Beginner's Guide to Linkers

This article is intended to help C & C++ programmers understand the essentials of what the linker does. I've explained this to a number of colleagues over the years, so I decided it was time to write it down so that it's more widely available (and so that I don't have to explain it again). [Updated March 2009 to include more information on the pecularities of linking on Windows, plus some clarification on the one definition rule.]

A typical example of what triggers this explanation is when I help someone who has a link error like:

```
g++ -o test1 test1a.o test1b.o
test1a.o(.text+0x18): In function `main':
: undefined reference to `findmax(int, int)'
collect2: ld returned 1 exit status
```

If your reaction to this is 'almost certainly missing `extern "C"`' then you probably already know everything in this article.

Table of Contents

- Naming of Parts: What's in a C File
- What The C Compiler Does
    - Dissecting An Object File
- What The Linker Does: Part 1
    - Duplicate Symbols
- What The Operating System Does
- What The Linker Does: Part 2
    - Static Libraries
    - Shared Libraries
    - Windows DLLs
        - Exporting Symbols
        - `.LIB` and Other Library-Related Files
        - Importing Symbols
        - Circular Dependencies
- Adding C++ To The Picture
    - Function Overloading & Name Mangling
    - Initialization of Statics
    - Templates
- Dynamically Loaded Libraries
    - Interaction with C++ Features
- More Details

Naming of Parts: What's in a C File

This section is a quick reminder of the different parts of a C file. If everything in the sample C file listing below makes sense to you, you can probably skip to the next section.

The first division to understand is between declarations and definitions. A *definition* associates a name with an implementation of that name, which could be either data or code:

- A definition of a variable induces the compiler to reserve some space for that variable, and possibly fill that space with a particular value.

- A definition of a function induces the compiler to generate code for that function.

A *declaration* tells the C compiler that a definition of something (with a particular name) exists elsewhere in the program, probably in a different C file. (Note that a definition also counts as a declaration—it's a declaration that also happens to fill in the particular "elsewhere").

For variables, the definitions split into two sorts:

- *global variables*, which exist for the whole lifetime of the program ("static extent"), and which are usually accessible in lots of different functions
- *local variables*, which only exist while a particular function is being executed ("local extent") and are only accessible within that function

To be clear, by "accessible" we mean "can be referred to using the name associated with the variable by its definition".

There are a couple of special cases where things aren't so immediately obvious:

- `static` local variables are actually global variables, because they exist for the lifetime of the program, even though they're only visible inside a single function
- likewise `static` global variables also count as global variables, even though they can only be accessed by the functions in the particular file where they were defined

While we're on the subject of the "`static`" keyword, it's also worth pointing out that making a *function* static just narrows down the number of places that are able to refer to that function by name (specifically, to other functions in the same file).

For both global and local variable definitions, we can also make a distinction between whether the variable is initialized or not—that is, whether the space associated with the particular name is pre-filled with a particular value.

Finally, we can store information in memory that is dynamically allocated using `malloc` or `new`. There is no way to refer to the space allocated by name, so we have to use pointers instead—a named variable (the pointer) holds the address of the unnamed piece of memory. This piece of memory can also be deallocated with `free` or `delete`, so the space is referred to as having "dynamic extent".

Let's put all that together:

	Code	**Data**				
		Global		**Local**		**Dynamic**
		Initialized	**Uninitialized**	**Initialized**	**Uninitialized**	
Declaration	`int fn(int x);`	`extern int x;`	`extern int x;`	N/A	N/A	N/A
Definition	`int fn(int x) { ... }`	`int x = 1;` (at file scope)	`int x;` (at file scope)	`int x = 1;` (at function scope)	`int x;` (at function scope)	`(int* p = malloc(sizeof(int));)`

An easier way to follow this is probably just to look at this sample program:

```
/* This is the definition of a uninitialized global variable */
int x_global_uninit;

/* This is the definition of a initialized global variable */
int x_global_init = 1;
```

```
/* This is the definition of a uninitialized global variable, albeit
 * one that can only be accessed by name in this C file */
static int y_global_uninit;

/* This is the definition of a initialized global variable, albeit
 * one that can only be accessed by name in this C file */
static int y_global_init = 2;

/* This is a declaration of a global variable that exists somewhere
 * else in the program */
extern int z_global;

/* This is a declaration of a function that exists somewhere else in
 * the program (you can add "extern" beforehand if you like, but it's
 * not needed) */
int fn_a(int x, int y);

/* This is a definition of a function, but because it is marked as
 * static, it can only be referred to by name in this C file alone */
static int fn_b(int x)
{
  return x+1;
}

/* This is a definition of a function. */
/* The function parameter counts as a local variable */
int fn_c(int x_local)
{
  /* This is the definition of an uninitialized local variable */
  int y_local_uninit;
  /* This is the definition of an initialized local variable */
  int y_local_init = 3;

  /* Code that refers to local and global variables and other
   * functions by name */
  x_global_uninit = fn_a(x_local, x_global_init);
  y_local_uninit = fn_a(x_local, y_local_init);
  y_local_uninit += fn_b(z_global);
  return (y_global_uninit + y_local_uninit);
}
```

What The C Compiler Does

The C compiler's job is to convert a C file from text that the human can (usually) understand, into stuff that the computer can understand. This output by the compiler as an *object file*. On UNIX platforms these object files normally have a `.o` suffix; on Windows they have a `.obj` suffix. The contents of an object file are essentially two kinds of things:

- *code*, corresponding to definitions of functions in the C file
- *data*, corresponding to definitions of **global** variables in the C file (for an initialized global variable, the initial value of the variable also has to be stored in the object file).

Instances of either of these kinds of things will have names associated with them—the names of the variables or functions whose definitions generated them.

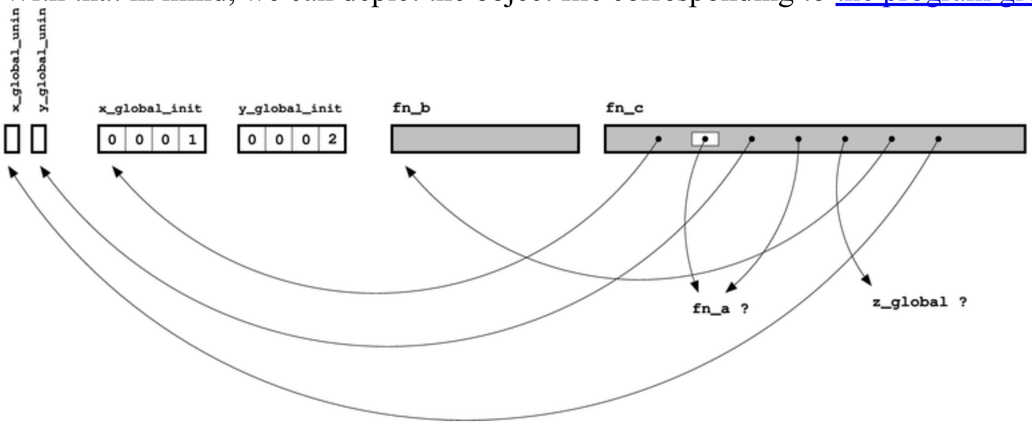
Object code is the sequence of (suitably encoded) machine instructions that correspond to the C instructions that the programmer has written—all of those `if`s and `while`s and even `goto`s. All of these instructions need to manipulate information of some sort, and that information needs to be kept somewhere—that's the job of the variables. The code can also refer to other bits of code—specifically, to other C functions in the program.

Wherever the code refers to a variable or function, the compiler only allows this if it has previously seen a declaration for that variable or function—the declaration is a promise that a definition exists somewhere else in the whole program.

The job of the linker is to make good on these promises, but in the meanwhile what does the compiler do with all of these promises when it is generating an object file?

Basically, the compiler leaves a blank. The blank (a "reference") has a name associated to it, but the value corresponding to that name is not yet known.

With that in mind, we can depict the object file corresponding to the program given above like this:



Dissecting An Object File

We've kept everything at a high level so far; it's useful to see how this actually works in practice. The key tool for this is the command **nm**, which gives information about the symbols in an object file on UNIX platforms. On Windows, the **dumpbin** command with the /symbols option is roughly equivalent; there is also a Windows port of the GNU binutils tools which includes an nm.exe.

Let's have a look at what nm gives on the object file produced from the C file above:

```
Symbols from c_parts.o:

Name                  Value   Class        Type         Size     Line  Section

fn_a                |        |   U  |            NOTYPE|        |     |*UND*
z_global            |        |   U  |            NOTYPE|        |     |*UND*
fn_b                |00000000|   t  |              FUNC|00000009|     |.text
x_global_init       |00000000|   D  |            OBJECT|00000004|     |.data
y_global_uninit     |00000000|   b  |            OBJECT|00000004|     |.bss
x_global_uninit     |00000004|   C  |            OBJECT|00000004|     |*COM*
y_global_init       |00000004|   d  |            OBJECT|00000004|     |.data
fn_c                |00000009|   T  |              FUNC|00000055|     |.text
```

The output on different platforms can vary a bit (check the man pages to find out more on a particular version), but the key information given is the class of each symbol, and its size (when available). The class can have a number of different values:

- A class of **U** indicates an undefined reference, one of the "blanks" mentioned previously. For this object, there are two: "`fn_a`" and "`z_global`". (Some versions of `nm` may also print out a *section*, which will be ***UND*** or **UNDEF** in this case)
- A class of **t** or **T** indicates where code is defined; the different classes indicate whether the function is local to this file (**t**) or not (**T**)—i.e. whether the function was originally declared with `static`. Again, some systems may also show a section, something like **.text**
- A class of **d** or **D** indicates an initialized global variable, and again the particular class indicates whether the variable is local (**d**) or not (**D**). If there's a section, it will be something like **.data**
- For an uninitialized global variable, we get **b** if it's static/local, and **B** or **C** when it's not. The section in this case will probably be something like **.bss** or ***COM***.

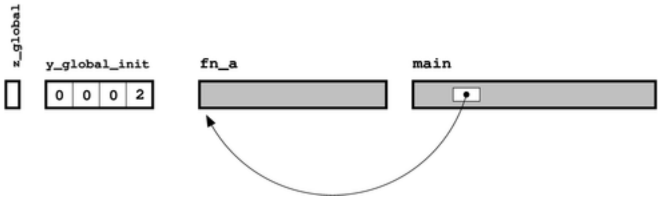
We may also get some symbols that weren't part of the original input C file; we'll ignore these as they're typically part of the compiler's nefarious internal mechanisms for getting your program to link.

What The Linker Does: Part 1

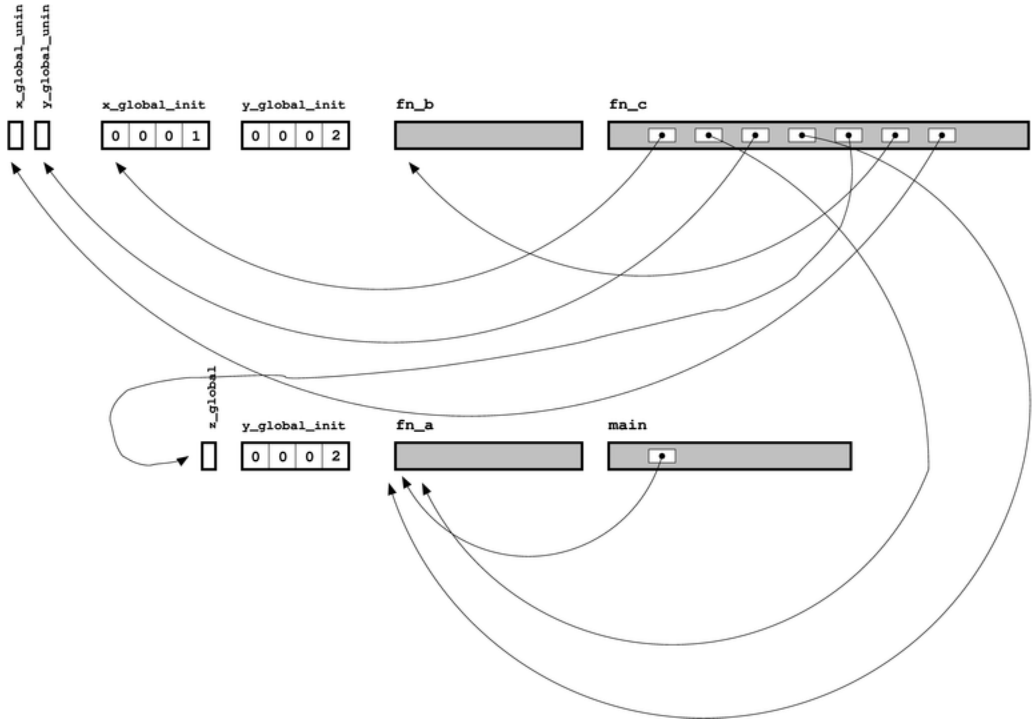
We mentioned earlier that a declaration of a function or a variable is a promise to the C compiler that somewhere else in the program is a definition for that function or variable, and that the linker's jobs is to make good on that promise. With the diagram of an object file in front of us, we can also describe this as "filling in the blanks".

To illustrate this, let's have a companion C file to the one given previously:

```
/* Initialized global variable */
int z_global = 11;
/* Second global named y_global_init, but they are both static */
static int y_global_init = 2;
/* Declaration of another global variable */
extern int x_global_init;

int fn_a(int x, int y)
{
  return(x+y);
}

int main(int argc, char *argv[])
{
  const char *message = "Hello, world";

  return fn_a(11,12);
}
```



With these two diagrams, we can see that all of the dots can joined up (if they couldn't be, then the linker would emit an error message). Every thing has its place, and every place has its thing, and the linker can fill in all of the blanks as shown (on a UNIX system, the linker is typically invoked with `ld`).



As for object files, we can use `nm` to examine the resulting executable file:

```
Symbols from sample1.exe:

Name                  Value   Class         Type         Size     Line  Section

_Jv_RegisterClasses |        |    w  |            NOTYPE|        |     |*UND*
__gmon_start__      |        |    w  |            NOTYPE|        |     |*UND*
__libc_start_main@@GLIBC_2.0|        |    U  |              FUNC|000001ad|     |*UND*
_init               |08048254|    T  |              FUNC|        |     |.init
_start              |080482c0|    T  |              FUNC|        |     |.text
__do_global_dtors_aux|080482f0|    t  |              FUNC|        |     |.text
frame_dummy         |08048320|    t  |              FUNC|        |     |.text
fn_b                |08048348|    t  |              FUNC|00000009|     |.text
fn_c                |08048351|    T  |              FUNC|00000055|     |.text
fn_a                |080483a8|    T  |              FUNC|0000000b|     |.text
main                |080483b3|    T  |              FUNC|0000002c|     |.text
__libc_csu_fini     |080483e0|    T  |              FUNC|00000005|     |.text
__libc_csu_init     |080483f0|    T  |              FUNC|00000055|     |.text
__do_global_ctors_aux|08048450|    t  |              FUNC|        |     |.text
_fini               |08048478|    T  |              FUNC|        |     |.fini
_fp_hw              |08048494|    R  |            OBJECT|00000004|     |.rodata
_IO_stdin_used      |08048498|    R  |            OBJECT|00000004|     |.rodata
__FRAME_END__       |080484ac|    r  |            OBJECT|        |     |.eh_frame
__CTOR_LIST__       |080494b0|    d  |            OBJECT|        |     |.ctors
__init_array_end    |080494b0|    d  |            NOTYPE|        |     |.ctors
__init_array_start  |080494b0|    d  |            NOTYPE|        |     |.ctors
__CTOR_END__        |080494b4|    d  |            OBJECT|        |     |.ctors
__DTOR_LIST__       |080494b8|    d  |            OBJECT|        |     |.dtors
__DTOR_END__        |080494bc|    d  |            OBJECT|        |     |.dtors
__JCR_END__         |080494c0|    d  |            OBJECT|        |     |.jcr
__JCR_LIST__        |080494c0|    d  |            OBJECT|        |     |.jcr
_DYNAMIC            |080494c4|    d  |            OBJECT|        |     |.dynamic
_GLOBAL_OFFSET_TABLE_|08049598|    d  |            OBJECT|        |     |.got.plt
__data_start        |080495ac|    D  |            NOTYPE|        |     |.data
```

```
data_start          |080495ac|   W  |            NOTYPE|        |     |.data
__dso_handle        |080495b0|   D  |            OBJECT|        |     |.data
p.5826              |080495b4|   d  |            OBJECT|        |     |.data
x_global_init       |080495b8|   D  |            OBJECT|00000004|     |.data
y_global_init       |080495bc|   d  |            OBJECT|00000004|     |.data
z_global            |080495c0|   D  |            OBJECT|00000004|     |.data
y_global_init       |080495c4|   d  |            OBJECT|00000004|     |.data
__bss_start         |080495c8|   A  |            NOTYPE|        |     |*ABS*
_edata              |080495c8|   A  |            NOTYPE|        |     |*ABS*
completed.5828      |080495c8|   b  |            OBJECT|00000001|     |.bss
y_global_uninit     |080495cc|   b  |            OBJECT|00000004|     |.bss
x_global_uninit     |080495d0|   B  |            OBJECT|00000004|     |.bss
_end                |080495d4|   A  |            NOTYPE|        |     |*ABS*
```

This has all of the symbols from the two objects, and all of the undefined references have vanished. The symbols have also all been reordered so that similar types of things are together, and there are a few added extras to help the operating system deal with the whole thing as an executable program.

There's also a fair number of complicating details cluttering up the output, but if you filter out anything starting with an underscore it gets a lot simpler.

Duplicate Symbols

The previous section mentioned that if the linker cannot find a definition for a symbol to join to references to that symbol, then it will give an error message. So what happens if there are *two* definitions for a symbol when it comes to link time?

In C++, the situation is straightforward. The language has a constraint known as the *one definition rule*, which says that there has to be exactly one definition for a symbol when it comes to link time, no more and no less. (The relevant section of the C++ standard is 3.2, which also mentions some exceptions that we'll come to later on.)

For C, things are slightly less clear. There has to be exactly one definition of any functions or initialized global variables, but the definition of an uninitialized global variable can be treated as a *tentative definition*. C then allows (or at least does not forbid) different source files to have tentative definitions for the same object.

However, linkers also have to cope with other programming languages than just C and C++, and the one definition rule isn't always appropriate for them. For example, the normal model for Fortran code is effectively to have a copy of each global variable in every file that references it; the linker is required to fold duplicates by picking one of the copies (the largest version, if they are different sizes) and throw away the rest. (This model is sometimes known as the "common model" of linking, after the Fortran `COMMON` keyword.)

As a result, it's actually quite common for UNIX linkers *not* to complain about duplicate definitions of symbols—at least, not when the duplicate symbol is an uninitialized global variable (this is sometimes known as the "relaxed ref/def model" of linking). If this worries you (and it probably should), check the documentation for your compiler linker—there may well be a `--work-properly` option that tightens up the behavior. For example, for the GNU toolchain the `-fno-common` option to the compiler forces it to put uninitialized variables into the BSS segment rather than generating these *common blocks*.

What The Operating System Does

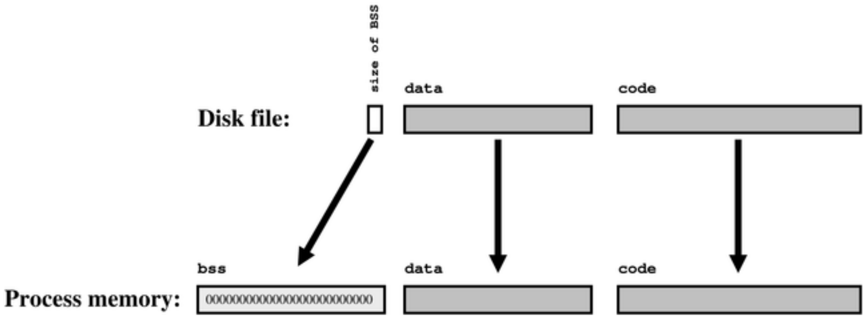
Now that the linker has produced an executable program with all of the references to symbols joined up to suitable definitions of those symbols, we need to pause briefly to understand what the operating system does when you run the program.

Running the program obviously involves executing the machine code, so the operating system clearly has to transfer the machine code from the executable file on the hard disk into the computer's memory, where the CPU can get at it. This chunk of the program's memory is known as the *code segment* or *text segment*.

Code is nothing without data, so all of the global variables need to have some space in the computer's memory too. However, there's a difference between initialized and uninitialized global variables. Initialized variables have particular values that need to be used to begin with, and these values are stored in the object files and in the executable file. When the program is started, the OS copies these values into the program's memory in the *data segment*.

For uninitialized variables, the OS can assume that they all just start with the initial value 0, so there's no need to copy any values. This chunk of memory, that gets initialized to 0, is known as the *bss segment*.

This means that space can be saved in the executable file on disk; the initial values of initialized variables have to be stored in the file, but for the uninitialized variables we just need a count of how much space is needed for them.

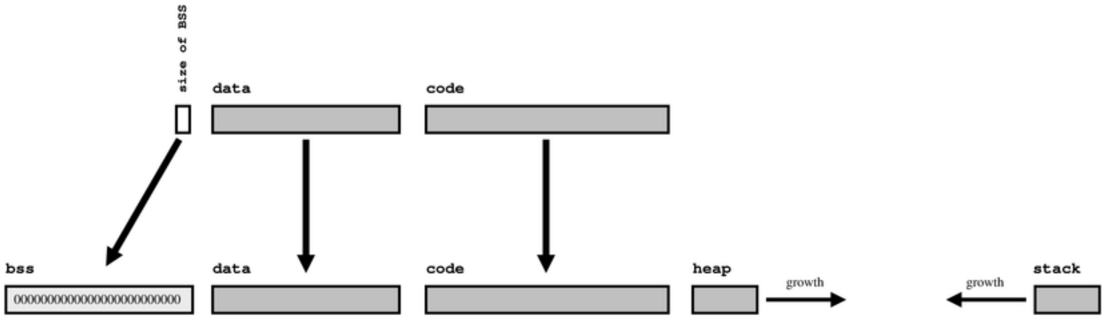


You may have noticed that all of the discussion of object files and linkers so far has only talked about global variables; there's been no mention of the local variables and dynamically allocated memory [mentioned earlier](#).

These pieces of data don't need any linker involvement, because their lifetime only occurs when the program is running—long after the linker has finished its business. However, for the sake of completeness, we can quickly point out here that:

- local variables are allocated on a piece of memory known as the *stack*, which grows and shrinks as different functions are called and complete
- dynamically allocated memory is taken from an area known as the *heap*, and the `malloc` function keeps track of where all the available space in this area is.

We can add in these chunks of memory to complete our picture of what the memory space of a running process looks like. Because both the heap and the stack can change size as the program runs, it's quite common to arrange matters so that the stack grows in one direction while the heap grows in the other. That way, the program will only run out of memory when they meet in the middle (and at that point, the memory space really will be full).



What The Linker Does: Part 2

Now that we've covered the very basics of the operation of a linker, we can plough on to describe some more complicating details—roughly in the order that these features were historically added to linkers.

The main observation that affected the function of the linker is this: if lots of different programs need to do the same sorts of things (write output to the screen, read files from the hard disk, etc), then it clearly makes sense to commonize this code in one place and have lots of different programs use it.

This is perfectly feasible to do by just using the same object files when linking different programs, but it makes life much easier if whole collections of related object files are kept together in one easily accessible place: a *library*

(Technical aside: This section completely skips a major feature of the linker: *relocation*. Different programs will be different sizes, so when the shared library gets mapped into the address space of different programs, it will be at different addresses. This in turn means that all of the functions and variables in the library are in different places. Now, if all of the ways of referring to addresses are relative ("the value +1020 bytes from here") rather than absolute ("the value at 0x102218BF") this is less of a problem, but this isn't always possible. If not, then all of these absolute addresses need to have a suitable offset added to them —this is relocation. I'm not going to mention this topic again, though, because it's almost always invisible to the C/C++ programmer—it's very rare that a linking issue is because of relocation problems)

Static Libraries

The most basic incarnation of a library is a *static library*. The previous section mentioned that you could share code by just reusing object files; it turns out that static libraries really aren't much more sophisticated than that.

On UNIX systems, the command to produce a static library is normally **`ar`**, and the library file that it produces typically has a `.a` extension. These library files are normally also prefixed with "`lib`" and passed to the linker with a "-l" option followed by the name of the library, without prefix or extension (so "`-lfred`" will pick up "`libfred.a`").

(Historically, a program called `ranlib` also used to be needed for static libraries, in order to build an index of symbols at the start of the library. Nowadays the `ar` tool tends to do this itself.)

On Windows, static libraries have a `.LIB` extension and are produced by the `LIB` tool, but this can be confusing as the same extension is also used for an "*import library*", which just holds a list of the things available in a DLL—see the section on Windows DLLs.

As the linker trundles through its collection of object files to be joined together, it builds a list of the symbols it hasn't been able to resolve yet. When all of the explicitly specified objects are done with, the linker now has another place to look for the symbols that are left on this unresolved list—in the library. If the unresolved symbol is defined in one of the objects in the library, then that object is added in, exactly as if the user had given it on the command line in the first place, and the link continues.

Note the granularity of what gets pulled in from the library: if some particular symbol's definition is needed, the *whole object* that contains that symbol's definition is included. This means that the process can be one step forwards, one step back—the newly added object may resolve one undefined reference, but it may well come with a whole collection of new undefined references of its own for the linker to resolve.

Another important detail to note is the *order* of events; the libraries are consulted only when then the normal linking is done, and they are processed *in order*, left to right. This means that if an object pulled in from a library late in the link line needs a symbol from a library earlier in the link line, the linker won't automatically find it.

An example should help to make this clearer; let's suppose we have the following object files, and a link line that pulls in `a.o`, `b.o`, `-lx` and `-ly`.

File	`a.o`	`b.o`	`libx.a`			`liby.a`		

Object	`a.o`	`b.o`	`x1.o`	`x2.o`	`x3.o`	`y1.o`	`y2.o`	`y3.o`
Definitions	`a1, a2, a3`	`b1, b2`	`x11, x12, x13`	`x21, x22, x23`	`x31, x32`	`y11, y12`	`y21, y22`	`y31, y32`
Undefined references	`b2, x12`	`a3, y22`	`x23, y12`	`y11`		`y21`		`x31`

Once the linker has processed `a.o` and `b.o`, it will have resolved the references to `b2` and `a3`, leaving `x12` and `y22` as still undefined. At this point, the linker checks the first library `libx.a` for these symbols, and finds that it can pull in `x1.o` to satisfy the `x12` reference; however, doing so also adds `x23` and `y12` to the list of undefined references (so the list is now `y22`, `x23` and `y12`).

The linker is still dealing with `libx.a`, so the `x23` reference is easily satisfied, by also pulling in `x2.o` from `libx.a`. However, this also adds `y11` to the list of undefineds (which is now `y22`, `y12` and `y11`). None of these can be resolved further using `libx.a`, so the linker moves on to `liby.a`.

Here, the same sort of process applies and the linker will pull in both of `y1.o` and `y2.o`. The first of these adds a reference to `y21`, but since `y2.o` is being pulled in anyway, that reference is easily resolved. The net of this process is that all of the undefined references have been resolved, and some but not all of the objects in the libraries have been included into the final executable.

Notice that the situation would have been a little different if (say) `b.o` also had a reference to `y32`. If this had been the case, the linking of `libx.a` would have worked the same, but the processing of `liby.a` would also have pulled in `y3.o`. Pulling in this object would have added `x31` to the list of unresolved symbols, and the link would have failed—by this stage the linker has already finished with `libx.a` and would not find the definition (in `x3.o`) for this symbol.

(By the way, this example has a cyclic dependency between the two libraries `libx.a` and `liby.a`; this is typically a Bad Thing, particularly on Windows)

Shared Libraries

For popular libraries like the C standard library (normally `libc`), having a static library has an obvious disadvantage—every executable program has a copy of the same code. This can take up a lot of unnecessary disk space, if every single executable file has a copy of `printf` and `fopen` and suchlike.

A slightly less obvious disadvantage is that once a program has been statically linked, the code in it is fixed forever. If someone finds and fixes a bug in `printf`, then every program has to be linked again in order to pick up the fixed code.

To get around these and other problems, *shared libraries* were introduced (normally indicated by a `.so` extension, or `.dll` on Windows machines and `.dylib` on Mac OS X). For these kinds of libraries, the normal command line linker doesn't necessarily join up all of the dots. Instead, the regular linker takes a kind of "IOU" note, and defers the payment of that note until the moment when the program is actually run.

What this boils down to is this: if the linker finds that the definition for a particular symbol is in a shared library, then it doesn't include the definition of that symbol in the final executable. Instead, the linker records the name of symbol and which library it is supposed to come from in the executable file instead.

When the program is run, the operating system arranges that these remaining bits of linking are done "just in time" for the program to run. Before the `main` function is run, a smaller version of the linker (often called `ld.so`) goes through these promissory notes and does the last stage of the link there and then—pulling in the code of the library and joining up all of the dots.

This means that none of the executable files have a copy of the code for `printf`. If a new, fixed, version of `printf` is available, it can be slipped in just by changing `libc.so`—it'll get picked up the next time any program runs.

There's another big difference with how shared libraries work compared to static libraries, and that shows up in the granularity of the link. If a particular symbol is pulled in from a particular shared library (say `printf` in `libc.so`), then the *whole* of that shared library is mapped into the address space of the program. This is very different from the behavior of a static library, where only the particular objects that held undefined symbols got pulled in.

Put another way, a shared library is itself produced as a result of a run of the linker (rather than just forming a big pile of objects like `ar` does), with references between objects in the same library getting resolved. Once again, `nm` is a useful tool for illustrating this: for the example libraries above it will produce sets of results for the individual object files when run on a static version of the library, but for the shared version of the library, `liby.so` has only `x31` as an undefined symbol. Also, for the library-ordering example at the end of the previous section, there wouldn't be a problem: adding a reference to `y32` into `b.c` would make no difference, as all of the contents of `y3.o` and `x3.o` are already pulled in anyway.

> As an aside, another useful tool is **`ldd`**; on Unix platforms this shows the set of shared libraries that an executable (or a shared library) depends on, together with an indication of where those libraries are likely to be found. For the program to run successfully, the loader needs to be able to find all of these libraries, together with all of their dependencies in turn. (Typically, the loader looks for libraries in the list of directories held in the `LD_LIBRARY_PATH` environment variable.)

```
/usr/bin:ldd xeyes
        linux-gate.so.1 =>  (0xb7efa000)
        libXext.so.6 => /usr/lib/libXext.so.6 (0xb7edb000)
        libXmu.so.6 => /usr/lib/libXmu.so.6 (0xb7ec6000)
        libXt.so.6 => /usr/lib/libXt.so.6 (0xb7e77000)
        libX11.so.6 => /usr/lib/libX11.so.6 (0xb7d93000)
        libSM.so.6 => /usr/lib/libSM.so.6 (0xb7d8b000)
        libICE.so.6 => /usr/lib/libICE.so.6 (0xb7d74000)
        libm.so.6 => /lib/libm.so.6 (0xb7d4e000)
        libc.so.6 => /lib/libc.so.6 (0xb7c05000)
        libXau.so.6 => /usr/lib/libXau.so.6 (0xb7c01000)
        libxcb-xlib.so.0 => /usr/lib/libxcb-xlib.so.0 (0xb7bff000)
        libxcb.so.1 => /usr/lib/libxcb.so.1 (0xb7be8000)
        libdl.so.2 => /lib/libdl.so.2 (0xb7be4000)
        /lib/ld-linux.so.2 (0xb7efb000)
        libXdmcp.so.6 => /usr/lib/libXdmcp.so.6 (0xb7bdf000)
```

The reason for this larger granularity is because modern operating systems are clever enough that you can save more than just the duplicate disk space that happens with static libraries; different running processes that use the same shared library can also share the code segment (but not the data/bss segments—two different processes could be in different places for their `strtok` after all). In order to do this, the whole library has to be mapped in one go, so that the internal references all line up to the same places—if one process pulled in `a.o` and `c.o` and another pulled in `b.o` and `c.o`, there wouldn't be any commonality for the OS to leverage.

Windows DLLs

Although the general principles of shared libraries are roughly similar on Unix platforms and Windows, that are a few details that can catch out the unwary.

Exporting Symbols

The most major difference between the two is that symbols are not automatically *exported* by Windows libraries. On Unix, all of the symbols from all of the object files that were linked into the shared library are visible to users of the library. On Windows, the programmer has to explicitly choose to make particular symbols visible—i.e. to export them.

There are three ways to export a symbol from a Windows DLL (and all three ways can be mixed together in the same library).

- In the source code, declare the symbol as `__declspec(dllexport)`, thusly:

  ```
  __declspec(dllexport) int my_exported_function(int x, double y);
  ```

- On the invocation of the linker, use the `/export:`*`symbol_to_export`* option to `LINK.EXE`.

  ```
  LINK.EXE /dll /export:my_exported_function
  ```

- Get the linker to pull in a *module definition* (`.DEF`) file (by using the `/DEF:`*`def_file`* linker option), and in that file include an `EXPORTS` section that contains the symbols you want to export.

  ```
  EXPORTS
    my_exported_function
    my_other_exported_function
  ```

Once C++ is added in to the mix, the first of these options is the easiest because the compiler takes care of the name mangling for you.

`.LIB` and Other Library-Related Files

This neatly leads on to the second complication with Windows libraries: the information about exported symbols that the linker needs to join things up is not held in the DLL itself. Instead, this information is held in a corresponding `.LIB` file.

The `.LIB` file associated with a DLL describes what (exported) symbols are present in the DLL, together with their locations. Any other binary that uses the DLL needs to look in the `.LIB` file so that it can join up the symbols correctly.

To confuse things, the `.LIB` extension is also used for static libraries.

In fact, there are a wide variety of different files that can be relevant for Windows libraries. As well as the `.LIB` file and the (optional) `.DEF` file mentioned in the previous section, you might see all of the following files associated with your Windows library.

- Link output files:
  - *`library`*`.DLL`: The library code itself; this is needed (at run-time) by any executable that uses the library.
  - *`library`*`.LIB`: An "import library" file which describes what symbols are where in the output DLL. This file is only produced if the DLL exports some symbols; if no symbols are exported, there is no point in having the `.LIB` file. This file is needed at link-time by anything that uses this library.
  - *`library`*`.EXP`: An "export file" for the library being linked, which is needed when linking binaries with circular dependencies.
  - *`library`*`.ILK`: If the `/INCREMENTAL` option was specified to the linker so that incremental linking is enabled, this file holds the status of the incremental linking. Needed for any future incremental linking of this library.
  - *`library`*`.PDB`: If the `/DEBUG` option was specified to the linker, this file is a *program database* containing debugging information for the library.
  - *`library`*`.MAP`: If the `/MAP` option was specified to the linker, this file holds a description of the internal layout of the library.
- Link input files:

- *library*.LIB: An "import library" file which describes what symbols are where in any other DLLs that are needed by the thing being linked.
- *library*.LIB: A static library file which contains a collection of object files that are needed by the thing being linked. Note the ambiguous use of the .LIB extension.
- *library*.DEF: A "module definition" file which allows control of various details of the linked library, including the export of symbols.
- *library*.EXP: An "export file" for the library being linked, which can indicate that a previous run of LIB.EXE for the library has already created the .LIB file for the library. Relevant when linking binaries with circular dependencies.
- *library*.ILK: Incremental linking status file; see above.
- *library*.RES: Resource file that contains information about the various GUI widgets that the executable uses; these are included in the final binary file.

This is contrast to Unix, where most of the information held in these extra files is (usually) just included in the library itself.

Importing Symbols

As well as requiring DLLs to explicitly declare which symbols they export, Windows also allows binaries that use library code to explicitly declare which symbols they *import*. This is optional, but gives a speed optimization due to some historical features of 16-bit windows.

To do this, declare the symbol as __declspec(dllimport) in the source code, thusly:

```
__declspec(dllimport) int function_from_some_dll(int x, double y);
__declspec(dllimport) extern int global_var_from_some_dll;
```

It's normal good practice in C to have a single declaration for any function or global variable, held in a header file. This leads to a bit of a conundrum: the code in the DLL that holds the definition of the function/variable needs to *export* the symbol, but any code outside the DLL needs to *import* the symbol.

A common way round this is to use a preprocessor macro in the header file.

```
#ifdef EXPORTING_XYZ_DLL_SYMS
#define XYZ_LINKAGE __declspec(dllexport)
#else
#define XYZ_LINKAGE __declspec(dllimport)
#endif

XYZ_LINKAGE int xyz_exported_function(int x);
XYZ_LINKAGE extern int xyz_exported_variable;
```

The C file in the DLL which defines the function and variable ensures that the preprocessor variable EXPORTING_XYZ_DLL_SYMS is #defined before it includes this header file, and so does an export of the symbols. Any other code that pulls in this header file doesn't define the symbol and so indicates an import of the symbols.

Circular Dependencies

One final complication with DLLs is that Windows is stricter than Unix in requiring every symbol to have a resolution at link time. On Unix, it's possible to link a shared library that contains an unresolved symbol that the linker has never seen; in this situation, any other code that pulls in this shared library must provide that symbol, or the program will fail to load. Windows doesn't allow this sort of laxity.

In most systems this isn't a problem. Executables rely on high-level libraries, the high-level libraries rely on lower-level libraries, and everything gets linked in the opposite order—low-level libraries first, then higher-level libraries, and finally the executables that rely on it all.

However, if there are circular dependencies between binaries, then things are trickier. If `X.DLL` needs a symbol from `Y.DLL`, and `Y.DLL` needs a symbol from `X.DLL`, then there is a chicken-and-egg problem: whichever library is linked first won't be able to find all of its symbols.

Windows does provide a way around this, roughly as follows.

- First, fake a link of library `X`. Run `LIB.EXE` (not `LINK.EXE`) to produce an `X.LIB` file that is the same as would have been produced by `LINK.EXE`. No `X.DLL` file is produced, but a `X.EXP` file does get emitted.
- Link library `Y` as normal; this pulls in the `X.LIB` file from the previous step, and outputs both a `Y.DLL` and a `Y.LIB` file.
- Finally link library `X` properly. This is almost the same as normal, but it additionally includes the `X.EXP` file created in the first step. As normal, this link will pull in the `Y.LIB` file from the previous step and will create a `X.DLL` file. Unlike normal, the link will skip the process of creating an `X.LIB` file, because there one already there from the first step (which is what the `.EXP` file indicates).

Of course, a better idea is usually to re-organize the libraries so that there aren't any circular dependencies….

Adding C++ To The Picture

C++ provides a number of extra features over and above what's available in C, and a number of these features interact with the operation of the linker. This wasn't originally the case—the first C++ implementations came as a front end to a C compiler, so the back end of the linker didn't need to be changed—but as time went on, sufficiently sophisticated features were added that the linker had to be enhanced to support them.

Function Overloading & Name Mangling

The first change that C++ allows is the ability to overload a function, so there can be different versions of the same named functions, differing in the types that the function accepts (the function's *signature*):

```
int max(int x, int y)
{
  if (x>y) return x;
  else return y;
}
float max(float x, float y)
{
  if (x>y) return x;
  else return y;
}
double max(double x, double y)
{
  if (x>y) return x;
  else return y;
}
```

This obviously gives a problem for the linker: when some other code refers to `max`, which one does it mean?

The solution that was adopted for this is called *name mangling*, because all of the information about the function signature is mangled into a textual form, and that becomes the actual name of the symbol as seen by the linker. Different signature functions get mangled to different names, so the uniqueness problem goes away.

I'm not going to go into details of the schemes used (which vary from platform to platform anyway), but a quick look at the object file corresponding to the code above gives some hints (remember, `nm` is your friend!):

```
Symbols from fn_overload.o:

Name                  Value   Class        Type         Size     Line  Section

__gxx_personality_v0|        |   U  |            NOTYPE|        |     |*UND*
_Z3maxii            |00000000|   T  |              FUNC|00000021|     |.text
_Z3maxff            |00000022|   T  |              FUNC|00000029|     |.text
_Z3maxdd            |0000004c|   T  |              FUNC|00000041|     |.text
```

Here we can see that our three functions called `max` all get different names in the object files, and we can make a fairly shrewd guess that the two letters after the "max" are encoding the types of the parameters—"i" for `int`, "f" for `float` and "d" for `double` (things get a lot more complex when classes, namespaces, templates and overloaded operators get added into the mangling mix, though!).

It's also worth noting that there will normally be some way of converting between the user-visible names for things (the *demangled* names) and the linker-visible names for things (the *mangled* names). This might be a separate program (e.g. `c++filt`) or a command-line option (e.g. `--demangle` as an option to GNU nm), which gives results like:

```
Symbols from fn_overload.o:

Name                  Value   Class        Type         Size     Line  Section

__gxx_personality_v0|        |   U  |            NOTYPE|        |     |*UND*
max(int, int)            |00000000|   T  |              FUNC|00000021|     |.text
max(float, float)            |00000022|   T  |              FUNC|00000029|     |.text
max(double, double)            |0000004c|   T  |              FUNC|00000041|     |.text
```

The area where this mangling scheme most commonly trips people up is when C and C++ code is intermingled. All of the symbols produced by the C++ compiler are mangled; all of the symbols produced by the C compiler are just as they appear in the source file. To get around this, the C++ language allows you to put **`extern "C"`** around the declaration & definition of a function. This basically tells the C++ compiler that this particular name should not be mangled—either because it's the definition of a C++ function that some C code needs to call, or because it's a C function that some C++ code needs to call.

For the example given right at the start of this page, it's easy to see that there's a good chance someone has forgotten this `extern "C"` declaration in their link of C and C++ together.

```
g++ -o test1 test1a.o test1b.o
test1a.o(.text+0x18): In function `main':
: undefined reference to `findmax(int, int)'
collect2: ld returned 1 exit status
```

The big hint here is that the error message includes a function signature—it's not just complaining about plain old `findmax` missing. In other words, the C++ code is actually looking for something like "`_Z7findmaxii`" but only finding "`findmax`", and so it fails to link.

By the way, note that an `extern "C"` linkage declaration is ignored for member functions (7.5.4 of the C++ standard).

Initialization of Statics

The next feature that C++ has over C that affects the linker is the ability to have object *constructors*. A constructor is a piece of code that sets up the contents of an object; as such it is conceptually equivalent to an initializer value for a variable but with the key practical difference that it involves arbitrary pieces of code.

Recall from an earlier section that a global variable can start off with a particular value. In C, constructing the initial value of such a global variable is easy: the particular value is just copied from the data segment of the executable file into the relevant place in the memory for the soon-to-be-running program.

In C++, the construction process is allowed to be much more complicated than just copying in a fixed value; all of the code in the various constructors for the class hierarchy has to be run, before the program itself starts running properly.

To deal with this, the compiler includes some extra information in the object files for each C++ file; specifically, the list of constructors that need to be called for this particular file. At link time, the linker combines all of these individual lists into one big list, and includes code that goes through the list one by one, calling all of these global object constructors.

Note that the *order* in which all of these constructors for global objects get called is not defined—it's entirely at the mercy of what the linker chooses to do. (See Scott Meyers' Effective C++ for more details—Item 47 in the second edition, Item 4 in the third edition).

We can hunt down these lists by once again using `nm`. Consider the following C++ file:

```
class Fred {
private:
  int x;
  int y;
public:
  Fred() : x(1), y(2) {}
  Fred(int z) : x(z), y(3) {}
};

Fred theFred;
Fred theOtherFred(55);
```

For this code, the (demangled) output of `nm` gives:

```
Symbols from global_obj.o:

Name                  Value   Class        Type         Size     Line  Section

__gxx_personality_v0|        |   U  |            NOTYPE|        |     |*UND*
__static_initialization_and_destruction_0(int, int)|00000000|   t  |              FUNC|00006
Fred::Fred(int)        |00000000|   W  |              FUNC|00000017|     |.text._ZN4FredC1Ei
Fred::Fred()        |00000000|   W  |              FUNC|00000018|     |.text._ZN4FredC1Ev
theFred             |00000000|   B  |            OBJECT|00000008|     |.bss
theOtherFred        |00000008|   B  |            OBJECT|00000008|     |.bss
global constructors keyed to theFred  |0000003a|   t  |              FUNC|0000001a|     |.te
```

There are various things here, but the one we're interested in is the two entries with class as **W** (which indicates a "weak" symbol) and with section names like "`.gnu.linkonce.t.`*`stuff`*". These are the markers for global object constructors, and we can see that the corresponding "Name" fields look sensible—one for each of the two constructors used.

Templates

In an earlier section, we gave an example of three different versions of a `max` function, each of which took different types of argument. However, we can see that the lines of source code for these three functions are absolutely identical, and it seems a shame to have to copy and paste identical code.

C++ introduces the idea of *templates* to allow code like this to be written once and for all. We can create a header file `max_template.h` with the single unique code for `max`:

```
template <class T>
T max(T x, T y)
{
  if (x>y) return x;
  else return y;
}
```

and include this header file in C++ code to use the templated function:

```
#include "max_template.h"

int main()
{
  int a=1;
  int b=2;
  int c;
  c = max(a,b);  // Compiler automatically figures out that max<int>(int,int) is needed
  double x = 1.1;
  float y = 2.2;
  double z;
  z = max<double>(x,y); // Compiler can't resolve, so force use of max(double,double)
  return 0;
}
```

This C++ file uses both `max<int>(int,int)` and `max<double>(double,double)`, but a different C++ file might use different instantiations of the template—say `max<float>(float,float)` or even `max<MyFloatingPointClass>(MyFloatingPointClass,MyFloatingPointClass)`.

Each of these different instantiations of the template involves different actual machine code, so by the time that the program is finally linked, the compiler and linker need to make sure that every instantiation of the template that is used has code included into the program (and no unused template instantiations are included to bloat the program size).

So how do they do this? There are normally two ways of arranging this: by folding duplicate instantiations, or by deferring instantiation until link time (I like to refer to these as the sane way and the Sun way).

For the duplicate instantiation approach, each object file contains the code for all of the templates that it uses. For the particular example C++ file above, the contents of the object file are:

```
Symbols from max_template.o:

Name                  Value   Class        Type         Size     Line  Section

__gxx_personality_v0|        |   U  |            NOTYPE|        |     |*UND*
double max<double>(double, double)   |00000000|   W  |              FUNC|00000041|     |.tex
int max<int>(int, int)   |00000000|   W  |              FUNC|00000021|     |.text._Z3maxIiET
main                |00000000|   T  |              FUNC|00000073|     |.text
```

and we can see that both `max<int>(int,int)` and `max<double>(double,double)` are present.

These definitions are listed as *weak symbols*, and this means that when the linker produces the final executable program, it can throw away all but one of these duplicate definitions (and if it's feeling generous, it can check that all the duplicate definitions actually look like they are the same code). The most significant downside of this approach is that all of the individual object files take up much more room on the hard disk.

The other approach (which is used by the Solaris C++ compiler suite) is to include none of the template definitions in the object files, but instead to leave them all as undefined symbols. When it comes to link time, the linker can collect together all of the undefined symbols that actually correspond to template instantiations, and go and generate the machine code for these instantiations there and then.

This saves space in the individual object files, but has the disadvantage that the linker needs to keep track of where the header file containing the source code, and needs to be able to invoke the C++ compiler at link time (which may slow down the link).

Dynamically Loaded Libraries

The last feature that we'll talk about on this page is the dynamic loading of shared libraries. A previous section described how using shared libraries means that the final link is deferred until the moment when the program is run. On modern systems, it's possible to defer linking to even later than that.

This is done with a pair of system calls, `dlopen` and `dlsym` (the rough Windows equivalents of these are called `LoadLibrary` and `GetProcAddress`). The first of these takes the name of a shared library and loads it into the address space of the running process. Of course, this extra library may itself have undefined symbols, so this call to `dlopen` may also trigger the loading of other shared libraries.

The `dlopen` also allows the choice of whether to resolve all of these references at the instant that the library is loaded (`RTLD_NOW`), or one by one as each undefined reference is hit (`RTLD_LAZY`). The first way means that the `dlopen` call takes much longer, but the second way involves the slight risk that sometime later the program will discover that there is an undefined reference that can't be resolved—at which point, the program will be terminated.

Of course, there's no way for a symbol from the dynamically loaded library to have a name. However, as ever with programming problems, this is easily solved by adding an extra level of indirection—in this case, by using a pointer to the space for the symbol, rather than referring to it by name. The call `dlsym` takes a string parameter that gives the name of the symbol to be found, and returns a pointer to its location (or `NULL` if it can't be found).

Interaction with C++ Features

This dynamic loading feature is all very spangly, but how does it interact with the various C++ features that affect the overall behavior of linkers?

The first observation is that mangled names are a bit tricky. When `dlsym` is called, it takes a string containing the name of the symbol to be found. This has to be the linker-visible version of the name; in other words, the mangled version of the name.

Because the particular name mangling schemes can vary from platform to platform and from compiler to compiler, this means that it's pretty much impossible to dynamically locate a C++ symbol in a portable way. Even if you're happy to stick to one particular compiler and delve around in its internals, there are more problems in store—for anything other than vanilla C-like functions, you have to worry about pulling vtables and such like.

All in all, it's usually best to just stick to having a single, well known `extern "C"` entrypoint that can be `dlsym`ed; this entrypoint can be a factory method that returns pointers to full instances of a C++ class, allowing all of the C++ goodness to be accessed.

The compiler can sort out constructors for global objects in a `dlopen`ed library because there are a couple of special symbols that can be defined in the library and which the linker (whether load-time or run-time) will call when the library is dynamically loaded or unloaded—so the necessary constructor and destructor calls can be put there. In Unix these are functions called `_init` and `_fini`, or for more recent systems using the GNU toolchain, these are any functions marked with `__attribute__((constructor))` or `__attribute__((destructor))`. In Windows, the relevant function is `DllMain` with a reason parameter or `DLL_PROCESS_ATTACH` or `DLL_PROCESS_DETACH`.

Finally, dynamic loading works fine with the "fold duplicates" approach to template instantiation, but is much trickier with the "compile templates at link time" approach—in this case, "link time" is after the program is running (and possibly on a different machine than holds the source code). Check the compiler & linker documentation for ways round this.

More Details

The contents of this page have deliberately skipped a lot of details about how linkers work, because I've found that the level of description here covers 95% of the everyday problems that programmers encounter with the link steps for their programs.

If you want to go further, some additional references are:

- John Levine, *Linkers and Loaders*: contains lots and lots of information about the details of linkers and loaders work, including all the things I've skipped here. There also appears to be an online version of it (or an early draft of it) here
- Excellent link on the Mach-O format for binaries on Mac OS X [Added 27-Mar-06]
- Peter Van Der Linden, *Expert C Programming*: excellent book which includes more information about how C code transforms into a running program than any other C text I've encountered
- Scott Meyers, *More Effective C++*: Item 34 covers the pitfalls of combining C and C++ in the same program (whether linker-related or not).
- Bjarne Stroustrup, *The Design and Evolution of C++*: section 11.3 discusses linkage in C++ and how it came about
- Margaret A. Ellis & Bjarne Stroustrup, *The Annotated C++ Reference Manual*: section 7.2c describes one particular name mangling scheme
- ELF format reference [PDF]
- Two interesting articles on creating tiny Linux executables and a minimal Hello World in particular.
- "How To Write Shared Libraries" [PDF] by Ulrich Drepper has more details on ELF and relocation.

Many thanks to Mike Capp and Ed Wilson for useful suggestions about this page.

Back to Home Page

Contact me