Linking and resolving references

Jonathan Van der Cruysse, Jan Broeckhove
Universiteit Antwerpen

March 9, 2017

1 Introduction

When instructed to compile a *translation unit* (that is, a C/C++ source file) the compiler will process all declarations and definitions in that file, without relating them to some outside context. The output of this step is either a source code file in *assembly language,* or an *object file.*

Object files are files that contain binary machine code; *assembly language* is textual machine code, and is much easier to read than a raw stream of instruction bytes. An *assembler* can be used to convert an assembly language file to an object file, and a *disassembler* converts object files back into assembly language.

Most modern production-quality compilers are configured to do many things at once by default: pre-processing, compilation, assembling and linking. But there's usually a switch or two that lets us control which steps are performed, and which are not.

This document will use `clang` for compiling C files, `clang++` for compiling C++ files, and the GNU binutils for pretty much everything else. `clang` and `clang++` are available for many operating systems, including Linux, Windows and Mac OS X. Alternatively, you can use `gcc` and `g++`, which are present by default on most Linux distributions.

A caveat: turning source code into an executable is a complex process that is dependent on lots of things, including, but not limited to: the processor's instruction set architecture, the operating system, the calling convention, the compiler, and the compiler's flags. Two unrelated computers may produce completely different output, so you shouldn't feel alarmed if your toolchain doesn't produce the exact same output as mine at times.

2 Compiling a C file: `example.c`

Before we can run the linker, we need to compile all source files. Let's start off by familiarizing ourselves with some basic compiler options.

Here's the source file that we'll be working with for now. We're starting off with C source code because linking and compilation is more complicated for C++ files. More on that in section 4.

```c
int muladd(int a, int b, int c) { return a * b + c; }
```

Listing 1: `example.c`

Did you notice that there's no `main` function, and that there are no `#include`s? Good. That's because we're trying to keep this example as simple as possible. Let's try to compile `example.c`. On my machine, `clang` outputs the following:

```
$ clang example.c
/usr/bin/../lib/gcc/x86_64-linux-gnu/5.4.0/../../../x86_64-linux
    -gnu/crt1.o: In function ‘_start’:
(.text+0x20): undefined reference to ‘main’
clang: error: linker command failed with exit code 1 (use -v to
    see invocation)
```

Listing 2: output for `clang example.c`

Well, that's an ugly error message. `clang` means well, though. It's just trying to tell us that we forgot to add a function called `main`. Without one, `clang` can't possibly hope to produce an *executable*, which it tries to do by default.

Fortunately, we don't *really* need a `main` function, because we're not looking to produce an *executable* yet. We just want to compile a single file of C code and produce an *object file* or perhaps a *assembly language* file. For now, the output is not going to be an executable: it clould not be, actually, because the source input file is simply a function definition. We have not provided code that could be turned into something executable.

The executable will be produced later on produced by the linking step. At present, we'd like to omit the linking step, and have `clang` stick to its core task, which is compiling translation units. And that, `clang` can do. We're just going to have to be a little more specific.

2.1 Compiling to assembly language

For starters, let's ask `clang` to emit assembly language for `example.c`.

```
$ clang example.c -S -Os -masm=intel -o example.s
```

Listing 3: compiling `example.c` to assembly language

Let's briefly go over the compiler's flags.

- `-S` instructs `clang` to produce assembly language.
- `-Os` tells `clang` to optimize its output for size, which makes reading the assembly code slightly more bearable. In this case, we mostly want the optimizer to elide function prologues and epilogues, which can only obscure the compiled function body.
- `-masm=intel` makes `clang` emit `x86` or `amd64` assembly language in the Intel dialect, which I find to be more intuitive than the AT&T dialect that `clang` uses by default. If you're not using an Intel or AMD CPU, then you can simply omit this option.
- `-o example.s` requests that `clang` write its output to `example.s`.

If you happen to be working on a machine that uses the `amd64` (a.k.a. 64-bit `x86` or `x86_64`) instruction set architecture, then you'll get a `example.s` file that looks more or less like this.

```
	.text
	.intel_syntax noprefix
	.file	"example.c"
	.globl	muladd
	.type	muladd,@function
muladd:                                 # @muladd
	.cfi_startproc
# BB#0:
	imul	edi, esi
	lea eax, [rdi + rdx]
	ret
.Lfunc_end0:
	.size	muladd, .Lfunc_end0-muladd
	.cfi_endproc

	.ident	"clang version 3.8.0-2ubuntu4 (tags/RELEASE_380/final)"
	.section	".note.GNU-stack","",@progbits
```

Listing 4: `example.s`

This is not an assembly language tutorial, so let's limit ourselves to a brief look at the important bits for the sake of completeness.

- Line 6, and lines 9 to 11 contain an assembly version of the `muladd` function we defined in `example.c`. Line 6 contains a label that defines `muladd`, line 9 performs a multiplication, line 10 performs an addition, and line 11 returns control to the caller.
- Lines 4 and 5 state that label `muladd` is a *global symbol,* and that `muladd` is a *function,* respectively. This is valuable information to the linker, which we'll get to soon.
- The `.text` directive on line 1 starts the text section, which contains machine instructions.

And that's about all there is to it. Almost everything else is metadata.

2.2 Compiling to an object file

You probably already know that CPUs don't consume textual assembly language. They execute binary machine instructions. To build an executable file, we'll have to convert assembly language to machine language. Note that we can't actually create an executable from `example.c` or `example.s`, but we can do the next best thing, which is to produce an object file.

Object files are the binary equivalent to assembly language code. Unsurprisingly, they contain machine instructions instead of assembly language. But they also contain pretty much everything else that was in the assembly language file. In our case, that's the `.text` section, the `muladd` label, and the metadata directives. This will be elaborated on in section 3.3.

We now have two ways to construct an object file. The first is to use the *assembler* to translate the assembly language (content of `example.s`) into binary code (content of `example.o`). The invocation of the assembler tool *as* is straightforward.

```
$ as example.s -o example.o
```

Listing 5: assembling `example.s`

Alternatively, we can have the compiler take care of assembling its output for us. This option is generally preferred, because it's simpler and less error-prone. Moreover, modern compilers like `clang` have integrated assemblers. When an integrated assembler is used, the compiler doesn't have to print textual assembly language, and the assembler doesn't have to parse the compiler's output. So there's a potential performance advantage over using an external assembler, too.

The following command creates an object file directly, without the unnecessary assembly detour. Note the `-c` flag, which instructs `clang` to produce an object file (as opposed to an executable).

```
$ clang example.c -c -Os -o example.o
```

Listing 6: compiling `example.c` to an object file

2.3 Linking C files

We now have an object file – `example.o` – but we can't turn that into an executable, because we don't have a *whole program:* a program where every declaration has a definition. To be precise, only those declarations that are actually used need to have a definition. Additionally, a *whole program* must be a true *program:* it must have an entry point. That is a point in the code where control can be handed over to by the loader to effectively start executing the binary program code. In our case, what have to do is create a new C file that defines a function called `main`.

So let's write some C code that includes a `main` function. And, while we're at it, we can use our brand new `muladd` function in that `main` function.

```
#include <stdio.h>

// 'muladd' is declared here, but it is defined
// in 'example.c'
int muladd(int a, int b, int c);

int main() { printf("2 * 5 + 3 = %d \n", muladd(2, 5, 3)); }
```

`muladd-main.c`

Note that, just like `example.c`, `muladd-main.c` alone is *not* a whole program: function `muladd` is declared, but not defined in `muladd-main.c`. As before, we can compile `muladd-main.c` to an object file.

```
$ clang muladd-main.c -c -Os -o muladd-main.o
```

Listing 8: compiling `muladd-main.c` to an object file

Now we have two object files: `example.o` and `muladd-main.o`. But what we really wanted was an executable file, which is the sum of the information in both files: `main` is defined in `muladd-main.o`, and `muladd` is defined in `example.o`. What we need now, is a tool to link those two files together: a *linker*.

On Unix-based systems, the system linker program `ld` can be used to link object files into executables. Let's see if we can link `muladd-main.o` and `example.o`.

```
$ ld example.o muladd-main.o
ld: warning: cannot find entry symbol _start; defaulting to
    00000000004000b0
muladd-main.o: In function ‘main’:
muladd-main.c:(.text+0x21): undefined reference to ‘printf’
```

Listing 9: output for `ld example.o muladd-main.o`

That failed spectacularly, because we still don't have a whole program: `printf` is *declared* in `stdio.h`, but not *defined*. In fact, `printf` is defined by the C standard library. We have to link our object files with the system's standard C library (located in some system directory and having *libc* in it's name) in order to produce an executable.

The good news is that `clang` will do that for us. Just type `clang` instead of `ld`, and it'll run the system linker for us with all the right options.

```
$ clang example.o muladd-main.o -o myprog
```

Listing 10: linking `example.o` and `muladd-main.o`

This yields an executable called which we'll call `myprog` (if you do not provide a name, it defaults to *a.out*). Go ahead and run it.

```
$ ./myprog
2 * 5 + 3 = 13
```

Listing 11: output for `myprog`

2.4 Separate compilation

Compiling each translation unit individually, and then linking them all together, is called *separate compilation*.

For our simple example, we could just as easily have invoked `clang` once: the command below will produce an equivalent binary with far less effort to the programmer.

```
$ clang example.c muladd-main.c -Os -o a.out
```

Listing 12: compiling and linking `example.c` and `muladd-main.c` directly

Note that the call above will still perform separate compilation under the hood: the source files will be compiled individually, and then linked together. If you are ever in doubt about the exact workings of `clang` or `gcc`, then you can always use the `-###` flag. If `-###` is passed, then the compiler won't actually compile anything. Instead, it will explain which commands it would run to process the input files.

Be prepared for verbose output, though. `clang` and `gcc` use lots of options by default, and they'll list them all if you use `-###`. I've taken the liberty of eliding everything but the most essential parts from `clang`'s output below, in the interest of brevity and readability.

```
$ clang example.c muladd-main.c -o a.out -###
clang version 3.8.0-2ubuntu4 (tags/RELEASE_380/final)
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
"/usr/lib/llvm-3.8/bin/clang" "-cc1" ... "-o" "/tmp/example-14
    e39f.o" "-x" "c" "example.c"
"/usr/lib/llvm-3.8/bin/clang" "-cc1" ... "-o" "/tmp/muladd-main-3
    a96b0.o" "-x" "c" "muladd-main.c"
"/usr/bin/ld" ... "/tmp/example-14e39f.o" "/tmp/muladd-main-3
    a96b0.o" ...
```

Listing 13: filtered output for `clang -###`

As you can see, `clang` first calls itself twice – once for each input file – and then calls `ld` to link `clang`'s output into an executable.

2.4.1 Taking advantage of separate compilation

There is merit to understanding separate compilation. Compiling many C/C++ translation units can take a long time – just try building a project that has lots of source files – and big projects often consist of multiple programs that share a large amount of translation units. In that case, one can take advantage of separate compilation.

For example, suppose that `example.c` was used not by one, but by two programs. Naively calling the compiler once for each input program will compile `example.c` *twice.*

```
$ clang example.c muladd-main.c -o a.out
$ clang example.c muladd-main2.c -o a2.out
```

Listing 14: compiling and linking twice

We don't have to compile `example.c` twice, though. We can compile it once, emit an object file, and then link that object file with the other source files twice. That is exactly what the commands below do.

```
$ clang example.c -c -o example.o
$ clang example.o muladd-main.c -o a.out
$ clang example.o muladd-main2.c -o a2.out
```

Listing 15: compiling once and linking twice

Moreover, separate compilation can also be helpful when *incremental* changes are made: not all source files have to be re-compiled when only some of them are changed. Only the affected files need to be re-compiled. Then, the program is linked again, combining the old and new object files.

Naively invoking the compiler on a list of source files will not recognize or re-use unchanged object files, and can be unnecessarily slow when incremental changes are made. It is almost always more efficient to exercise more fine-grained control over the compilation and linking process.

2.4.2 Build tools

Efficiently managing the compilation and linking process of tens or hundreds of source files can be a bit of a kludge.

Fortunately, the process can be automated by using a *build tool*, such as GNU `make` or CMake. The former requires more manual configuration, the latter will automatically generate a Makefile as input to GNU `make`.

These tools may take some effort to set up initially, but once that is over, they can speed up the edit-compile-debug cycle significantly. Moreover, the build tool configuration file are usually less platform dependent and easier to maintain than explicit scripts detailing all the build steps.

3 How linkers operate

Intuitively, one can think of a linker as a program that “concatenates” multiple object files: the sections of machine code they contain are quite literally merged by joining them end-to-end.

However, linkers are also responsible for making sure that code which originated from different object files can interact properly. This can be achieved by *patching* specific parts of the machine code.

Before we examine how linkers patch machine code, we should take a look at the underlying reason for this patching process, which is rooted in assembly language.

3.1 Labels and branches

The basic tool for control-flow transfer in assembly language is the *branch* or *jump:* when a machine encounters a branch or jump instruction, it may transfer control to another instruction at an address which is specified by the branch or jump instruction.

To make writing assembly language easier, assemblers offer a feature called *labels:* user-defined names that are given to specific instructions. These names can then be used as the target of branch or jump instructions, and so transfer control to the instruction associated with the label.

Consider the following pseudo-assembly.

```
    jump lbl
    ...
lbl:
    add r1, r2, r3
```

Listing 16: pseudo-assembly example: input

It’s important to note that labels are nothing more than syntactic sugar for specific addresses. When the assembler is instructed to assemble the code into an object file or executable, it will have to translate `jump lbl` as a jump instruction that transfers control to an *address,* rather than a *label.*

This is usually implemented by introducing two distinct passes. The first pass assembles instructions into machine code. Branch or jump instructions that transfer control to a label are replaced by instructions that transfer control to some arbitrary address, typically 0.

Instructions that took a label are remembered by the assembler and so are the addresses of instructions that are marked with labels. After applying the first pass, our hypothetical assembly code will look like this:

```
0x10: jump 0            # target was "lbl"
...   ...
0xf3: add r1, r2, r3    # label was "lbl"
```

Listing 17: pseudo-assembly example: after first pass

The second pass resolves the references to labels. It revisits all branch and jump instructions that referred to a label and replaces their branch and jump targets by the explicit address of this label. Label addresses are known at this point, because the assembler has traversed the entire file.

After patching the machine instructions, our example will look like the listing below. Note that no trace of label `lbl` is left at this point. Only a jump to an address remains.

```
0x10: jump 0xf3
...   ...
0xf3: add r1, r2, r3
```

Listing 18: pseudo-assembly example: after second pass

3.2 Symbols and relocations

The technique described in the previous section relies on two assumptions.

- All labels that in use, are defined somewhere.
- The definition of a label can be found in the same file as its users.

We obviously can't compromise on the first requirement. The user must always define a label if it is used.

The second requirement, on the other hand, is violated by C/C++'s separate compilation. C/C++ functions and globals are tagged by labels when they are compiled to assembly language, as can indeed be observed on line 6 of listing 4 (see page 3).

The second requirement must be relaxed to enable separate compilation. This can be achieved by delaying the second pass from the previous section. Labels are preserved as *symbols* in the object file, and references to those labels are preserved as *relocation records* in the *relocation table*.

Symbols are addresses that are relative to the start of the section in which they are defined. *Relocation records* are pairs of symbols and pointers into the object code (specified a range of bytes) to the addresses that must be patched of fixed up to refer to the explicit, final address associated with the symbol.

Each object file contains various sections such as code, data, etc. When the linker runs, it first concatenates all sections of the same type and then assign final, run time addresses to each section and the symbols defined in that section. Next the linker will use the relocation table to resolve all references to symbols by binding them to run time addresses.

3.2.1 A pseudo-assembly example

Suppose that the example from listing 16 was split across two files: `user.s` uses label `lbl`, `declaration.s` declares it.

```
# user.s
.globl lbl
.text
    jump lbl
    ...
# ============
# declaration.s
.globl lbl
.text
lbl:
    add r1, r2, r3
    ...
```

Listing 19: separately assembled pseudo-assembly example

In the listing above, the `.globl lbl` directives tell the assembler to retain `lbl` as a *global* or *public* symbol. Such symbols can be referred to from other object files. The `.globl` directive is also present in `clang`'s output, which can be observed on line 4 of listing 4.

The assembler will also replace `jump lbl` by `jump 0`, and include a relocation record that tells the linker to replace that `0` by the eventual run time address of symbol `lbl`.

The linker runs, it will notice that `user.o` contains an *undefined* external reference to the symbol `lbl`, and that `declaration.o` *defines* `lbl`. It will react to this situation by matching both `lbl` symbols, and patching the `jump 0` instruction to refer to `lbl`'s run time address in the concatenated executable.

When a symbol cannot be resolved, i.e. it is *undefined* in all object files that refer to it, the linker will issue an error. Listings 2 and 9 are examples of these dreaded linker errors.

Sometimes a symbol can be multiply defined, e.g. when you have used the same function name in two different source files, both containing a definition of the function. Some linkers will flag this as an error, others will issue a warning and then use the last definition that they have encountered to resolve references to the function. Some linkers have flags to specify a strategy with respect to this situation.

3.3 Examining symbols

The `nm` tool can be used to list symbols from object files. Let's try that out on the `example.o` file from our previous example.To get additional details about `example.o`'s symbols, `--format=sysv` can be used instead of `--format=bsd`.

```
$ nm example.o --format=bsd
0000000000000000 T muladd
```

Listing 20: listing all symbols in `example.o`

This table tells us that `example.o` contains exactly one symbol: `muladd`. The two most important bits of information about `muladd` is that it is located at offset 0, and that its *class* is 'T'. Every symbol has a class, which tells the linker a little bit more about what kind of symbol it is dealing with. According to `man nm`, class 'T' means that "the symbol is in the text (code) section."

We can also analyze `muladd-main.o`.

```
$ nm muladd-main.o --format=bsd
0000000000000000 T main
U muladd
U printf
```

Listing 21: listing all symbols in `muladd-main.o`

As one might expect from analogy with `example.o`'s symbol table, `muladd-main.o` defines `main` in the text section. Interestingly, `muladd-main.o` also defines two symbols with class 'U'. This is the class of undefined symbols, and both `muladd` and `printf` are indeed defined elsewhere. The linker will unify these undefined symbols with 'T' symbols from `example.o` and the C standard library, respectively.

4 Linking C++ code

C++ adds complexity to C in many respects; the link phase is no exception in this regard. In this section, we'll take a brief look at two C++ implementation details: *name mangling* and *weak symbols*.

4.1 Name mangling

Function overloading is a C++ feature that C lacks. In a nutshell,*function overloading* is the ability to create multiple methods of the same name with different call signatures. Calls to an overloaded function will run a specific implementation of that function appropriate to the context of the call, allowing one function name to be associated with different tasks depending on the calling context.

Function overloading often proves itself to be a useful language feature, but linkers resolve references based on symbol names. So compiling the source file below, *as if it were C code,* would yield a compiler error if it were a single source file because the `muladd` gets redefined. If, still working *as if it were C code,*, it is separated into two source files then the compiler accepts this (it does not know about the redefinition because each translation unit gets compiled separately). But, even if the compiler doesn't mind, the the linker does. It will tell us – in an error message – that the label `muladd` is defined more than once.

```
int muladd(int a, int b, int c) { return a * b + c; }

double muladd(double a, double b, double c) { return a * b + c;
    }
```

Listing 22: `overloading`

Enter C++ and *name mangling.* Instead of creating two symbols named `muladd`, the C++ compiler will create two different symbols based on the *signatures* of the `muladd` functions in `overloading.cpp`. Let's see which symbol names `clang++` generates.

```
$ clang++ overloading.cpp -c -o overloading.o
$ nm overloading.o --format=bsd
0000000000000020 T _Z6muladdddd
0000000000000000 T _Z6muladdiii
```

Listing 23: mangled names

The mangled names can be dissected as follows: `_Z` is a common prefix for all mangled names, `6` denotes the number of characters in identifier `muladd` and `ddd` identifies a function that has three parameters of type `double`. Analogously, `iii` means that a function has three `int` parameters.

We don't have to demangle names manually, though. The `c++filt` tool converts mangled names back into C++ function signatures.

```
$ c++filt _Z6muladdddd _Z6muladdiii
muladd(double, double, double)
muladd(int, int, int)
```

Listing 24: using `c++filt`

As usual with C/C++, mangled names are both compiler-specific and platform-specific. No specific name mangling scheme is mandated by the C++ standard, so other compilers can and will produce entirely different symbol names. On Linux, `clang++` uses the same name mangling scheme as `g++`. On Windows, `clang++` uses the Visual C++ name mangling for compatibility.

Name mangling stays in the background most of the time. It is entirely invisible during successful builds, but some linkers may report mangled names instead of function signatures in their error messages.

Though name mangling is requisite to function overloading, it's usefulness extends beyond that. Consider a situation in which you build an executable and link multiple libraries that refer to one another. Assume a function signature gets changed in one of these libraries and the library is recompiled. All references to the function in the other libraries now have a mismatch between the actual arguments to the function provided in the calling context and the formal parameters expected by the call. A C linker will not detect this and the result will be program crashes or, worse still, erroneous output. A C++ linker will flag this as an error because the name mangled symbol referred to in the calling context does not match the name mangled symbol in the recompiled library and an "unresolved symbol" situation ensues.

4.2 Weak symbols

Traditionally, C code requires that every declaration can be matched to *exactly one* definition. C++ relaxes this rule to support a few language constructs. The first such language construct is *templates.*

4.2.1 Template instantiations

Templates definitions are often included in headers, so that every translation unit that instantiates a template definition will have its own copy of the template's body.

Thus, to make the `#include`-based template instantiation model viable, the linker must allow duplicate definitions. This can be accomplished by marking template instantiations *weak symbols*.

Whenever the linker encounters more than one definition for a *weak symbol*, it just picks one based on the assumption that these definitions are *semantically equivalent*, and makes all references to that symbol point to this arbitrarily chosen definition.

Weak symbols can be observed by explicitly instantiating a template, and then running `nm` on the resulting object file. Consider the template definition below.

```cpp
template <typename T>
T muladd(T a, T b, T c)
{
    return a * b + c;
}

// Explicitly instantiate 'muladd<T>' for 'int's.
template int muladd<int>(int, int, int);
```

Listing 25: `template.cpp`

We can now compile `template.cpp` and list its symbols. Note the `--demangle` flag, which makes `nm` apply `c++filt` to symbol names. This is especially useful for mangled template instance names, which are even harder to read than regular mangled names.

```
$ clang++ template.cpp -c -o template.o
$ nm template.o --format=bsd
0000000000000000 W _Z6muladdIiET_S0_S0_S0_
$ nm template.o --format=bsd --demangle
0000000000000000 W int muladd<int>(int, int, int)
```

Listing 26: compiling and listing all symbol names in `template.o`

As was expected, symbol `_Z6muladdIiET_S0_S0_S0_` has class 'W', which makes it a weak symbol.

4.2.2 `inline` and member functions

Functions that are annotated with the `inline` keyword also have weak linkage. Member functions defined within a class definition are implicitly inline. An example of both has been included below.

```cpp
#include <iostream>

class MulAddHelpers
{
public:
    static int muladd(int a, int b, int c) { return a * b + c; }
};

inline int muladd(int a, int b, int c) { return a * b + c; }

int main()
{
    std::cout << MulAddHelpers::muladd(1, 2, 3) << std::endl;
    std::cout << muladd(1, 2, 3) << std::endl;
}
```

Listing 27: `inline-functions.cpp`

Verifying that `MulAddHelpers::muladd` and `muladd` are compiled as weak symbols is left as an exercise to the reader.

Inline functions and templates have similar use-cases. They both allow for the creation of header-only libraries, and make it easier for the compiler to create highly optimized code.

Inline functions can be used to guide optimization. Compilers that encounter a call to a small function defined in the current translation unit – as is the case with inline functions – may apply the *inlining* optimization: it replaces the call by a copy of the function's body. Hence the name and keyword 'inline.' Note, however, that `inline` will not, in any way, *force* the compiler to perform the inlining optimization – it merely *enables* the compiler to do so, if it so pleases.

That being said, modern compilers are often able to inline functions across translation units, at link-time. This particular type of optimization – *link-time optimization* – at least partially obsoletes an advantage of inline functions; it is no longer necessary to make functions inline from a performance perspective, provided that the build process is set-up correctly.

The takeaway here is that greater performance gains can be accomplished by updating the build process to use link-time optimization. Use inline functions when an `#include`-based compilation model makes sense, or for truly tiny functions that would otherwise have been implemented as preprocessor macros, and stick to normal functions otherwise.

4.2.3 The perils of weak symbols

Weak symbols enable a number of features, but they are brittle. A careless or malicious programmer can abuse weak linkage to produce programs whose meaning depends entirely on the linker's inner mechanisms.

Linkers cannot verify whether two weak symbols are equivalent; they cannot issue an error or warning when two semantically distinct versions of a weak symbol are provided. So linkers pick exactly one definition of a weak symbol when multiple definitions are available.

The following example illustrates this. Suppose that we have two C++ source files – `incompatible-template.cpp` and `template-main.cpp` – in addition to `template.cpp` from listing 25. Their definitions are given below.

```cpp
template <typename T>
T muladd(T a, T b, T c)
{
    // Template definition with incorrect semantics.
    return a + b + c;
}

// Explicitly instantiate 'muladd<T>' for 'int's.
template int muladd<int>(int, int, int);
```

Listing 28: `incompatible-template.cpp`

```cpp
#include <iostream>

template <typename T>
T muladd(T a, T b, T c);

int main() { std::cout << muladd<int>(1, 2, 3) << std::endl; }
```

Listing 29: `template-main.cpp`

Compiling and linking `template-main.cpp`, `incompatible-template.cpp` and `template.cpp` is illegal according to C++'s *one definition rule,* but present-day compilers and linkers fail to diagnose the problem, even when all conventional warnings have been enabled.

The resulting executable's behavior is entirely undefined, and may depend on trivial parameters, such as the order of command-line arguments. I observed the following on my machine.

```
$ clang++ template-main.cpp template.cpp incompatible-template.
    cpp -o a.out -Wall -Wextra -pedantic
$ ./a.out
5
$ clang++ template-main.cpp incompatible-template.cpp template.
    cpp -o a.out -Wall -Wextra -pedantic
$ ./a.out
6
```

Listing 30: undefined behavior due to weak linkage

This above is clearly a pathological example, but incompatible weak symbol definitions can also arise from different preprocessor macro expansions in templates or inline functions defined in included header files.

5 Debugging programs

Finding bugs in programs is often hard. One approach is to strategically insert statements that print information about the program's state. This is often called `printf` debugging, due to the use of the `printf` function in C.

Anyone who has used this technique knows that it's not ideal. Inserting a print statement requires the program to be compiled and linked. It often takes many edit-build-run cycles to get the placement of those print statements right.

There is a much better way to debug programs. *Debuggers* are programs that will help you find bugs by suspending running programs, and examining their state. Unlike `printf` debugging, with the debugger-based technique there is no need to modify source text. Ionly requires only special build options.

5.1 Emitting debug information

To debug programs, a debugger relies on special *debug information:* extra data that relates the compiled version of the program to its source code. Compilers are able to generate this debug information, and place it in the object files they produce. The information is then copied into the executable by the linker.

When the debugger runs, it will look for debug information, and use that to tie the machine instructions it executes back to the source code statements.

Getting clang to produce debug information isn't all that hard. All it takes is the -g flag. Alternatively, you can use the -g3 flag, which includes extra information such as macro definitions.[1]

Let's compile template-main.cpp and incompatible-template.cpp with debug information.

```
$ clang++ template-main.cpp incompatible-template.cpp -O0 -g -o
    a.out
```

Listing 31: compiling with debug information

Note that we used the -O0 optimization level. This disables all non-essential compiler optimizations, which may make the debugging experience less frustrating. Compiler optimizations can sometimes twist the program's structure in unexpected ways, and you don't want to deal with the compiler's shenanigans when you're trying to debug some code.

If -O0 degrades performance too much, then you can probably get away with -O1 or even -O2. If you're using gcc, then you should use gcc's special debugging optimization level: -Og.[2] Try to avoid aggressive optimizations (-O3) when debugging, though. They can be pretty invasive.

5.2 Using gdb, the GNU debugger

We now have an executable with debug information, which means we can now use a debugger to examine it. We'll be using the GNU debugger (gdb), a popular command-line debugger. Let's launch gdb for executable a.out.

```
$ gdb a.out
... (the sexy legal bit) ...
Reading symbols from a.out...done.
(gdb)
```

Listing 32: running gdb

We can now start issuing commands. gdb ships with an incredible variety of commands, so let's just go over the basic ones.

The simplest command is run, which will start a.out. Most commands in gdb can be abbreviated, and r is short for run.

[1]Consult the gcc docs for more information about clang/gcc debug information options: https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html

[2]Documentation on gcc's optimization options can be found at https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

```
(gdb) run
Starting program: ..../gobelijn/main/cpp/linking/a.out
6
[Inferior 1 (process 9017) exited normally]
(gdb) r
Starting program: .../gobelijn/main/cpp/linking/a.out
6
[Inferior 1 (process 9137) exited normally]
```

Listing 33: running `a.out` in `gdb`

As you might imagine, the `run` command isn't very exciting on its own. Running a program is hardly a novel concept, but we can pair the `run` command with other commands. For example, `break` can be used to ask `gdb` to suspend the program's execution at a so-called *breakpoint.* Let's insert a breakpoint on the first statement in the `main` function, and then run `a.out` again.

```
(gdb) break main
Breakpoint 1 at 0x4008d7: file template-main.cpp, line 8.
(gdb) r
Starting program: .../gobelijn/main/cpp/linking/a.out

Breakpoint 1, main () at template-main.cpp:8
8       std::cout << muladd<int>(1, 2, 3) << std::endl;
```

Listing 34: running `a.out` in `gdb`, with a breakpoint

The program has been suspended at the breakpoint, which presents us with an ideal opportunity to order `gdb` around for a bit. We'll use three elementary commands, which often prove themselves useful when debugging a program: `list`, `print` and `backtrace`.

`list` can be used to take a peek at the source code in the vicinity of the breakpoint the program is currently paused on.

```
(gdb) list
3   template<typename T>
4   T muladd(T a, T b, T c);
5
6   int main()
7   {
8       std::cout << muladd<int>(1, 2, 3) << std::endl;
9   }
```

Listing 35: using `list` to print relevant source code

`print` evaluates an expression, and prints the result. We can use it to run `muladd<int>` without having to continue the program's execution.

```
(gdb) print muladd<int>(1, 2, 3)
$1 = 6
(gdb) print 1 * 2 + 3
$2 = 5
```

Listing 36: using the `print` command

`gdb` tells us that `muladd<int>(1, 2, 3)` evaluates to `6`, but we expect it to evaluate to `5`. To find out more about what's going on in `muladd`, we should set another breakpoint, and then resume `a.out`'s execution with `continue` (abbreviated form: `c`).

```
(gdb) break muladd<int>
Breakpoint 2 at 0x4008ad: file incompatible-template.cpp, line
    6.
(gdb) c
Continuing.

Breakpoint 2, muladd<int> (a=1, b=2, c=3) at incompatible-
    template.cpp:6
6       return a + b + c;
```

Listing 37: adding another breakpoint

It seems that we have uncovered the source of the "bug:" the definition of `muladd` in `incompatible-template.cpp` simply adds its operands together.

We're more or less done debugging the program now. We'll first use the `backtrace` command to print a stack trace, and then let the program run to completion. `gdb` can be closed by pressing the Ctrl+D keys, or by using the `quit` command, which can be abbreviated as `q`.

```
(gdb) backtrace
#0  muladd<int> (a=1, b=2, c=3) at incompatible-template.cpp:6
#1  0x00000000004008dc in main () at template-main.cpp:8
(gdb) c
Continuing.
6
[Inferior 1 (process 9676) exited normally]
(gdb) q
```

Listing 38: running `backtrace`, followed by `c` and `q`

5.3 Other debuggers

`gdb` is a great debugger, but it is not the only wonderful debugging tool out there. There's no shortage of compelling alternatives, such as `lldb` and the Visual Studio debugger. These two will be discussed briefly now.

5.3.1 `lldb`, the LLVM debugger

`lldb`[3] describes itself as a "next generation, high-performance debugger," but it's mostly used as a clean-slate re-imagining of `gdb`. It features a command-line interface with a highly regular set of commands that have clearly-defined, predictable behavior. Some `gdb` commands can have bizarre behavior for complex programs, and `lldb` tries to keep the magic to a minimum.

In the end, whether you pick `gdb` or `lldb` is mostly a matter of taste, especially from a user's perspective. Both debuggers are definitely worth trying.[4]

5.3.2 Visual Studio's debugger

Windows users who have installed Visual Studio can use the integrated debugger to squash bugs in their code. This debugger is wrapped in a user-friendly GUI, and that makes it a lot easier to set breakpoints in Visual Studio than in `gdb` or `lldb`, and variables can be inspected by hovering over them with the mouse pointer.

On the other hand, `gdb` and `lldb` have rich sets of commands, which the Visual Studio debugger lacks. There's a bit of an expressiveness/intuitiveness trade-off between debuggers.

Furthermore, the Visual Studio debugger is not unique. Other C/C++ IDEs tend to offer similar debugging tools, though the Visual Studio debugging workflow is often touted as the best of its kind. That being said, whether you find any of this to be the case or not is entirely up to you. Some programmers prefer console-based debuggers, others prefer IDEs.

6 Wrapping up

Hopefully this document has given you at least a passing understanding of how linking works, and how the linker and compiler interact.

[3] `lldb` home page: http://lldb.llvm.org/

[4] An `lldb` tutorial for `gdb` users can be found at `http://lldb.llvm.org/tutorial.html`

To summarize:

- Turning source files into an executable may seem simple at first, but it is often a daunting task that must be carefully managed. Build systems such as GNU `make` and CMake often prove themselves to be invaluable tools when dealing with the complexity of compiling and linking a medium-sized or large program.
- Missing definitions are only detected at link-time due to C/C++'s separate compilation paradigm. The linker errors that ensue can be notoriously hard to understand, and a solid understanding of how linking works can prove invaluable when diagnosing these problems.
- C++ language features that are implemented by weak linkage can ease the process of developing complex, high-performance applications, but special care must be taken to avoid conflicting definitions.
- Debuggers are awesome tools, and there's a great line-up of debuggers out there: `gdb`, `lldb` and the Visual Studio debugger are all production-quality tools. Familiarize yourself with at least one of them.

The source code for all C/C++ files referred to in this document can be found in the `main/cpp/linking` folder of the public Gobelijn repository.