NetHack 4 homepage | NetHack 4 blog | ATOM feed

Building C Projects

Tags: aimake internals c | Wed Nov 5 11:10:26 UTC 2014 | Written by Alex Smith

C is a compiled language, and as such, C programs need to be converted to executables in order to use them. Typically, programmers do this with some sort of build system, rather than running the necessary commands manually, because the compilation process for a large project tends to be quite complex.

NetHack's build system is one of the parts of the code that is more frequently replaced in variants. The problem is that the build system in NetHack 3.4.3 requires manual intervention from the end user in order to work correctly, which nowadays (in 2014) is almost unheard of. However, the normal methods of modifing the build system (e.g. using `autoconf` as in UnNetHack or AceHack, or `cmake` as in NitroHack) basically just add extra layers of abstraction; and in a world as poorly standardised as that of building C, what typically happens is that each of these layers is trying to work around incompatibilities and nonstandard behaviour in the layers below it, meaning that you have a complex stack built mostly out of workarounds.

Another issue is that there are many more steps in building a C program than most people realise. Some of these are handled by the build system, but others are typically left to the programmer or end user to do manually. This wastes everyone's time, really; the vast majority of the build process can be automated, and thus should be, because the time of a computer is so much less valuable than the time of a human (and computers are faster at it, anyway).

For NetHack 4, I aimed to flatten these levels of abstraction as much as possible; instead of writing yet another wrapper around yet another set of poorly standardised existing tools, I'm using `aimake`, a build system I wrote originally for NetHack 4 (but is not specific to it). It's still somewhat unfinished – there are known bugs, missing features, and interface issues – but it works quite effectively for building NetHack 4, and hopefully will become more widely used some day, when it's ready for release.

There are two main differences between `aimake` and other existing build tools. One is that `aimake` aims to avoid relying on lower-level build tools; most build systems work by producing output that serves as input to *other* build systems, and this can go down multiple layers before reaching anything that actually builds the code (e.g. with GNU Autotools, `Makefile.am` is the source for `Makefile.in` which is the source for `Makefile`, which contains instructions for actually building the code). The other is that `aimake` automates many more of these steps than most build systems do, handling steps which are often left to humans to deal with.

One problem with this method, though, is that it can lead to gaps in understanding; most people who work on build systems are merely trying to adapt to existing tools, rather than actually handling the build themselves. This can make `aimake` quite hard to understand, as it works at a relatively unfamiliar level of abstraction. Therefore, I'm writing this blog post to explain what actually goes on when you build a C program. A disclaimer: this post focuses on how things typically work on modern consumer operating systems (especially Linux and Windows), rather than trying to cover all possible obscure systems. C doesn't *have* to work like this. It just normally does in practice.

Normally, I'd start with the first step of the build and move forwards from there. However, that order of presenting things is actually quite confusing when it comes to building C; many of the steps are only needed to prepare for future steps, and it's hard to explain something without first explaining why it's needed. Thus, I'll start in the middle and work backwards and forwards from there. However, I'll first list the steps in order, before explaining them out of order (this order is not set in stone, but is a sensible and logical one to use):

1. Configuration
2. Standard directory detection

3. Source file dependency calculation
4. Header file location
5. Header precompilation
6. Preprocessing
7. Compilation and assembly
8. Object file dependency calculation
9. Linking
10. Installation
11. Resource linking
12. Package generation
13. Dynamic linking

Most build systems handle steps 3, 5 to 7, 9 and 10 in this list (leaving the others to be done manually, before or after the build). `aimake` currently handles 1 to 11 and has made a start on 12. Step 13 is almost always handled by the operating system.

And now, the main body of this post, a description of the build itself.

6: Preprocessing

As an example throughout this blog post, I'm going to use the following Hello World program in C.

```
#include <stdio.h>

int main(void)
{
    fputs("Hello, world!\n", stdout);
    return 0;
}
```

Right now, I want to focus on the `#include <stdio.h>` line. A common misconception among people new to C is that it has something to do with libraries, typically that it causes the definition of `fputs` to be "linked in" to the program somehow. Actually, its effect is quite different: it serves to tell the compiler what *types* `fputs` and `stdout` have. The above program is, on Linux, 100% equivalent to this one:

```
struct foo;
typedef struct foo FILE;

extern int fputs(const char *, FILE *);
extern FILE *stdout;

int main(void)
{
    fputs("Hello, world!\n", stdout);
    return 0;
}
```

We no longer have any mention of `stdio.h` anywhere, and yet the program still works. So clearly, it's nothing to do with linking.

What's actually going on here is that `stdio.h` is just a list of declarations of types, functions, and variables, together with some macro definitions. The C standard places no restrictions on how a compiler chooses to implement these definitions, but in practice, headers like `stdio.h` are nearly always implemented in a slightly extended dialect of C (that uses compiler-specific extensions to do things that could not be expressed in C directly, such as adding buffer overflow check code based on the lengths of arrays in the source program).

In this case, we're defining `FILE` as an incomplete type (the compiler doesn't need to know the definition of `FILE` to run the program, just that it's some sort of `struct`, because all sane compilers treat pointers to all `struct` identically); `fputs` as a function from a `const char *` and `FILE *` to an `int`; and `stdout` as a variable of type `FILE`. The `extern` means that this is just a declaration, and this file does not necessarily contain a definition of the function and variable in question. (We'll see where the definition actually is later on, when the linker gets involved.)

On Linux, `stdout` is actually a variable. On Windows using `mingw`, however, an accessor function is needed to find the value of the standard filehandles `stdin`, `stdout` and `stderr`, so the preprocessor expands the program to something like this (I've omitted some details, and thousands of irrelevant lines):

```
struct foo;
typedef struct foo FILE;

extern int fputs(const char *, FILE *);
extern FILE *__iob_func();

int main(void)
{
    fputs("Hello, world!\n", &__iob_func()[1]);
    return 0;
}
```

We can now see what the purpose of standard header files like `stdio.h` is: it's to paper over system-specific differences in the implementation details of the standard library. What the preprocessor is doing, then, is evaluating "preprocessor directives" like `#include` (to use definitions in a header file) or `#define` (to do symbol substitution on a source file, such as replacing `stdout` with `&__iob_func()[1]`). The result is a C source file that you could just have written directly, but is a little more system-specific than would be appropriate for the source files of your project.

In addition to standard header files like `stdio.h`, projects can also have their own header files (e.g. NetHack's `hack.h`). From the point of view of the preprocessor, these work exactly the same way as standard header files, and they contain similar content (although, obviously, tend to be less system-specific). These are distinguished by using double quotes rather than angle brackets, i.e. you would write `#include "hack.h"` but `#include <stdio.h>`.

The preprocessor is also traditionally responsible for removing comments from the source. It still does by default, for backwards compatibility, but modern compilers have no issue with comments, and the build process will work just fine even if you tell the preprocessor to leave the comments in.

Nowadays, the preprocessor is nearly always built into the compiler, because they work so closely together. However, the preprocessor can also be run individually. The standard name for a command that does preprocessing is `cpp` (standing for, I believe, "**C** **P**re**p**rocessor"); this is normally implemented as a wrapper for the C compiler, providing the `-E` command-line option to tell it to do preprocessing only (and in fact, `aimake` currently uses the `-E` method when it needs to access the preprocessor individually in order to prevent mismatches between which preprocessor and compiler it uses). The standard filename for preprocessed output is `.i`.

5: Header precompilation

In the early of C programming, the preprocessor worked on a very low level, pretty much just textually manipulating the source. Although preprocessors can still do that – in case someone decides to use them for something other than C, which is not unheard of – this rapidly runs into scaling problems in practice. The problem is that the standard headers have to be designed to handle all possible programs that might want to include them; our Hello World program just used `fputs` and `stdout`, but `stdio.h` has many other features, any or all of which a program might use. The result is that our preprocessed Hello World program is 1192 lines long;

most of them are blank, and most of the rest are unused, but the compiler has no way to know this until *after* preprocessing.

A traditional build process would preprocess each source file individually, meaning that each header file would have to be read by the preprocessor once for each source file. This has two problems. One of these is that it takes a long time (especially when using C++, whose build process is pretty similar to that of C). Modern computers are fast enough that the time taken is of the order of tens of seconds, rather than days, but even having to wait an extra minute in your build can significantly slow down your development process (because it reduces the rate at which you can use your compiler to catch errors, and means you can't get useful feedback from an IDE). The other problem is that if there is, say, a mistake in a declaration in a header file, this will cause an error compiling every single file that includes that header, whether it cares about the declaration or not.

Given that modern compilers work so closely with their preprocessor counterparts, there is a solution to these problems: the preprocessor and compiler can cooperate to compile as much of a header file as possible ahead of time, without needing to know which source files will eventually use it. Although not every part of a header file can be precompiled (e.g. the effect of the textual substitution that `#define` performs is impossible to precalculate because it might end up being stringified), much of a header file is declarations, which can be. This means that the header file only needs to be parsed once (rather than once per file that uses it), and that errors in it may be caught before it's used. (It also helps in the common case where the source files are changed without changing any headers; the same precompiled header can be used as last time.)

Although relatively old (I remember using them back in the 1990s), precompiled headers are nonetheless a newer innovation than most other parts of the C toolchain, and thus are relatively nonstandardised. `gcc` and `clang` use the extension `.h.gch` extension for precompiled headers; Visual Studio seems to use `.ipch`.

`aimake` currently always precompiles headers. I'm not 100% convinced this actually saves time – although it speeds up the compiler in most cases, it slows down `aimake` because it has to keep track of them – but the improved error messages are worth it.

4: Header file location

In order to be able to include header files, you have to be able to find them on the filesystem. The algorithm used by the vast majority of preprocessors is very simple: the preprocessor has a hardcoded list of directories, which the user can add to, and the preprocessor will scan them in sequence until it finds the file it's looking for. In the case of `include "file.h"`, it will also scan the directory containing the source file.

Although sufficient for small projects, in large projects, this approach is highly inadequate. Unlike in a small project, large projects will typically be formed out of several, mostly unrelated, parts (e.g. NetHack 4's source ships with libuncursed, which was originally written for NetHack 4 but is conceptually separate, and with libjansson, which was written entirely independently of NetHack and chosen as a dependency by the NitroHack developer). And with disparate sources comes naming clashes. So far, NetHack 4 has managed to avoid naming clashes among header files we ship (although there are currently six names which are used for multiple source files: `dump.c`, `log.c`, `messages.c`, `options.c`, `topten.c`, `windows.c`), but this increasingly becomes unlikely as projects get larger.

There is another problem, which has a simple solution, but is worth mentioning because it catches some people out: not all C compilers are configured correctly for the system they run on, and may miss a few system headers as a result. I've most commonly seen this happen with the directory `/usr/local/include` on Linux systems, which is intended for system-wide header files that were installed manually, rather than by the operating system's package manager. The fix is just to add the directory as an include directory for every compile using command-line options.

Anyway, as a result of the potential for name clashes, the set of header files to include for each individual source file needs to be determined for that file individually. The normal solution in build systems is to do this manually;

the user will specify a list of directories to scan for header files, typically a project-wide list with options to override it for particular source files. Although these specifications tend to be quite simple to specify with some sort of loop, and are not hard to write, the fact that they need to be written at all increases the chance of human error (and this sort of error has definitely caused problems in practice). Their existence also encourages the tendency to use multiple nested build systems with different configurations that is frequently seen in large projects, and which is nearly always a bad idea. (There is a relatively famous paper about this called Recursive Make Considered Harmful [PDF].)

In `aimake`, I use the approach of scanning the entire source tree for header files with appropriate names, then working out which header is intended via comparing paths (we take the closest matching header file in terms of directory structure). Different parts of a project have to be placed in different directories anyway when using a hierarchical build, and doing so is sensible anyway, so this is not a particularly odious requirement.

It's also necessary to scan the system include directories for header files, because they aren't always at a consistent location on every system. For instance, on my Ubuntu laptop, the Postgresql header that NetHack 4's server daemon needs is stored at `/usr/include/postgresql/libpq-fe.h`. On some other people's systems, it's just `/usr/include/libpq-fe.h`. The solution to this problem I've most commonly seen used in practice is to require the user to specify the location as a configuration variable, then use conditional compilation to select between `#include <postgresql/libpq-fe.h>` and `#include <libpq-fe.h>` respectively. Getting the build system to just search for the header file in question on the file system is simpler, and requires less human intervention, and thus is preferable. `aimake` does this, in addition to scanning the source tree.

There is one other subtlety here: because the preprocessor works at the level of directories rather than files, it's impossible to exclude a file from the header file search if another file in the same directory is included. This might seem minor, as there's no reason to arrange header files like that during the actual build. What it does do, however, is make it much more difficult to reliably discover dependency loops; a build system wants to avoid using files that aren't meant to have been built yet (and are left over from a previous compile), but the directory-level granularity of a preprocessor's header file search makes this impossible to avoid without using a separate directory for *each* generated header file (in case some are meant to have been generated at that point in the build, but not others). As a workaround for this problem, and for numerous other reasons, `aimake` now copies each header file in the source tree to a directory of its own in the build tree before building (with a `#line` directive so that errors are still reported against the original source file); I haven't had to resort to this for system headers, because they can't be confused with user headers due to the different syntax and system toolchain maintainers work to avoid clashes, but it would be a possibility if necessary.

It should be noted that the original reason I wrote `aimake` was that I'd accidentally introduced a dependency loop into a beta of my "attempt to merge AceHack with NitroHack" in late March 2012, a couple of days before it got the name NetHack 4, and couldn't figure out what was going wrong with the build (back then, I was using the `cmake` build system inherited from NitroHack, which didn't realise that anything was amiss and was hopeless at giving appropriate feedback for this kind of mistake). Because I was under time pressure (wanting to get the program ready for April 1), I quickly knocked a script together as a replacement build system to get the build working, and after a couple of rewrites and numerous smaller changes, that script eventually became `aimake`.

3: Source file dependency calculation

When working on a large project, you want to avoid recompiling it every time there are changes made to any file; rather, you want to compile just the bits that actually changed. This is probably the motivating factor behind the earliest build process automation tools; this sort of optimization is hard to express in a shellscript or batch file, leading to the creation of tools like `make`, which compare timestamps to determine what needs to be done in any given build.

I'd argue that nowadays, trying to generate input for `make` is probably the #1 biggest mistake you can make in a build system (even though it's very common practice; the problems being that dependency resolution is a relatively small part of a build, and working around differences in different `make` implementations (some of which are very primitive) is quite complex, and thus it's normally simpler to just resolve the dependencies yourself rather than trying to get `make` to do it. `make` is also quite limited in what it can do; for example, it has no obvious support for rebuilding a file when the build rules for that file in the `Makefile` change (as opposed to rebuilding nothing, or the entire project). Most build systems work as wrappers around `make` (or the equivalent for IDEs like Visual Studio); `aimake` just skips this level of abstraction and handles the dependencies itself. (I'm not the first person to come to this conclusion; for instance, the standard toolchain for Perl used to use `ExtUtils::MakeMaker`, which generated `Makefiles`, but is now moving in the direction of `Module::Build`, which does the dependencies itself.)

Anyway, regardless of whether you use `make`, `aimake`, or some completely different tool to process dependency calculation, you need to know what those dependencies actually are. This was originally left to the person writing the code to specify, and often still is in simple projects. However, as a task that can reasonably be automated (and that is somewhat time-consuming to do by hand for large projects), automating it makes a lot of sense.

Something many people don't realise about dependency calculation is that correct dependency information is actually needed for two different purposes:

1. Ensuring that if a file's dependencies change, it will be rebuilt.
2. Ensuring that a file won't be built until after its dependencies.

The most common method of dependency automation nowadays is to get the preprocessor to report on every header file it includes, during the build (and preprocessors have pretty advanced support for this; most can generate Makefile fragments directly). However, this leads to a chicken-and-egg problem (you can't calculate dependencies until you compile, but can't compile without knowing the dependencies). The standard solution to this (e.g. as used by GNU autotools) is to start the first build with no dependencies. This satisfies goal 1 (because on the first build, everything has to be rebuilt anyway, thus it doesn't matter that you don't know its dependencies), but fails goal 2, which tends to be dealt with using ugly workarounds. (Just look at this mess.)

An alternative tradeoff (and one that `aimake` uses, although other choices are reasonable depending on what your goals are) is to run the source files through the preprocessor in a separate stage, without compiling them, to get hold of the dependencies. Thus, in `aimake`, this is step 3 of the build (whereas with many other build systems, it would be step 6).

Although this seems simpler than producing dependencies as a side-effect of a compile would be, modern preprocessors actually make it much harder than you would think. The first problem is that they include header files transitively; this is what you want for a build (because you're actually building), but not what you want for dependency calculations (it's much faster, in the sense of "linear rather than quadratic", to look at what headers are needed without actually compiling them). The next problem is that the headers in question might not exist yet (e.g. they're generated files), and (in the order `aimake` does things) won't have been located yet. At least `gcc` and `clang` can be told to report on but ignore nonexistent files, which would seem to solve this problem.

My first attempt at implementing dependency automation was to ignore the first problem (as it only affected performance) and use the obvious solution to the second, but it lead to unexpected compile failures in NetHack, due to a reasonably subtle problem. What happened was that one of the files in NetHack is called `magic.h`; although it's part of the source tree, it hadn't been located yet (and in general, because a good build system would handle arbitrary files being generated during the compile without prior warning, there's no point in locating header files early because that won't help with generated files). This is fine if there's no file `magic.h` anywhere, because then `gcc` and `clang` will treat it as empty and just report on the dependency, which is exactly what we want. However, some operating systems have a system header `magic.h` (which I believe is related to detecting file formats), leading the preprocessor to report a dependency on the system `magic.h` (and all its transitive dependencies) rather than the one in the source tree!

Clearly, the behaviour I actually needed was to treat all header files as empty, rather than actually including them. Sadly, though, typically preprocessors don't have an option to do this. I actually seriously considered writing my own preprocessor to deal with the problem, but it would have been a mess and probably too large (`aimake` is implemented as a single file that you can ship with your project, much like `configure`, so I want to keep the size down if possible; it's already a little over ten thousand lines long). I eventually found a simpler solution, even if it is a bit sad: `aimake` reads the header files itself, looking for `#include` statements, then creates an empty file named after each header it finds, and tells the preprocessor to look there for include files.

Note that one limitation of this method of dependency handling is that a file can't determine which header files to include dynamically based on constants included in a different header file. There are plausible useful reasons to do this, and it's permitted by the C standard. However, looking at what actually happened in practice in NetHack 4, it turned out that almost every location where it happened was a mistake that caused header file inclusion order to be relevant when it shouldn't have been (and the other cases were reasonably fixable). So again, this seems like a reasonable tradeoff to make.

2: Standard directory detection

In order to be able to do any sort of processing of the system headers and libraries, you first have to know where they are. With header files, you need to know the location of the system headers to be able to scan the subdirectories of that location for header files like `libpq-fe.h`, which I discussed earlier. Libraries intended for public use are rarely stored in subdirectories; however, aimake needs to be able to look inside them in order to determine *which* library holds a particular symbol, so it needs to find the set of standard libraries, too.

It's actually a surprisingly hard problem to determine where the system headers and libraries are stored with no hardcoded knowledge of the system you're on, and this may be part of the reason why header file location and object file dependency calculation are normally left to humans by most build systems, rather than being automated. The headers and libraries could be just about anywhere on a system (e.g. on my Windows system, I installed the compiler to a subdirectory of my Documents directory, which could have had an entirely arbitrary name); assuming that all we know the location of is the executables of the C toolchain, can we find the rest?

I tried several techniques, and eventually settled upon two that seem reasonably portable (by which I mean that they work on Linux with `gcc`/`clang` and GNU `ld`/`gold`, on Windows with mingw, and apparently on Mac OS X too, although that's mostly been tested by other people because I don't have a Mac handy):

- For header files, the preprocessor must know where the header files are in order to be able to include them; and although there's no standard option for asking where they are, there is a standard option (`-M`) for reporting dependencies. Thus, all we need is to produce a file with one dependency in each of the possible standard header file directories.

  Currently, `aimake` uses five header files as dependencies. There's a full explanation in its source, but here's a quick summary of the possible directories that header files can be sorted into on systems I've seen, and which header file `aimake` uses as a test:

  - Header files that ship with the compiler (`<iso646.h>`, because there is literally no reason not to ship it with a compiler, especially as many C libraries don't ship it);
  - Header files that were patched by the compiler (`<limits.h>`, the only file that `gcc` patches *unconditionally*);
  - Architecture-dependent header files (`<sys/types.h>`; none of the standard C headers are stored here on most systems which have it, so I used the most architecture-dependent POSIX header);
  - Header files that ship with the C library (`<setjmp.h>` is the only header I've found that's in this category on every system I tested, probably because it's too weird for the compiler to mess with it);

  - Nonstandard (i.e. non-C/POSIX) header files that shipped with libraries (`<zlib.h>`, because zlib is one of the world's most widely-used libraries).

- For libraries, again there's no standard way to find out where the linker is looking, even though it must know; I attempted to use `gcc --print-search-dirs` even though it's `gcc`-specific, but it appears to only find library directories that `gcc` needs for its own internal use, rather than all the library directories on the system. The most portable method I've found is to ask the linker to link in each of the libraries we care about to a small test file (one at a time so that the failure to find a library won't affect the search for the others), and get it to produce a debug trace as it does so.

  There are several problems with this method (even though I use it anyway). The first is that although the option to ask linkers for a trace is standard (`-t`), the format of the output is not, and can vary wildly even on a single linker depending on the reason that the library is being linked in. Here's a current regex `aimake` might use for parsing the output (with the part `aimake` uses to avoid matching the source file removed, and with file extensions set appropriately for Linux; the file extensions need to change on other systems):

  ```
  /^(?:(?:-l|lib)[^ ]+ )?\(?(.*?\/.*?(?:\.a|\.o|\.so)(?:\.[0-9]+)?)(?:\)|\([^\/]*\)|\z)/
  ```

  The next problem is that, looking at the files matched by this, sometimes some of them will be text files! This problem is due to the fact that library developers sometimes have to do weird things for backwards compatibility. Here's what `libncurses.so` looks like on my Ubuntu system, in full:

  ```
  INPUT(libncurses.so.5 -ltinfo)
  ```

  I don't know for certain, but my guess as to what happened here is that `libncurses` got split into two libraries sometime in the past (`libncurses` and `libtinfo`), and thus `libncurses.so` became a wrapper linker script that pulls in both libraries. Obviously, this script is useless for things like determining which symbols exist in which libraries. However, the linker trace will also contain both of the libraries it pulls in, so all that's necessary with such wrappers is that we ignore them. `aimake` scans the first kilobyte or so of the library looking for non-ASCII characters; if it fails to find any, it assumes that the "library" is actually just a linker script.

  Another issue is that, if we don't use at least one symbol from a library, the linker may optimize it out; and in some circumstances (e.g. if it's a static library), it then won't show on the linker trace output. `aimake` currently only searches for libraries that the user mentions might be required in the configuration file (for performance reasons; searching every library on the entire system for symbols takes about an hour on my laptop), so I solved this by asking the user to mention the name of a symbol that they expect to be in that library, and `aimake` then forces the linker to link that symbol using the standard `-u` option. An added benefit of this approach is that it ensures that we find the library we expected, rather than a different library with the same name.

  A bad side-effect of this is with respect to the standard libraries of the system, the ones that are linked in by default. (In addition to needing to analyse these in order to do dependency calculations correctly, as any dependencies on them won't need to be explicitly satisfied, we also need to not pass them to the linker, because Mac OS X's linker complains and refuses to build if you try to explicitly link a standard library.) We can't know how those are divided up between files, so specifying a symbol is of no use. Thus, I ended up having to settle on the rather drastic method of asking the linker to link in *every* symbol in the standard libraries (`--whole-archive`, plus `--allow-multiple-definition` to reduce the amount of error spam you get as a result). This normally doesn't produce a usable output file (nor could it really be expected to), but `aimake` doesn't care about this, just about the trace output. Incidentally, it causes `mingw`'s linker to crash, but only after it's already printed the trace output `aimake` needs. (This is why building NetHack 4 on Windows shows an error dialogue box about `ld.exe` crashing.)

All this annoys me, as something this apparently basic really shouldn't be this difficult. I guess the problem is that nobody really expects someone to try to determine inherently system-specific information (where the headers and libraries are installed) in a system-agnostic way.

1: Configuration

Traditionally, the first thing done in any build is to fix values for the parts of the build that vary from build to build. This includes everything that needs to be specified by a human before the build starts; thus, configuration is quite lengthy and time-consuming with NetHack 3.4.3, as you have to answer questions like "what terminal codes library does your system use?". `aimake` cuts down considerably on this sort of question, because it can determine so much by itself, but there are always going to be questions that only a human can answer, such as "are you interested in building the tiles ports", or "do you want me to install this to your home directory or system-wide?". (At least `aimake` mostly manages to condense all these choices down to a couple of command-line options. It does have one major problem in terms of user-friendliness right now, though, which is that there's no simple way to specify unusual options you want to pass to the compiler, or similar special build rules; you can do it but the syntax is horrendous. This is something I need to work on before `aimake` is really ready for production use.)

However, there are also lots of questions whose answers can be determined experimentally, and which a computer can answer as easily (in fact, more easily) than a human. This is normally performed by a program named `configure` (typically, but not always, generated by a GNU `autoconf`), with a separate `configure` program written for each project and shipped with the distribution tarball. `configure`'s job is to paper over nonportabilities between systems by determining what needs to be done on each one.

Just as happened with curses, though (as I wrote in my blog post on portable terminal control codes), `configure` is increasingly solving the wrong problem. I have written systems based on GNU Autotools that aimed to do everthing by the book as much as possible, but it was mostly as a joke / thought exercise; modern programs don't need to worry about, say, `stdlib.h` not existing (and don't really have an obvious workaround for if it doesn't). At least `autoconf`'s documentation mentions that that particular check is obsolete nowadays, and recommends leaving it out.

There are several checks that are more useful, but so far, I've only actually needed two checks of this nature in NetHack 4 (and `aimake` has support for doing them), a very small portion of the build system as a whole (and not something that warrants being one third of the "standard" build sequence, which for autotools, is `configure`, `make`, `make install`). One is about how to set the compiler to C11 mode (`--std=c11`, `-std=c11`, or nothing and let the compiler fall back to non-standards-compliance mode, which contains all the C11 and C99 features we use in the case of `gcc` and `clang`). The other is how to indicate that a function never returns: is it `_Noreturn` (the version recently standardised in C11), `__declspec(noreturn)` (used by Microsoft compilers), or `__attribute__((noreturn))` (used by `gcc` and `clang` before the standard syntax was invented)? I originally tried to detect this using compiler predefined macros, but this caused problems on some specific old versions of `clang` (which are still used nowadays on some Mac OS X systems). Neither of these checks exists in my current version of `autoconf`, so it wouldn't have helped much here.

It should be noted that this is the most user-visible step (because it requires the most manual intervention), and also the one that varies the most between build systems. Thus, it needs the best interface, and it's a particular pity that `aimake` falls down here.

7: Compilation and assembly

We've now seen pretty much everything that happens at the C level, from the source files up to the preprocessor. As a recap from earlier, the preprocessor's output is typically a slightly extended C, which contains declarations for all the relevant parts of the standard library (and typically for irrelevant parts, too, but those get optimized out). The next part of the build process is the compilation and assembly, which takes the preprocessor's output and transforms it into an "object file".

Just as the preprocessor's output is slightly extended C, the object file (which is the linker's input) is slightly extended machine code. The difference from the machine code that a processor actually runs is that it's full of instructions to the linker, as well as instructions to the processor; typical linker instructions are along the lines of "this string is read-only data, place it with the other strings for me and tell the MMU to mark it read-only", or "I know I said to call the all-zeroes address as a subroutine, but can you replace that with the address of `strcmp` when you figure out where it is?". In short, an object file contains as much of the machine code as can be calculated by looking at one file in isolation; the rest is made out of linker instructions to patch up the file later on in the build.

There are two basic approaches that can be used for this C to machine code transformation. One is to do it directly; the other is to go via assembly code (which is almost as low-level as machine code, but which is much more human-readable, and which can express various linker instructions that machine code can't. When assembly code exists as an intermediate step (or sometimes even when it doesn't – some compilers can produce it on demand), compilers typically support the `-S` option to let you take a look at it. Our Hello World program expands to just 30 lines of assembly. It's also possible to "disassemble" the object file, to determine what its machine code would look like as assembly, to make it more human readable. Here's the Hello World program as C (as a reminder):

```
#include <stdio.h>

int main(void)
{
    fputs("Hello, world!\n", stdout);
    return 0;
}
```

And here's how it looks as assembly produced by the compiler, and how the object file looks after disassembly (rearranged so that common lines are next to each other; my disassembler also printed the machine code as hexadecimal, but I've removed that to save space, even though it leads to the newline and NUL at the end of "Hello, world!" displaying ambiguously as periods):

```
    .file   "t.c"
    .section    .rodata             Contents of section .rodata:
.LC0:
    .string "Hello, world!\n"       Hello, world!..
    .text                           Disassembly of section .text:
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp                    push   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp              mov    %rsp,%rbp
    .cfi_def_cfa_register 6
    movq    stdout(%rip), %rax      mov    0x0(%rip),%rax
    movq    %rax, %rcx              mov    %rax,%rcx
    movl    $14, %edx               mov    $0xe,%edx
    movl    $1, %esi                mov    $0x1,%esi
    movl    $.LC0, %edi             mov    $0x0,%edi
    call    fwrite                  callq  22 <main+0x22>
    movl    $0, %eax                mov    $0x0,%eax
    popq    %rbp                    pop    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
.section    .note.GNU-stack,"",@progbits
```

On the left is the assembly output for the Hello World program; we can see that it contains many notes for the linker (the lines starting with `.`), and various constructs that don't exist in machine code (e.g. it can say `call fwrite`, and mention `stdout`, even without knowing where `fwrite` and `stdout` are in memory). Comparing this to the machine code produced by disassembling the object file, we can see that many of the lines are basically the same, but some have bits replaced by zeroes; `movq stdout(%rip), %rax` cannot sensibly be interpreted as `mov 0x0(%rip),%rax`, for instance. Likewise, the call to `fwrite` has been replaced by a call to the current instruction pointer location (which also translates as zeroes in machine code).

It's worth noting that, even though this program was compiled without optimization, the compiler has done a small optimization anyway; the original Hello World program called `fputs`, but the compiler has compiled it into a call to `fwrite`. Both these functions need to use a buffer to store their output, according to the C standard; `fputs` has to do a series of `strncpy` or equivalent to fill the buffer (`strncpy` instead of `strcpy` because it has to allow for the potential that the string might be very long and so overflow the buffer, causing it to be output in sections), whereas `fwrite` knows the length before it even starts, and (for short strings like this) can `memcpy` it into the buffer directly. `memcpy` is faster than `strncpy` because it can copy more than one character at a time (`strncpy` has to take care not to read past the end of the string, and doesn't know where that will be in advance), so `fwrite` should be faster than `fputs`. (We can see the "14" in the assembly code, that isn't in the original program, containing the length of the string `"Hello, world!\n"` that appears in our program, not counting the terminating NUL because that isn't output.)

The object file places its output into various *sections*, which tells the linker (and eventually the OS kernel) about the purpose of the data stored there. Our `main` function goes into a section that is, rather confusingly, named `.text`; this contains the actual code that makes up our program. (This is on Linux; on other operating systems; the name of the section varies, but is nearly always some variation on "text" for historical reasons, even though executable code is about as binary on the binary/text distinction as you can get.) The string `"Hello, world!\n"` itself is in a separate section, `.rodata`, which stores read-only data.

The `.cfi_` lines tell the assembler to generate stack unwind information, which is mostly irrelevant here (it's used by exception handling in C++ to figure out which `catch` blocks might be relevant for the current code location, and by debuggers to produce stack backtraces); they also compile into a section (`.eh_frame`) in the object file.

In addition to the sections, our object file also contains *headers*, which contain instructions to the linker; the information about names like `main` and `stdout` hasn't just disappeared, but as it can't be expressed in machine code, it needs to be expressed some other way. On Linux, we can see these headers using `objdump -x`. This is more than a screenful of text, but here are some interesting excerpts:

```
SYMBOL TABLE:
[...]
0000000000000000 g     F .text  0000000000000029 main
0000000000000000         *UND*  0000000000000000 stdout
0000000000000000         *UND*  0000000000000000 fwrite

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000007 R_X86_64_PC32     stdout-0x0000000000000004
0000000000000019 R_X86_64_32       .rodata
000000000000001e R_X86_64_PC32     fwrite-0x0000000000000004
```

The "symbol table" tells the linker about any identifiers in the source code that need to be given a consistent meaning between files. In this case, `main` is 0x29 bytes from the start of the `.text` section, and the compiler doesn't know where `stdout` and `fwrite` are. Meanwhile, the "relocations" tell the linker which parts of the machine code will need patching up; it's asking the linker to substitute in appropriate values for just before `stdout`, just before `fwrite`, and for the start of this file's part of the `.rodata` section (which is where the `"Hello, world!\n"` string is stored). The compiler cannot figure out where any of these will be; it can't know where the `.rodata` section is relative to the `.text` section because it doesn't know how large those sections will be, and `stdout` and `fwrite` are in the standard library, not in the Hello World program itself. The hexadecimal numbers in the left column tell the linker where in the `.text` section the new values will need to be substituted (and the central column explains what format the linker should use for the substitutions).

The traditional name for a compiler is `cc`, and for an assembler is `as`. However, the compiler also traditionally runs the linker once it's done; although appropriate for very small programs, this is *not* what you normally want for a large project (where you're going to want to recompile just the source files that changed, but relink all the executables that depend on any file that changed). Thus, it's typical to tell the compiler to do *just* the compile (and assemble, if necessary) using the `-c` command line option, which is standard. Object files have the extension `.o` with most toolchains, although some Windows toolchains use `.obj`.

8: Object file dependency calculation

Once we have our object files, the next step is to work out how to fit them together to form executables. There are two sources of object files; one is the C source files we just built, and the other is the system libraries. (System libraries can ship object files directly; more commonly, though, they're bundled together into libraries of object files, with extensions like `.lib` or `.a`, which are simply archives containing multiple object files, typically with some extra metadata to allow the linker to figure out which of the contained object files are relevant more quickly. On Linux or UNIX-based systems like Mac OS X you can use the command `ar x` to unpack a library and look at the object files inside it directly.)

In order to produce an executable, we need a list of object files and libraries that together have no unmet dependencies; any undefined symbols in any of the object files (such as `stdout` in our Hello World object file) need to be defined by one of the other object files. We also need an "entry point" to the program, so that it has somewhere to start running. (The way this is typically implemented is for the entry point to be somewhere in a standard library, normally in an object file whose sole purpose is to provide an entry point; and that object file calls `main`, which it leaves undefined, performing initialization before the call and cleanup afterwards.)

Thus, one thing we need to do as part of the build is to produce such a list. This is a job that's normally left to the programmer, and in my opinion, making the programmer do it is a terrible idea. As usual with jobs that are left to humans, it's error-prone; unlike some of the other things that are normally left to humans, it's *also* somewhat time-consuming, as whenever you add a new file to your project, you'll need to add it as a source for one or more of your executables (or libraries, if you build libraries as part of your build).

This is thus an excellent candidate for automation, and the concept here is pretty simple: we have one executable for each `main` function, and we find the files required to produce it by recursively looking for object files or libraries that define symbols that are undefined in the object files that are already part of the build. In the case of our Hello World program, `aimake` would start by checking the symbol table for our main program, finding it needed definitions of `stdout` and `fwrite`, then finding those in the standard library, and it'd be done.

One potential problem here is if symbols with the same name are defined in multiple files, but `aimake` uses heuristics to figure out which file to link that work excellently in practice for correct programs. However, there's another problem: when the input programs contain subtle errors (such as linking against implementation details of a library defined elsewhere in the project, rather than using the public API of that library), `aimake` can normally find a solution that works anyway (for this example, linking against the relevant object files in that library directly), even though we wouldn't want it to. At the moment, I check to ensure this hasn't happened by inspecting the symbol tables manually once every few months, but this is unsatsifactory. Ideally, there should be some way to tell `aimake` (or some way to have it automatically deduce) that certain files should be kept separate; a crude way to do this at the moment is to define symbols with the same name in each to force a link error, but this isn't really satisfactory.

In order to actually discover which symbols are defined (or referenced but undefined) in the various object files, we need to look at their symbol table headers. We could use a program like `objdump` for this, but it's rather system-specific; there's a more generally usable program, `nm`, that deals with symbol tables specifically. Again, we have to deal with variations in the output format, and come across a particularly weird problem: POSIX defines a standard output format for `nm`, but the way you get at that format varies from `nm` implementation to `nm` implementation, making it not particularly useful as a standard. Meanwhile, the default output format of `nm` is consistent enough that just matching it directly seems to cause no problems.

9: Linking

Once we know which object files we need to combine to make an executable, the next step is to use the linker to actually make our executable.

The first job of the linker is to work out how to lay everything out in memory; it will generally combine sections with the same name into contiguous blocks. On modern systems, the sections keep their identity all the way into the executable, and so combining them means that the executable's headers will be simpler and shorter. There are multiple reasons that the executable needs to know about the sections. One is to correctly configure permissions on memory while the executable is running; the kernel will tell the computer's memory management unit (MMU) which parts of memory are supposed to be readable, writable, and/or executable, and violating these rules will cause a crash. For our the sections used by our Hello World program, `.rodata` is readable but not writable or executable (which is useful for diagnosing errors in which you try to write to a string literal, or to deallocate one); `.text` is executable and readable but not writable (for security reasons, toolchains try to avoid memory that's both writable and executable as far as possible). Another reason is that some of the sections can be optimized out of the output entirely; the `.bss` section is for read-write data that's initialized entirely with zeroes, which is a common enough special case that it makes sense to not have to pad the executable with that many bytes of zeroes. The kernel knows about this rule, but needs the `.bss` section identified so it can allocate memory for it at runtime.

The linker has a lot of control over the memory addresses used by the program; object files typically have no opinion on where something should be placed in memory, but it's in the linker's power to fix all the addresses within the resulting file, and this is what it does in typical usage. (There's no need to worry about clashes between executables; the MMU will place each executable in its own separate address space, so each executable has perfect control over how it wants to lay out memory.)

Once the linker's decided all the addresses that the program should use, it creates an output file that's basically a memory image of each section, and substitutes in addresses according to the relocations requested by the compiler. This list of sections, plus appropriate headers, is the executable file (i.e. `.exe` on Windows) that's actually produced on disk. In order to run such a file, the operating system basically just has to memory-map all the sections in the executable at the virtual addresses that the linker indicates, then starts the program running at the start address indicated. (Modern operating systems can thus optimize memory storage for executables the same way they would for memory-mapped files.)

It's worth noting that some operating systems have rather simpler executable formats than this. A DOS `.com` file is a memory image of a single section (starting at a virtual address of 0x0100), with an entry point at the start of the section, that's readable, writable and executable. The advantage of this format is that it doesn't need a header at all, as all the information that it would specify is fixed; the disadvantage is that because the 8086's memory management was so primitive, it means that the entire program is limited to 65280 bytes of program, static data, and stack.

Although the main job of the linker is to fix memory addresses and patch up the relocations accordingly, it also has a few other jobs. One is handling code that define a variable in multiple source files. There are three ways to specify variables that the linker might potentially have to process, in C:

1. Declaration that is never a definition, e.g. `extern int foo;`
2. Tentatively defining a variable as zero-initialized, e.g. `int foo;`
3. Definitely a definition due to an initializer, e.g. `int foo = 0;`

The first and third cases are well-defined and have the same meaning everywhere; the first uses `extern` to explicitly state it isn't a definition, and the third initializes the variable (and thus must be a definition). The second case is more ambiguous; the standard states that it's always a definition, but not all C programmers seem to have got the message, and sometimes it's incorrectly used as just a declaration instead. Worse, sometimes all your object files just say `int foo;`, and none of them initialize it; now (in pre-standard C) one of them is a definition, the others are declarations.

Many linkers thus have special logic to handle this case, converting one all but one plain `int foo;`-style definition into a declaration if it's required for the program to link correctly (by allocating all the definitions at the same memory address). Some don't, perhaps for technical issues (e.g. building shared libraries on Mac OS X). Because well-written programs shouldn't be doing this sort of thing anyway, and because it would confuse `aimake`'s heuristics, `aimake` tells the compiler to tell the linker that all definitions are definitely definitions (rather than possibly declarations), and thus can help catch this sort of error. (NetHack 4 was unintentionally doing this for a long time; two different variables `windowprocs` were getting merged together, when they shouldn't have been. It didn't affect the operation of the program, as it happens, so I'd never have caught it without `aimake` telling me something was wrong. It didn't manage to articulate the problem very clearly, though, and I had an incorrect fix in place for months, until I finally figured out what was wrong when working on the Mac OS X port of NetHack 4.)

The linker can also catch certain problems with an executable that the compiler can't; for instance, it can notice that a function isn't defined anywhere (leading to an undefined symbol error). However, the linker can't check as much as you'd like; if you declare a variable as an `int` in one file and a `float` in another, then it's very unlikely that the linker will be able to discover the type mismatch (it might know about the expected *size* of the two resulting symbols, on some operating systems, but both types are usually 4 bytes long; in order to know about the *type*, the only source of information the linker could rely on is the debug information, which linkers typically don't parse and which the compiler doesn't always generate).

10: Installation

Although you might think that the build is done once the executables are produced, there's still several steps left in a full build process. One problem at this point in the build is, although we have a working executable, it would be very hard to ship. One possible build layout is for the executables and object files and source files to all be mixed together in the source tree (something I personally hate because it makes it impossible to have multiple builds from the same source, and makes reversing a build hard, but which is nonetheless very common), which is too disorganized to ship easily. The alternative, having files generated during the build in a separate directory, means that the executables are separate from any data they might need (which is still sitting in the source tree). Either way, some files are going to need to be copied and/or reorganized before they're really usable.

The simplest possible form of installation is simply taking the files we just built, together with relevant data files from the source, and copying them to the location the user asked us to install them in (back during configuration, at the start of the build). There's nothing conceptually too difficult about this; however, it's nearly always a separate step from the actual build due to the potential need for elevated permissions (it's common to want to install into system directories that aren't writable by ordinary users, like `/usr/bin` on Linux or a subdirectory of `CSIDL_PROGRAM_FILES` (which is typically `C:\Program Files` on the filesystem) on Windows).

There are also some subtle details that build systems take care of during this step. For instance, the build system needs to create directories before installing in them. Another example is that it's common to "strip" executables while installing them; this discards all information in the executable (symbol tables, debug information, etc.) that isn't absolutely necessary for it to run, and so reduces disk usage at the expence of debuggability. (A more recent innovation is to split out the debug symbols into a separate file, typically stored on a network and downloaded when debugging information is needed.)

Another job of the install process is to set the permissions on the files it installs. This is surprisingly easy to get wrong, especially when they need to be different from the defaults. For example, on Linux and UNIX-based systems, NetHack in general and NetHack 4 in particular set permissions so that players can't write the high score table, or write bones files, without going through the `nethack` executable or using administrator privileges. Some distributions (which modified this step of the install process, despite warnings in the documentation that their new configuration was a bad idea) managed to screw this up to the extent of introducing a security bug that could allow anyone with local access to take over the whole system.

It's also worth noting the "DESTDIR convention", which will become relevant later. The idea is that you compile as if you were installing into one location (telling your program to look for its data files in `/usr/share`, for instance), then actually place the compiled files in a different location (typically somewhere in your home directory). It's easy to add this feature to an installer, and forgetting will make some of the later steps of the build process almost impossible to complete. (The way you specify this using GNU `automake`, and with most hand-written makefiles, is to set a `make` variable `DESTDIR`; `aimake` uses a command-line option `--destdir`.) In addition to its use in generating packages, this is also handy when installing into a chroot; I use this feature when updating the server on nethack4.org.

Still, this is one of the simpler steps. The main problem from the point of view of a build system designer is that because it needs enhanced privileges, the UI will inevitably be more complex than for the other steps. `aimake` supports three different models for handling the install step:

1. If you don't actually need permissions for your install, you can just use `-i` (i.e. "install as well as building") directly, and things work fine.

2. You can tell `aimake` how to elevate permissions with the `-S` option, e.g. `-S sudo`, and it'll use that method (`sudo` on Linux is very similar to UAC on Windows; it can only be used on an administrative account, and prompts for your password, elevating one process if i's entered correctly).

3. You can run `aimake` without `-i` to do the build; then elevate permissions, and run with `-i --install-only`. (This is how `-S` is implemented internally.)

The second method is the most convenient, and minimizes the risk of permissions-related errors (a build directory full of files you don't own is always frustrating to deal with). The third method is the one that most other build systems use, so it's included to make it easier to write a wrapper that makes `aimake` look like, say, GNU autotools.

11: Resource linking

We're now past the point where most build systems consider their work to be done. This is a pity, as resource linking is something that typically has to be done at the same time as the installation; the normal workaround (that's been repeated so many times across so many projects that many people don't realise it *is* a workaround) is to manually write build rules to perform the relevant steps. On Windows, the situation is less bad, because Visual Studio does a resource link just after linking (and thus it happens immediately before installation).

The term "resource link" comes from Windows, but the general principle is the same across operating systems: a plain executable by itself is not particularly usable, because all it has is a filename, and (if you're lucky) documentation. Documentation might be useful to a human, but what can a computer do with a name like `nethack4.exe` but place it in a list of thousands of other executables that nobody will ever scroll through? (This is not just a hypothetical; it's Firefox's actual UI on Linux for selecting a program to open a file, when it doesn't know of an appropriate option already, because it just opens up an "Open File" dialog box that you even have to navigate to `/usr/bin` by hand. This really sucks when all you wanted was to open a text file in a text editor.)

Clearly, what is needed is some sort of metadata: a longer description than the filename, an icon, perhaps a version number, things like that. On Windows, this is compiled into the executable. On Linux, it's typically done by means of `.desktop` files in a known directory (`/usr/share/applications`); many desktop environments maintain an index of these files so that they can grab the metadata for a given executable on demand. (I'm not sure how this works on Mac OS X, although would be interested for someone to tell me.)

It's impossible for a build system to know what all the metadata should be without being told. However, `aimake` can certainly translate it to an appropriate format for the OS by itself, and the amount of information it needs to make a decent shot at it is actually very small. (The only pieces of information that it can't at least make a reasonable guess at or a plausible placeholder for are the version number and icon; and if not specified, it'll just tell the OS to use its default placeholder icon.)

It would be reasonable to do this step before the install, rather than after; that's where Visual Studio does it, and I originally tried to put it there. However, doing it just after the install (while `aimake` still has elevated permissions, so that it can write to system directories like `/usr/share/applications`) turned out to be much simpler in terms of `aimake` code (which in turn made NetHack 4's `aimake.rules` simpler as it didn't have to try to deal with the pre-resource-link and post-resource-link versions of executables separately). Besides, on Windows, you want to make a Start menu shortcut as well, and because shell links on Windows have a curious mix of overengineering and underengineering (some of which is to compensate for bad design decisions made elsewhere), it's impossible to create a robustly behaving shortcut unless its target exists at the time. (`aimake` currently does not make these shortcuts during the install process for this reason, also because most of the existing wrappers use the wrong APIs and when I wrote a new wrapper, it was 153 lines of C++ before even writing the code to communicate with `aimake` itself.)

As a fun fact, the two main NetHack 4 executables, `nethack4` and `nethack4-sdl`, differ only in filename and metadata (`nethack4-sdl` is a symlink on Linux, because the metadata is in external files). The filename difference determines the default interface plugin to use (an ending of `-sdl` loads the graphical interface by default); as well as explaining the interface difference, the metadata is used to request a terminal window for `nethack4`, while not opening one for `nethack4-sdl` because it doesn't need one and so it would look ugly.

12: Package generation

So far, we've been focusing on the build and install process as it looks to a developer. However, distributing a source tarball and telling users to build it is considered a very user-unfriendly method of distribution nowadays. On Linux, nontechnical users want a package that they can point their distribution's package manager to. On Windows, most users will want an installer that they can double-click on. (Additionally, nothing about the previous steps has any particular support for an *uninstaller*, something which any software package should have; the sort of end-user installation framework discussed in this section typically comes with an uninstaller as well.)

This step is going to be inherently somewhat system-specific; Windows Installer has many differences from `dpkg` (the Debian equivalent), for instance. However, some things are the same. The most notable is that the input into these install tools is a folder hierarchy containing the files that would be installed; the DESTDIR convention is very useful here. (The package generation tools also need to be told about the permissions on the files, but ideally, administrator permissions wouldn't be needed for a DESTDIR install; on Linux, the workaround for this problem is a program called `fakeroot` which intercepts attempts to set permissions and remembers which permissions the files "should" have, and on Windows, `aimake` will make a list of all the files it's installing and the permissions they need and pass that to the package generation tool separately.)

Currently, `aimake`'s support for generating an installer on Windows is almost complete; it can get as far as generating an input file for WiX, which can then take the build process the rest of the way. The only thing it doesn't do is actually run WiX; that's currently left to the user, even though it should be easier to automate than do manually. (One problem is that Windows Installer has a few design issues that need working around, and some of the recommended workarounds are ridiculous; eventually, I stopped trying to use them, and instead came up with my own workarounds which both practically and theoretically seem to work better, but now the build process spams warnings and errors because I'm not using the recommended workarounds. (Interestingly, the errors and many of the warnings are false positives caused by the same design issues that lead to the workarounds being necessary in the first place. It's a good thing that "errors" from Windows Installer's consistency checkers are really just warnings, and thus can be turned off.)

The situation on Linux is a little more complex. So far I've been focusing on Debian (who have the most widely used package format). Debian's policy is written assuming that either the package was developed for Debian in particular, or that it was developed without knowledge of Debian and was packaged separately by one of their developers. The case of "a package that exists on its own but knows how to deploy itself on Debian" makes it quite hard to work out an appropriate source package (a binary package is much easier). Currently, `aimake` itself has all the appropriate options to handle being run *by* `dpkg-buildpackage`, which requires a short wrapper script (31 lines in NetHack 4, probably about 20 for most packages) to tell `dpkg-buildpackage` how `aimake` wants to be run. Much of the Debian metadata should be the same for all `aimake`-using programs, though (just as happened with WiX), so I'm hopeful that eventually, `aimake` will have an option to just automate all the steps in generating a Debian package. This would lead to a weird situation in which `aimake` is running `dpkg-buildpackage` to run itself, but this method maximises the chance that the build is reproducible.

13: Dynamic linking

As something that happens at runtime, this is a little irrelevant to the design of a build system, but I'm mentioning it for completeness. So far, we've been implicitly assuming that the build is a so-called *static build*, in which all the code that runs at runtime is part of the executable. In practice, though, this isn't normally the case; an executable will depend on *shared libraries* (`.so` files on Linux, `.dll` on Windows, `.dylib` on Mac OS X), which contain code that's shared between multiple executables.

Conceptually, dynamic linking is pretty similar to regular, static linking; the dynamic linker picks an address for the shared library and maps it into the address space of the executable, which then makes calls into it. This means that the executable needs to contain some sort of relocations, just the same way that the object file did. Here are the five relocations that end up in our Hello World executable on my 64-bit Linux system (as printed by `readelf -r`):

```
Relocation section '.rela.dyn' at offset 0x3a8 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600ff8  000200000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000601040  000400000005 R_X86_64_COPY     0000000000601040 stdout + 0

Relocation section '.rela.plt' at offset 0x3d8 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main + 0
000000601020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 __gmon_start__ + 0
000000601028  000300000007 R_X86_64_JUMP_SLO 0000000000000000 fwrite + 0
```

We can see our old friends `stdout` and `fwrite` here, still waiting to be linked against `libc`. However, we can see that the offsets for the relocations are much higher than existed in the object file; that's because the linker has already determined addresses for everything in the executable itself, and so knows where the relocations need to be at runtime. Dynamic linking is harder than static linking because the OS really wants the shared library to be bitwise identical between all the processes that run it (so that it can maintain just one copy of it in physical memory); on 64-bit Linux, the solution to this is to place the relocations in various tables that the dynamic linker updates, rather than in the code directly, and making calls indirectly via the table. (This won't work for `stdout`, which is a variable rather than a function; the linker resolved the problem by putting it in a separate table.) This also means that the shared library needs to be written in such a way that it can be moved around in memory and still run correctly, but on x86_64, the processor makes this pretty easy by allowing memory references to be relative to the program counter. (Meanwhile, on 32-bit x86, it's a complete nightmare; the compiler can handle it, but it has to jump through such hoops that you can instead tell the compiler to not bother, in which case the dynamic linker has to make a copy of the shared library in memory and patch up all the relocations the same way that the regular linker does. Linux's dynamic linker is actually willing to do this; on x86_64, though, it refuses, explaining that it really should be the compiler's job to make the library position-independent on a system where it's so easy.)

I won't go into all the details of dynamic linking here, partly because they aren't relevant to understanding build systems and partly because I don't know what they are. However, there is one issue that needs to be addressed: what does it mean to link against a shared library, given that its code doesn't actually end up in the executable? The answer lies in the concept of an *import library*, which is a file that looks like a static library to the linker, but actually just puts relocations into the output executable, rather than code. On Linux, each shared library can also act an import library for itself, which makes shared library deployment pretty easy, but writing build systems confusing (especially when you consider the case of two shared libraries that need symbols from each other at runtime; this is not technically a circular dependency, but you need to somehow come up with a working import library by other means, such as by first linking the libraries without the dependency on each other). On Windows, import libraries tend to be separate files that are generated by a program `dlltool` from a description of which symbols go into the shared library; `aimake` gives it the object files from which the shared libraries are built as a handy method of describing the shared library without actually needing a copy of it (and thus breaking up the circular dependency that way).

There's also a point that's important for programmers, too. On Windows, imports from a shared library will need special code generated for them, so they have to be marked in the source file. Likewise, exports need to be marked as such inside the shared library on Windows. On Linux, exports don't *need* to be marked, but they should be; this makes it possible for a build system to hide all the non-exported symbols from the API of the shared library by means of command-line options, thus reducing namespace pollution.

Handling appropriate marking is normally done by the preprocessor. You typically have a header file declaring the API of your shared library, which looks something like this:

```
#ifdef IN_SHARED_LIBRARY_INTERNALS
# define IMPORT_EXPORT(x) AIMAKE_EXPORT(x)
#else
# define IMPORT_EXPORT(x) AIMAKE_IMPORT(x)
#endif

int IMPORT_EXPORT(function1) (void);
int IMPORT_EXPORT(function2) (int, int);
```

Then, users of the shared library just include the header; the shared library itself does `#define IN_SHARED_LIBRARY_INTERNALS` and then includes the header, and thus gets the "export" definitions rather than the "import" definitions. The `AIMAKE_EXPORT` and `AIMAKE_IMPORT` macros are defined by `aimake` to expand into whatever system-specific annotation is needed to import or export from a shared library. (The exact syntax is subject to change, because `gcc` doesn't like the way the current syntax interacts with functions that return pointers; I'm currently working around that with `typedef`, but really need a better solution.)

As far as I can tell, this sort of marking is necessary to create shared libraries, as the API of the shared library has to be specified *somehow*. However, `aimake` will spot these annotations and automatically generate shared libraries rather than executables when it sees them, so no extra work is needed beyond the bare minimum. (Some improvements to this are needed, though; it currently doesn't work on Mac OS X because I don't know enough about shared libraries on that system, and there really should be some way to override the whole mechanism and generate static libraries instead.)

Conclusions

Hopefully, this post should give C programmers something of a better understanding of the toolchain that goes into actually building their programs; perhaps it'll even inspire someone to go into toolchain development. I hope I've also made the point that a lot more of a build toolchain can and should be automated than typically is; there's a lot of programmer time being wasted right now dealing with things that should really be done by a computer. Most of the attempts I've seen to fix this are dealing with the wrong problem; people look at existing build systems, and think "we need a better tool for writing Makefiles" or "we need a better tool for generating configuration", when these are really relatively small parts of a build.

I hope `aimake` acts as a proof of concept that almost the entire build process can and should be automated, and perhaps one day grows into a tool that can be widely used for this purpose (even if today, it suffers from UI problems, the occasional bug, missing features, and incomplete platform support). At the very least, maybe the world's build system designers will be inspired to deal with every part of the build, not just the small corners they previously worked on, and I'll be able to use something standard rather than being forced to work on `aimake`.