.7 Linking

The use of the word "object" to describe chunks of machine code is an unfortunate artifact. The word came into use before objectoriented programming was in general use. "Object" is used in the same sense as "goal" when discussing compilation, while in objectoriented programming it means "a thing with boundaries."

The *linker* combines a list of object modules into an executable program that can be loaded and run by the operating system. When a function in one object module makes a reference to a function or variable in another object module, the linker resolves these references; it makes sure that all the external functions and data you claimed existed during compilation do exist. The linker also adds a special object module to perform start-up activities.

The linker can search through special files called *libraries* in order to resolve all its references. A library contains a collection of object modules in a single file. A library is created and maintained by a program called a *librarian.*

1.2 C/C++ Preprocessing

1.2.1 The need of preprocessing

1.2.2 Different preprocessing algorithms

1.2.3 Preprocessing algorithm utilized by the current g++

Chapter 2

Specifications of This Lab

2.1 Regulations in the preprocessing process

2.2 Test cases designed in the lab

2.3 Other specifications of the programming

Chapter 3

Structure and the OO Ideas Adopted

3.1 Objected-oriented ideas adopted in the implementation

3.1.1 Encapsulation

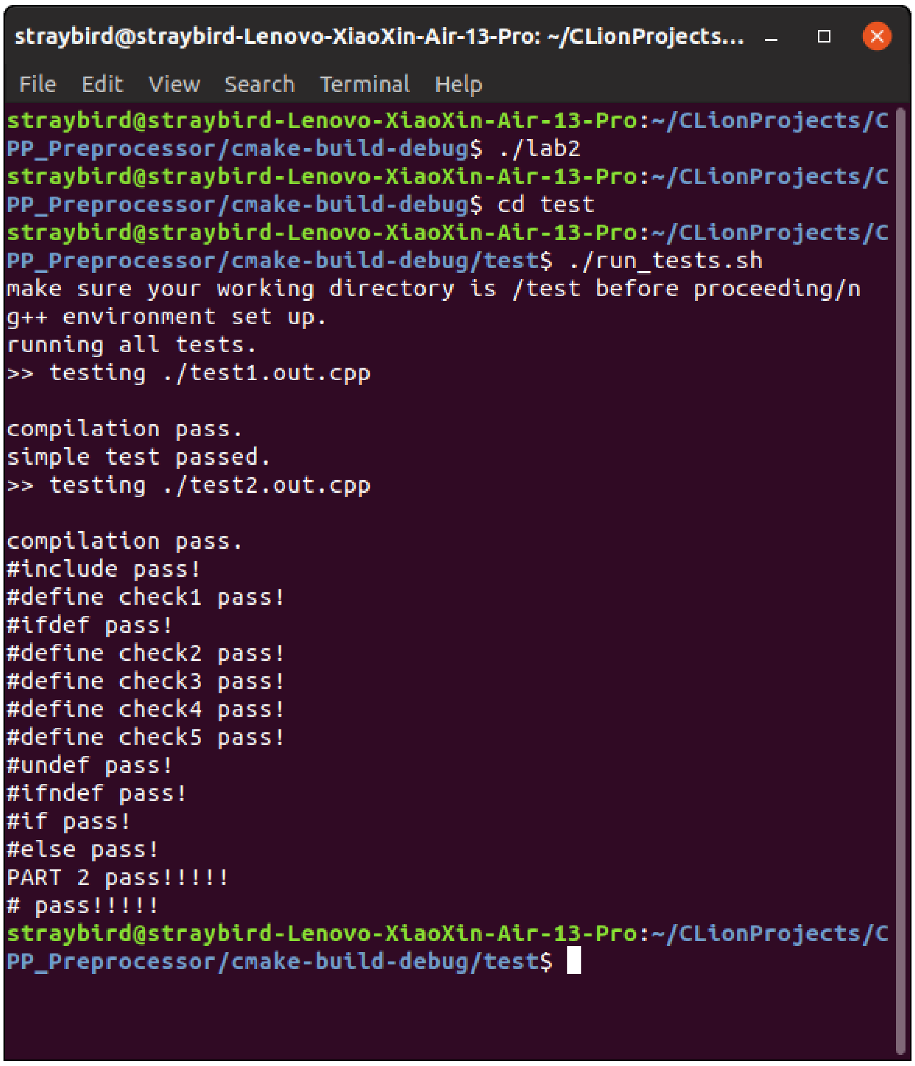
Chapter 4

Running Result of My Implementation

The following screenshots are the tests that are identical to the steps in the requirement documentation and proves that my version of implementation functions identical to the standard version.

4.1 Test result of the testcases

The results are shown as Figure 4.2.

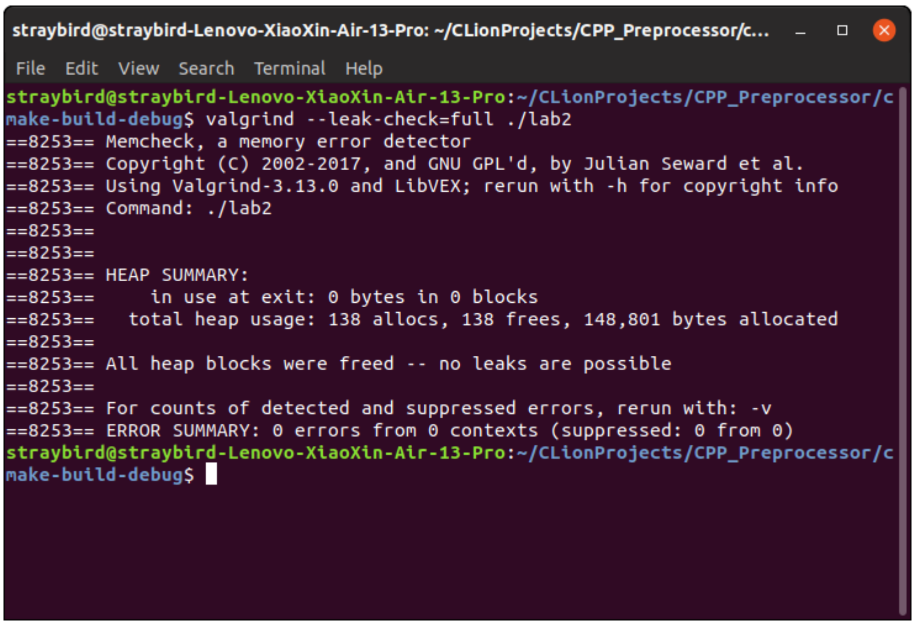

straybird@straybird-Lenovo-XiaoXin-Air-13-Pro: ~/CLionProjects...
File Edit View Search Terminal Help

```
straybird@straybird-Lenovo-XiaoXin-Air-13-Pro:~/CLionProjects/C
PP_Preprocessor/cmake-build-debug$ ./lab2
straybird@straybird-Lenovo-XiaoXin-Air-13-Pro:~/CLionProjects/C
PP_Preprocessor/cmake-build-debug$ cd test
straybird@straybird-Lenovo-XiaoXin-Air-13-Pro:~/CLionProjects/C
PP_Preprocessor/cmake-build-debug/test$ ./run_tests.sh
make sure your working directory is /test before proceeding/n
g++ environment set up.
running all tests.
>> testing ./test1.out.cpp

compilation pass.
simple test passed.
>> testing ./test2.out.cpp

compilation pass.
#include pass!
#define check1 pass!
#ifdef pass!
#define check2 pass!
#define check3 pass!
#define check4 pass!
#define check5 pass!
#undef pass!
#ifndef pass!
#if pass!
#else pass!
PART 2 pass!!!!!
# pass!!!!!
straybird@straybird-Lenovo-XiaoXin-Air-13-Pro:~/CLionProjects/C
PP_Preprocessor/cmake-build-debug/test$
```

Figure 4.1: Testcase Result


straybird@straybird-Lenovo-XiaoXin-Air-13-Pro: ~/CLionProjects/CPP_Preprocessor/c...
File Edit View Search Terminal Help

```
straybird@straybird-Lenovo-XiaoXin-Air-13-Pro:~/CLionProjects/CPP_Preprocessor/c
make-build-debug$ valgrind --leak-check=full ./lab2
==8253== Memcheck, a memory error detector
==8253== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8253== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==8253== Command: ./lab2
==8253==
==8253==
==8253== HEAP SUMMARY:
==8253==     in use at exit: 0 bytes in 0 blocks
==8253==   total heap usage: 138 allocs, 138 frees, 148,801 bytes allocated
==8253==
==8253== All heap blocks were freed -- no leaks are possible
==8253==
==8253== For counts of detected and suppressed errors, rerun with: -v
==8253== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
straybird@straybird-Lenovo-XiaoXin-Air-13-Pro:~/CLionProjects/CPP_Preprocessor/c
make-build-debug$
```

Figure 4.2: Memory Leak Check

Chapter 5

Memory Leak

5.1 Potential Memory Leak

5.2 Prove of Free from Memory Leak in my Implementation

Bibliography

[1] Wikipedia contributors. (2019, February 3). Encapsulation (computer programming). In *Wikipedia, The Free Encyclopedia.* Retrieved 10:19, March 23, 2019, from `https://en.wikipedia.org/w/index.php?title=Encapsulation_(computer_programming)&oldid=881507936`

[2] Wikipedia contributors. (2019, March 17). Reversi. In *Wikipedia, The Free Encyclopedia.* Retrieved 10:20, March 23, 2019, from `https://en.wikipedia.org/w/index.php?title=Reversi&oldid=888167585`

[3] Wikipedia contributors. (2019, March 15). Polymorphism (computer science). In *Wikipedia, The Free Encyclopedia.* Retrieved 10:21, March 23, 2019, from `https://en.wikipedia.org/w/index.php?title=Polymorphism_(computer_science)&oldid=887878749`

[4] Wikipedia contributors. (2019, February 27). Object-oriented programming. In *Wikipedia, The Free Encyclopedia.* Retrieved 10:22, March 23, 2019, from `https://en.wikipedia.org/w/index.php?title=Object-oriented_programming&oldid=885274966`

[5] Wikipedia contributors. (2019, February 21). Inheritance (object-oriented programming). In *Wikipedia, The Free Encyclopedia.* Retrieved 10:22, March 23, 2019, from `https://en.wikipedia.org/w/index.php?title=Inheritance_(object-oriented_programming)&oldid=884436146`