Learning to Program with Haiku

Lesson 2

Written by DarkWyrm

Our first couple of programs weren't all that useful, but as we go along, you'll find that you can do more and more with your programs. This time around we're going to look at two main concepts: comments, and the different stages the compiler goes through to generate your program, and we'll also learn a little about debugging your code.

Comments are notes that you put into the code. They have many uses, such as clarifying a certain section of code, providing warnings, or temporarily disabling a section of code. See the below example for how they can be used.

```cpp
// This is a one-line comment.
// So is this. Everything after the two forward slashes is considered part of it.

int main(void)
{
    PushTheRedButton();            // This code doesn't work.
    return 1;
}
```

Inherited from C is the multiline comment. They start with /* and end with */ in a way similar to parentheses or curly braces with one difference: you can't put a multiline comment inside another one.

```cpp
/*-----------------------------------------------------------------------
RedButton.cpp

This code is an example of how the compiler will complain if you use
a function that it doesn't recognize.
-----------------------------------------------------------------------*/

// This is a one-line comment.
// So is this. Everything after the two forward slashes is considered part of it.

int main(void)
{
    PushTheRedButton();            // This code doesn't work.
    return 1;
}
```

Now we're going to take a step back and look at the process the compiler goes through when building your program. This is important to understand because there are different kinds of errors that can be made when writing code and knowing something about the process will help you find them more easily.

When a program is built from source code, there are four tools that operate on it before it is an executable: the preprocessor, the compiler, the assembler, and the linker.

Stage 1: The Preprocessor

The preprocessor accepts raw source code as input and does a little massaging before the code is sent to the compiler. It removes comments and handles #include directives by inserting the contents of the included header file into the code. There are other directives which it handles that will be discussed later.

Stage 2: The Compiler

The compiler translates the C++ code into Assembly language. Assembly is much, much closer to the instructions which the computer understands while still being human-readable. It is also much harder to write programs and is specific to the processor for which it is written.

Stage 3: The Assembler

The assembler creates **object code** from the Assembly code created by the compiler and places them in object files which have a .o extension. Object code is the actual machine-executable instructions used by the computer to run your program. It's not quite ready to run, however. In this state the object files used to make your program are a lot like a set of puzzle pieces that are ready to be put together.

Stage 4: The Linker

The linker pieces together your object files along with any libraries that they use into an executable program.

Debugging

By nature of programmers being human, they make mistakes and lots of them. Writing code and debugging go hand-in-hand and often done at the same time. As such, we will be learning about how to debug programs as we learn about writing them.

Bugs come in two types: syntactic and semantic. Syntactic bugs are easy to find because the compiler finds them for us. These are problems like capitalization errors, missing or extra parentheses, and mistyped function names. Semantic errors are often harder to find because they are errors in the logic of perfectly legal code. A semantic error in English would be "The oxygen censor on my car needed to be replaced" – the sentence is perfectly legal and correctly constructed, but the word needed is *sensor*, not *censor*. Examples of semantic errors are extra semicolons in certain places, adding the wrong amount to a number, and making assumptions about the return value of a function.

Let's take a look at a few simplified examples of common errors:

Example 1

Code

```
#include <stdio.h>

int main(void)
{
        return 1;
} }
```

Errors

```
foo.cpp:6: error: expected declaration before '}' token
```

What we have here is an extra curly brace. gcc has given us a fairly cryptic error, as usual, but it has also given us two clues: the file name and the line number. The line number given by gcc and the actual location of the error are not always the same, but in this case they are.

At this point, you might be wondering, "What in the world is a token, genius?" A **token** is any language element. Just as a regular written language has words and punctuation, so do computer languages. Just as two commas in a row are a punctuation error in the English language, having an extra curly brace is a C++ punctuation error.

Example 2

Code

```
/*----------------------------------------------------------------------
RedButton.cpp

/*This code is an example of how the compiler will complain if you use
a function that it doesn't recognize.*/

----------------------------------------------------------------------*/

// This is a one-line comment.
// So is this. Everything after the two forward slashes is considered part of it.

int main(void)
{
    PushTheRedButton();           // This code doesn't work.
    return 1;
}
```

Errors

```
foo.cpp:7: error: expected unqualified-id before '--' token
```

This is an example of how the line number for the error is not the same place as the actual error. The complaint is about the dashes at the end of the multiline comment at the top. It is caused, however by nesting a multiline comment inside another one. The preprocessor removes all comments, so what the compiler sees is this:

```
----------------------------------------------------------------------*/


int main(void)
{
    PushTheRedButton();
    return 1;
}
```

The compiler doesn't know what to do with the dashed line and complains.

Example 3

Code

```c
int Main(void)
{
    return 1;
}
```

Errors

```
/usr/lib/gcc/i486-linux-gnu/4.4.1/../../../../lib/crt1.o: In function `_start':
/build/buildd/eglibc-2.10.1/csu/../sysdeps/i386/elf/start.S:115: undefined
reference to `main'
/tmp/ccv39Cuo.o:(.eh_frame+0x12): undefined reference to `__gxx_personality_v0'
collect2: ld returned 1 exit status
```

This is an error of a different kind. Remember that `main()` needs to be defined in every program? We didn't – we defined `Main()`. The program is otherwise valid, so it compiled just fine, but when the linker attempted to piece it all together, it couldn't find the one required function and threw a hissyfit. Any time you see an error containing `undefined reference to`, you have a linker error.

Resolving `undefined reference` linker errors isn't generally difficult. It usually means one of two things: you forgot to link in a library that you used, or a source code file was accidentally left out when your program was built.