MEMORIAL UNIVERSITY

Faculty of Engineering and Applied Science

Lab Handout
ENGI3891

30 Sep, 7 Oct 2015

Lab 1: Compiler workflow

1 Purpose and outcomes

This lab will help you explore **what a compiler does**. The compiler is the primary tool used by the programmer, yet so far you may not understand what it really does or how. This lab is designed to illuminate the compiler and to introduce a simple automated workflow. After completing this lab, you should be able to:

1. understand the differences among source, object and executable files,
2. understand — in broad terms — the process of symbol resolution,
3. compile source files to object or executable files and
4. diagnose common linker errors.

2 Preparation

You need a compiler, preferably g++ as installed on the computers in EN3000/3029, and its associated tools. These tools can be installed on Mac OS X as part of XCode; Windows versions can be obtained through MinGW as described in `https://www.engr.mun.ca/~anderson/teaching/ENGI3891/reference`.

3 Procedure

As a pair of lab partners, follow the instructions given below. Unlike the revision control lab, there is only one set of instructions and you will only need one computer. **Whenever you encounter the text <<output>> in the instructions,** make a note of the program output you see on the console. These outputs should go in your lab report, along with answers to any questions below.

For this and future labs, you may choose your lab partner. You may choose to remain with your partner from lab 0, but you are not required to.

3.1 Create a workspace

Create a directory for this lab, then fill it with some C++ source files using a text editor (Notepad++ is installed on the lab computers and is a sensible enough choice):

```
H:\engi3891>mkdir lab1
H:\engi3891>cd lab1
H:\engi3891\lab1>
```

Listing 1: increment.h

```
namespace engi3891 {
namespace lab1 {
```

```
int increment(int&);

}
}
```

Listing 2: increment.cpp

```
#include "increment.h"

int engi3891::lab1::increment(int& x)
{
        ++x;
        return x;
}
```

Listing 3: decrement.h

```
namespace engi3891 {
namespace lab1 {

int decrement(int&);

}
}
```

Listing 4: decrement.cpp

```
#include "decrement.h"

int decrement(int& x)
{
        --x;
        return x;
}
```

Listing 5: sillyMath.h

```
namespace engi3891 {
namespace lab1 {

int add(int, unsigned int);
int subtract(int, unsigned int);

}
}
```

Listing 6: sillyMath.cpp

```
#include "sillyMath.h"
#include "increment.h"
#include "decrement.h"
using namespace engi3891;

int engi3891::lab1::add(int x, unsigned int y)
{
        for (unsigned int i = 0; i < y; i++)
                increment(x);

        return x;
}

int lab1::subtract(int x, unsigned int y)
{
        for (unsigned int i = 0; i < y; i++)
                decrement(x);

        return x;
}
```

Listing 7: main.cpp

```
#include "sillyMath.h"
#include <iostream>
using namespace engi3891::lab1;

int main(int argc, char *argv[])
{
        int x = -7;
        unsigned int y = 10;

        std::cout << x << " + " << y << " = " << add(x,y) << "\n";
        std::cout << x << " - " << y << " = " << subtract(x,y) << "\n";

        return 0;
}
```

3.2 Compile increment.cpp

Previously in this course, we've used the compiler to compile all of our sources files at once. Now, we will see the command for compiling **a single source file** into an *object file*. An object file (not to be confused with a C++ object!) is the result of compiling a single source file: it includes the definitions of the functions defined in that source file, but not the functions defined in other source files (which we need to create a complete program).

We can tell g++ to produce an object file by passing it the `-c` flag, which means "compile but don't link":

```
H:\engi3891\lab1>g++ -c increment.cpp -o increment.o
```

This produces an object file called `increment.o`:

```
H:\engi3891\lab1>dir
```

```
<<output>>
H:\engi3891\lab1>file increment.cpp
<<output>>
H:\engi3891\lab1>file increment.o
<<output>>
```

The `file` command inspects the contents of a file and tells us what kind of file it thinks we've run it against. In the first case, it sees content that looks like a C program: we're not using many special C++ keywords like `class`, so it can't tell the difference. In the second case, `file` tells us that that `increment.o` is:

1. meant for use with Microsoft Windows,
2. written in the Common Object File Format (COFF),
3. meant for CPUs that understand Intel 80386 instructions[1] and
4. an object file.

We can inspect the contents of this file using the `nm` command:

```
H:\engi3891\lab1>nm increment.o
<<output>>
```

We will ignore the symbols with types other than `T` (so, in this case, all but `__ZN8engi38914lab19incrementERi`). The remaining element, `__ZN8engi38914lab19incrementERi`, is the definition of our `increment()` function. Its name has been processed using the C++ *name mangler* into a name that is a valid identifier (symbol names, like C and C++ identifiers, cannot contain most special characters like colons or ampersands). We can *demangle* the name using the `c++filt` tool or by passing the `--demangle` parameter to `nm`:

```
H:\engi3891\lab1>c++filt __ZN8engi38914lab19incrementERi
<<output>>
H:\engi3891\lab1>nm increment.o | c++filt
<<output>>
H:\engi3891\lab1>nm --demangle increment.o
<<output>>
```

In the first of these commands, we simply passed the mangled directly to the demangler. In the second case, we ran the `nm` command as we did above, passing its output directly to `c++filt` via the *pipe* operator.

Either way, `nm` has told us that the definition of `engi3891::lab1::increment(int&)` is located at offset `00000000` in `increment.o`: it's the first (and only!) function definition in the object file.

3.3 Compile decrement.cpp

Next, we will compile `decrement.cpp` into an object file:

```
H:\engi3891\lab1>g++ -c decrement.cpp -o decrement.o
```

Again, this produces an object file:

```
H:\engi3891\lab1>dir
<<output>>
```

[1] The Intel 80386, or just "386", is the basis for almost all modern desktop and laptop computers. Although later CPU generations introduced many new features, the core 386 instructions are still used today.

```
H:\engi3891\lab1>file decrement.cpp
<<output>>
H:\engi3891\lab1>file decrement.o
<<output>>
```

Next, **run `nm` against decrement.o and pipe the output through c++filt**, capturing the output.

```
H:\engi3891\lab1><<command>>
<<output>>
```

Question: *What is the offset of* `decrement(int&)` *in the object file?*

Question: *What is different about* `nm`*'s description of decrement.o, as compared with increment.o?*

Question: *Is there any disagreement between the declaration (in decrement.h) and definition of* `decrement()`*?*

We will come back to this mismatch when we try to link the whole program together into an executable file.

3.4 Compile sillyMath.cpp

Compile sillyMath.cpp into an object file.

```
H:\engi3891\lab1><<command>>
```

Show the offsets of the functions defined in `sillyMath.o`.

```
H:\engi3891\lab1><<command>>
<<output>>
```

Question: *What are the offsets of the definitions of the functions* `engi3891::lab1::add(int, unsigned int)` *and* `engi3891::lab1::subtract(int, unsigned int)` *within sillyMath.o?*

We will now use the `objdump` tool to inspect the actual machine instructions that the compiler has translated our source code into:

```
H:\engi3891\lab1>objdump --source sillyMath.o
<<output>>
```

The computer engineers in the class will learn more about what those instructions mean in ENGI 4862. For now, note the meaning of the offset that `nm` told us about: it's the location of our function (more precisely, the instructions that the compiler has translated our function into) within the object file's code section.

Question: *What is the instruction at offset* `1a`*? (**note:** you are not required to interpret this instruction)*

The output from `nm` included two symbols with no offset:

```
U engi3891::lab1::decrement(int&)
U engi3891::lab1::increment(int&)
```

Recall that a symbol of type `T` is one that is defined in the object file[2]. In these lines, the `U` means that the symbols are undefined.

Question: *Consulting sillyMath.cpp, why would these two symbols be undefined in the object file sillyMath.o?*

3.5 Compile main.cpp

Compile `main.cpp` and show the offsets of its symbols:

```
H:\engi3891\lab1><<command>>
H:\engi3891\lab1><<command>>
<<output>>
```

Question: *What symbol is defined in the text segment of main.cpp?*

You may have noticed that the `main` symbol has not been mangled like the others: it's not called `__Z5mainiPPc` or `main(int,char*[])`. This is because `main` is a holdover from the days of C, and name mangling was only introduced in C++ to support *overloading* (a feature that we will learn about soon in lectures).

3.6 Link complete program

We will now use the compiler to attempt to *link* our object files together into a complete program:

```
H:\engi3891\lab1>g++ decrement.o increment.o main.o sillyMath.o -o example
<<output>>
```

Question: *Why do we encounter this error?*

Question: *How can we fix it?*

Apply the fix that you suggest, **recompile any source files you change**, then link the complete program:

```
H:\engi3891\lab1>g++ *.o -o example
```

This time, rather than listing all of our object files, we used `*.o` to mean "every file whose name ends in `.o`". The result is an executable program that we can run:

```
H:\engi3891\lab1>dir
<<output>>
H:\engi3891\lab1>file example.exe
<<output>>
H:\engi3891\lab1>example
<<output>>
```

We can also examine the symbols defined in the executable program, sending the output of `nm` into `c++filt` and the output of `c++filt` into `grep` (a program that lets us pick out certain lines of output, in this case the lines that contain the string "engi3891"):

```
H:\engi3891\lab1>nm example.exe | c++filt | grep engi3891
<<output>>
```

[2] The letter T is used because function definitions are kept in the "text segment" of the object file (and the final program). The segment containing all of the code is called the "text" segment for historical reasons, but it is frequently referred to as the more-intuitive "code segment".

We can also look for all defined symbols:

```
H:\engi3891\lab1>nm example.exe | c++filt | grep " T "
<<output>>
```

This is a very long list! When we link a modern C++ application, the linker links in relevant parts of the standard library (e.g., the definitions of `std::iostream` and `std::string`).

4 Postamble

You should now understand the differences among source, object and executable files and roughly understand the process by which one is turned into another. You should also understand a bit more of the process of linking object files together, particularly whan an "unresolved symbol" error means and how you can fix it.

Lab reports are due one week after the lab, at the beginning of that day's lecture. Unlike the last lab, you will submit this week's lab in one partner's assignments directory on the Subversion repository: you have no further need of the `engi3891/lab0/groupXX` directories. Since there will only be one copy per lab group, please ensure that you have the following line at the very top of your report:

```
# Lab report 1: <<username>> (<<student id>>) and <<username>> (<<student id>>)
```