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**).

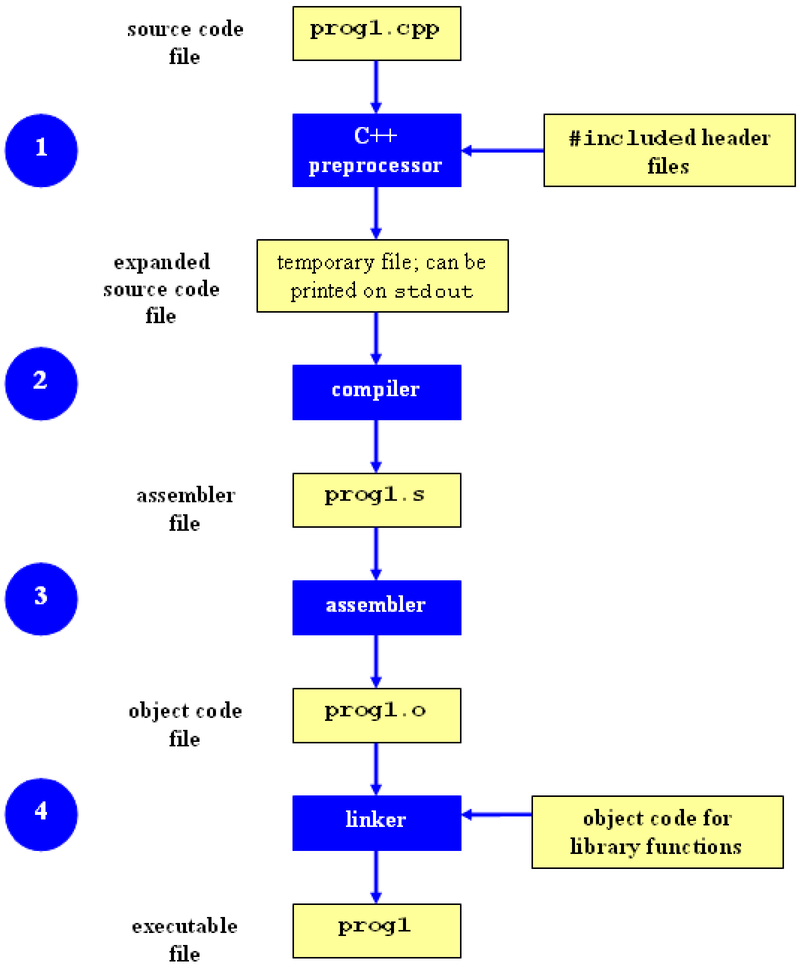

Figure 1.1: Overall compiling process

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 Preprocessing

1.2.1 The need of preprocessing

In computer science, a **preprocessor** is a program that processes its input data to produce output that is used as input to another program. The output is said to be a **preprocessed** form of the input data, which is often used by some subsequent programs like compilers. The amount and kind of processing done depends on the nature of the preprocessor; some preprocessors are only capable of performing relatively simple textual substitutions and macro expansions, while others have the power of full-fledged programming languages. A common example from computer programming is the processing performed on source code before the next step of compilation. In some computer languages (e.g., C and PL/I) there is a phase of translation known as *preprocessing.* It can also include macro processing, file inclusion and language extensions.

1.2.2 Different preprocessing algorithms

Preprocessors can be divided into different types: lexical preprocessors, syntactic preprocessors and general purpose preprocessors. Each type will be further discussed in the following subsections.

1.2.3 Lexical preprocessors

Lexical preprocessors are the lowest-level of preprocessors as they only require lexical analysis, that is, they operate on the source text, prior to any parsing, by performing simple substitution of tokenized character sequences for other tokenized character sequences, according to user-defined rules. They typically perform macro substitution, textual inclusion of other files, and conditional compilation or inclusion.

The most common example of this is the C preprocessor, which takes lines beginning with '#' as directives. Because it knows nothing about the underlying language, its use has been criticized and many of its features built directly into other languages. For example, macros replaced with aggressive inlining and templates, includes with compile-time imports (this requires the preservation of type information in the object code, making this feature impossible to retrofit into a language); conditional compilation is effectively accomplished with if-then-else and dead code elimination in some languages. However, a key point to remember is that all preprocessor directives should start on a new line.

Other lexical preprocessors include the general-purpose m4, most commonly used in cross-platform build systems such as autoconf, and GEMA, an open source macro processor which operates on patterns of context.

1.2.4 Syntactic preprocessors

Syntactic preprocessors were introduced with the Lisp family of languages. Their role is to transform syntax trees according to a number of user-defined rules. For some programming languages, the rules are written in the same language as the program (compile-time reflection). This is the case with Lisp and OCaml. Some other languages rely on a fully external language to define the transformations, such as the XSLT preprocessor for XML, or its statically typed counterpart CDuce.

Syntactic preprocessors are typically used to customize the syntax of a language, extend a language by adding new primitives, or embed a domain-specific programming language (DSL) inside a general purpose language.

1.2.5 General purpose preprocessor

Most preprocessors are specific to a particular data processing task (e.g., compiling the C language). A preprocessor may be promoted as being *general purpose*, meaning that it is not aimed at a specific usage or programming language, and is intended to be used for a wide variety of text processing tasks.

M4 is probably the most well known example of such a general purpose preprocessor, although the C preprocessor is sometimes used in a non-C specific role.

1.2.6 Preprocessing algorithm utilized by the current g++

The **C preprocessor** or **cpp** is the macro preprocessor for the C and C++ computer programming languages. The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control.

In many C implementations, it is a separate program invoked by the compiler as the first part of translation.

The language of preprocessor directives is only weakly related to the grammar of C, and so is sometimes used to process other kinds of text files.

Preprocessing is defined by the first four (of eight) *phases of translation* specified in the C Standard.

1. Trigraph replacement: The preprocessor replaces trigraph sequences with the characters they represent.
2. Line splicing: Physical source lines that are continued with escaped newline sequences are *spliced* to form logical lines.
3. Tokenization: The preprocessor breaks the result into *preprocessing tokens* and whitespace. It replaces comments with whitespace.
4. Macro expansion and directive handling: Preprocessing directive lines, including file inclusion and conditional compilation, are executed. The preprocessor simultaneously expands macros and, in the 1999 version of the C standard, handles **_Pragma** operators.

However, in this lab we are not required to accomplish all the steps of the C preprocessor program and the test case only cover a small part of the required tasks of a C/C++ preprocessor. The details of this lab are shown in the consequent chapters.

Chapter 2

Specifications of This Lab

2.1 The description of this lab

The experiment requires a source code preprocessor to implement precompilation of macro directives in the code file. The preprocessor will process the macro definitions in the code before the code is compiled. Common instructions are: #include, #define, #undef, #ifdef, #ifndef, #if, #endif and so on.

You can use *g++ -E test1.cpp ¿ output.cpp* to see the code after *test1.cpp* is precompiled in *output.cpp*. In this lab, you need to implement a precompiled processor (without needing to deal with the C++ standard library) to handle instruction precompilation in simple scenarios.

2.2 Specifications of this lab

1. Implementation: In the lab we need to write our code in *lab2.cpp*. *lab2.cpp* provides an entry function, we need to implement our code at this entry location.

2. Test: There are two cpp test files in the test folder. *Test1.cpp* is very simple and can help us debug at an early stage. *Test2.cpp* is relatively complex and requires us to carefully identify the various macro processing scenarios.

3. Run: First we need to compile *lab2.cpp* (compiled with c++11 standard), and run the compiled file. After running, two *test.out.cpp* files are generated in the test folder. Then run the *run_tests.sh* file in the test folder and make sure we are in the test directory before running this file.

2.3 Test cases designed in the lab

The test cases in this lab are designed to check the following directives.

- #include – 10% (do not need to deal with **#include "iostream"**)
- #define (check1 to check5)– 10%

- #undef – 10%
- #ifdef – 10%
- #else – 10%
- #ifndef – 10%
- #if – 10%
- #define function(PART 2) – 10%
- #define function(PART 3) – 10% (for 5% and ## processing, _# and ## can be processed as long as they can pass the test file)
- **No memory leaks - 10%**

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.

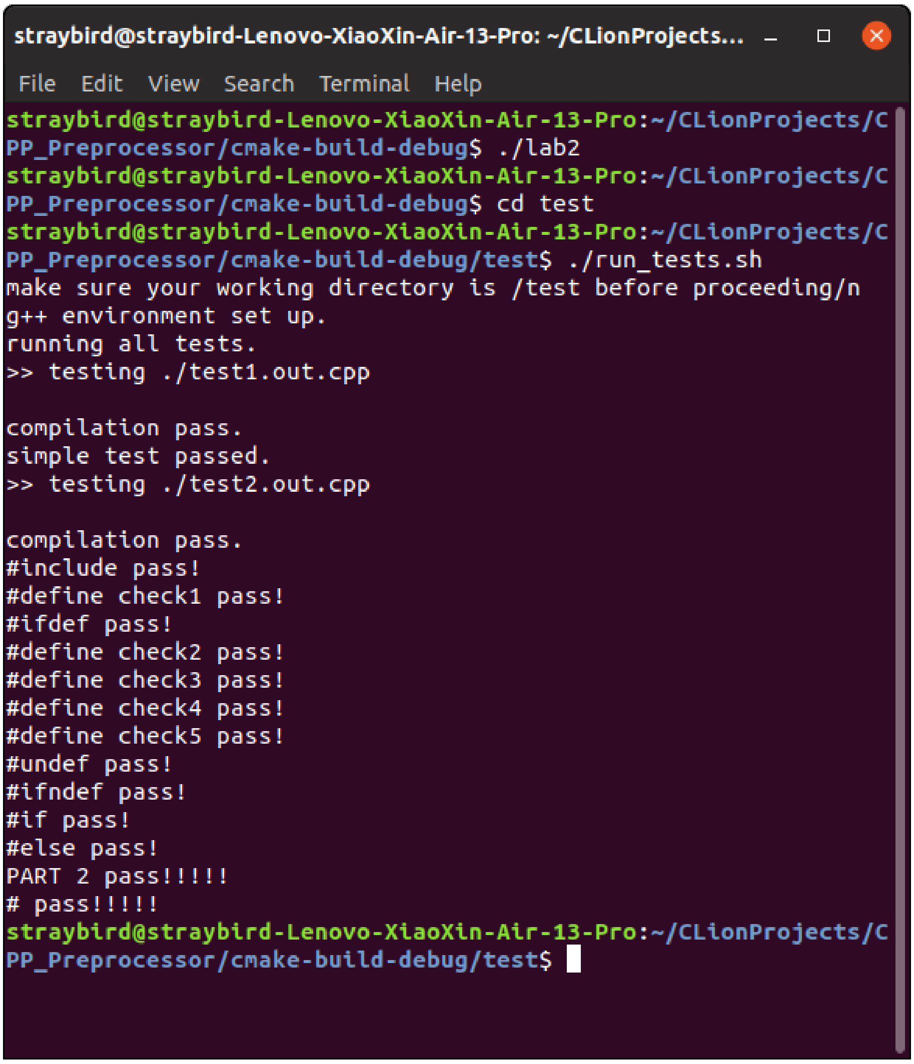

Figure 4.1: Testcase Result

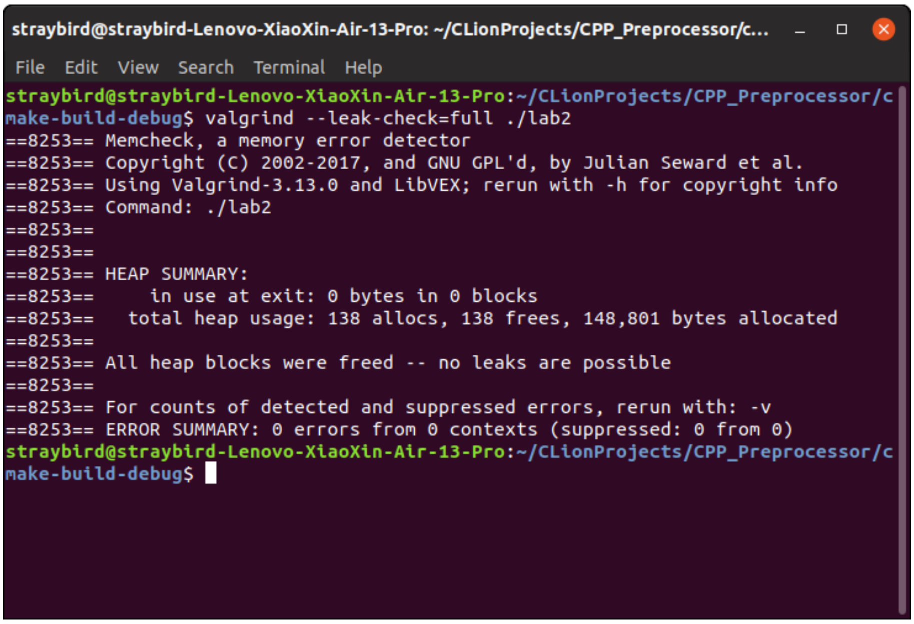

Figure 4.2: Memory Leak Check

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`