Creating Executable Programs

Simple Introduction to Compiling and Linking

Here is a simple example to go into my latex notes regarding compiling and linking.

We use three simple C programs main.c, sum.c and sum.h. We will compile the two source files seperately so that we can see what the preprocessor, compiler, and linker each do. The main.c source file

```
#include "sum.h"
int main()
{
    int x = 10;
    int y = 12;
    int z = sum(x,y);
}
```

is run through the preprocessor using the `-E` flag (note that the preprocessor adds some extra output that helps with compiler warnings etc. These lines start with a `#` so I will just remove these lines for simplicity's sake.)

```
patricklnoble$ gcc -E main.c | grep -v '#'
int sum(int a, int b);
int main()
{
    int x = 10;
    int y = 12;
    int z = sum(x,y);
}
```

The preprocessor has literally copied the contents of `sum.h` which was included into `main.c`. This is the code that the compiler actually sees - it now knows everything it needs about the call to the `sum` function - its name `sum`, it takes two `int` arguments and its return type `int`. This is the *declaration* of the function `sum` - the name, the number and types of its arguments, and its return types.

To compile (and assemble) we use the `-c` flag, and we get a binary *object* file `main.o`. This contains the actually machine instruction in binary form that the cpu can execute. We will later look at these files in more detail, but for now lets just look at the names inside it. To do this we look at the *symbol table* of the program `main.o` using the program `nm`.

```
patricklnoble$ nm main.o
0000000000000000 T _main
                 U _sum
```

The `U` before `_sum` (on mac, symbol names like `sum` get a leading underscore for some reason) means that the function `sum` is *undefined*. A function *definition* tells the compiler what the function actually *does*, as opposed to just its definition (type information). The `T` means that `main` is defined in the `TEXT` section of the object file. Again we will look at these files in more detail later.

The final step on the way to a running executable file is to run the linker. The `gcc` linker is called `ld`, but it is almost never a good idea to try to link a program directly using `ld`. Instead, we again call `gcc` which knows that if the input files are object files (ie `.o` files) it should use the linker and not the compiler. Calling the `gcc` linker on our object file `main.o` gives

```
patricklnoble$ gcc main.o -o test
Undefined symbols for architecture x86_64:
"_sum", referenced from:
_main in main.o
ld: symbol(s) not found for architecture x86_64
```

A few things. Note from the bottom line that we called `gcc main.o etc`, but the error has come from the linker `ld`. `gcc` was clever enough to know that since we passed an object file it should use

`ld` to try and link it. The error gives us information we already knew from looking at the object file ourself with `nm`: the *name* `_sum` in the object file is undefined. Noone know what the function `sum` actually does, so we can't expect to be able to create a executable program using it!

To fix the situation we need a *definition* for the function `sum`. It should be consistent with the *declaration* we provided in `sum.h` (ie that it should take two integer arguments, and return an integer result). The definition is in `sum.c`

```
1 // sum.c
2 int sum(int a, int b)
3 {
4     return a + b;
5 }
```

We compile this into an object file too `gcc -c sum.c` which produces `sum.o`

```
1 patricklnoble$ nm sum.o
2 0000000000000000 T _sum
```

Inside `sum.o` we see that `sum` is defined (it no longer has the `U` prefix that undefined names have). Now it is the linker's job to take the definition of `sum` from `sum.o` and jam it into `main.o` so that `sum` is no longer undefined there.

```
1 patricklnoble$ gcc sum.o main.o -o test
```

The input files can be in any order, they just have to be object files (with a `.o` suffix). The `-o` option allows you to give the outputed executable a particular name - here I chose the name `test`. Let's look at the names in `test`:

```
1 patricklnoble$ nm test
2 0000000100000000 T __mh_execute_header
3 0000000100000f80 T _main
4 0000000100000f60 T _sum
5 U dyld_stub_binder
```

There are a couple of points to be made here. First and most importantly, note that `sum` (which on Mac is called `_sum`) is no longer undefined. The linker took the `main.o` object file, saw that `_sum` was undefined in it, and found the definition in `sum.o`. It was able to combine the two files correctly so everything is defined properly. Second, there are some extra bits here. What are `__mh_execute_header` and `dyld_stub_binder`? Why is it ok that `dyld_stub_binder` remains undefined - can we have undefined names in an executable program? The answer is that in addition to the steps outlined above there is a lot more that the compiler, assembler and linker (and loader) need to do to prepare a program to actually run. It is possible (common) to use external source code (ie a library) that is only *linked* with your program when your program is actually run, as opposed to when it is being compiled and linked as in this chapter. This type of library is known as a *dynamic* library. These will be undefined in the executable file (and so will show up as `U` in the symbol table) and will not be resolved until run time. In this case `dyld_stub_binder` helps with this run time process for certain (unknown to me) low-level system libraries on mac. The `__mh_execute_header` helps when the program is actually loaded and fed into the cpu to run.

Locating headers and object files

In the above examples `main.c`, `sum.c` and `sum.h` are all in the same directory. This makes things easier because we don't have to tell `gcc` where the required files are - it looks in the current directory by default. So what if your headers or object files are somewhere else? This is very common when using code from a library (or a library of your own).

In the present directory I create a new directory `dir` and move `sum.c` and `sum.h` here. Now the code from above will fail.

```
main.c:2:10: fatal error: 'sum.h' file not found
#include "sum.h"
^~~~~~~
1 error generated.
```

This failure comes from the preprocessor when trying to load the header `sum.h` - it cannot find the file. Adding the `-v` option shows where `gcc` looked for `sum.h` (in addition to the current directory)

```
patricklnoble$ gcc -c main.c -v
Apple LLVM version 9.0.0 (clang-900.0.39.2)
Target: x86_64-apple-darwin17.3.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/
    ↪ XcodeDefault.xctoolchain/usr/bin
#include "..." search starts here:
#include <...> search starts here:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.
    ↪ xctoolchain/usr/lib/clang/9.0.0/include
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.
    ↪ xctoolchain/usr/include
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/
    ↪ Developer/SDKs/MacOSX10.13.sdk/usr/include
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/
    ↪ Developer/SDKs/MacOSX10.13.sdk/System/Library/Frameworks (framework
    ↪ directory)
End of search list.
main.c:2:10: fatal error: 'sum.h' file not found
#include "sum.h"
^~~~~~~
1 error generated.
```

Note that I removed some of the non-necessary output. To add another directory to search use the `-I` option and say to also search in `dir`.

```
#include "..." search starts here:
#include <...> search starts here:
dir
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.
    ↪ xctoolchain/usr/lib/clang/9.0.0/include
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.
    ↪ xctoolchain/usr/include
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/
    ↪ Developer/SDKs/MacOSX10.13.sdk/usr/include
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/
    ↪ Developer/SDKs/MacOSX10.13.sdk/System/Library/Frameworks (framework
    ↪ directory)
End of search list.
```

So `dir` was added to the search path, and `sum.h` was found. Now we have the `main.o` object file. However the function `sum` is still undefined! While the preprocessor just needed the new directory to search, the linker needs two pieces of information - the name of the file to link to `main.o` and the directory for the linker to search in. The syntax is as follows.

```
patricklnoble$ gcc main.o -lsum.o -Ldir
```

The `-l` should proceed every file that needs to be linked, and `-L` should specify which directories the linker should search for the required `-l` files. You can see which directories the linker searched using the linker `ld` itself. Leaving out the `-L` notice in the output

```
patricklnoble$ ld -v main.o -lsum.o
@(#)PROGRAM:ld  PROJECT:ld64-305
configured to support archs: armv6 armv7 armv7s arm64 i386 x86_64 x86_64h
    ↪ armv6m armv7k armv7m armv7em (tvOS)
Library search paths:
/usr/lib
Framework search paths:
/Library/Frameworks/
/System/Library/Frameworks/
ld: library not found for -lsum.o
```

the linker shows its default search locations (and the different architectures this linker supports - tbd later). Now add the `-L` and run again,

```
Patricks-MacBook-Pro:CompilingAndLinking1 patricklnoble$ ld -v main.o -lsum
    ↪ .o -Ldir
@(#)PROGRAM:ld  PROJECT:ld64-305
configured to support archs: armv6 armv7 armv7s arm64 i386 x86_64 x86_64h
    ↪ armv6m armv7k armv7m armv7em (tvOS)
Library search paths:
dir
/usr/lib
Framework search paths:
/Library/Frameworks/
/System/Library/Frameworks/
```

and see the the new directory `dir` has been added to the library search path.

- Important: ** The **compiler** uses the `-I` arguments to find headers. Since it already knows the header file name, only the directories to search are needed. ** The **linker** uses both the `-l` and `-L` arguments. The linker needs to know both the name of the file containing object code and the directories to search for.

When we move on to makefiles and libraries we are just automating the process of compiling projects in small parts, exactly how we compiled `main.c` and `sum.c` seperately and then linked them together. The program `make` helps automate this process when building complicated projects.

Compiling C and C++ Programs

Linking C++ programs with source code in C (and other languages) can be a bit involved because C++ has more complicated naming rules than C because of function/method overloading (tbd). Lets do an example.

First, the extensions given to source code matters because `gcc` uses it to infer the language the code is written in. The list of file extensions is here Note that `gcc` will try to compile `.c`, `.` etc files in `C++` but it won't include the standard library properly and will usually fail with a wall or errors.

- Use `gcc` to compile `C` programs, and use `.c` and `.h` for source files and headers.
- Use `g++` to compile `C++` programs.

A trivial `C++` version of the previous `main.c` is given below in `main`.

```
// main.c
#include <iostream>
#include "sum.h"
int main()
{
    int x = 10;
    int y = 12;
    int z = sum(x,y);
    std::cout << x << " + " << y << " = " << z << std::endl;
}
```

Try to compile it with `gcc` and watch it crash! Compiling it as a `C++` program using `g++ -c main. -Idir` gives a much longer symbol table than our previous `C` programs. Most of this is standard library code. Look at what has happened to the `sum` function name:

```
patricklnoble$ nm main.o | grep sum
U __Unwind_Resume
U __Z3sumii
```

Whereas the relationship between the name of a function in a `C` source file and the name given by the compiler in the symbol table is obvious, in `C++` that is no longer true. This can make linking together compiled `C++` code (eg `main.o` above) with `C` code (ie `sum.o` we compiled earlier). One option would be to recompile `sum.c` using `g++` instead of `gcc`, so that we use `C++` naming instead of `C`. Moving into directory `dir` we compile with `C++` and looking at the symbol table

```
patricklnoble$ g++ -c sum.c
patricklnoble$ nm sum.o
0000000000000000 T __Z3sumii
```

we see that the name given to the `sum` function by the `C++` compiler is the same as what is expected by the compiler in `main.cc`. Now the whole process just works

```
patricklnoble$ g++ main.o -o main -Ldir -lsum.o
patricklnoble$ ./main
10 + 12 = 22
```

This is a horrible solution though - it means we need to have two different versions of `sum.o`, one for use in `C++` and one for `C`. It would be much better to be able to tell the `C++` compiler that the `sum` function defined in `sum.o` was compiled by a `C` compiler. To do this change the declaration of the `sum` function

```
patricklnoble$ cat dir/sum.h
extern "C" int sum(int x, int y);
```

where the `extern "C"` tells compilers including `sum.h` that the function `sum` has a `C` name rather than a `C++` name. Now compile `main.` again

```
patricklnoble$ g++ -c main. -Idir
```

with the new `sum.h` header, and look in the symbol table for the name of the `sum` function:

```
patricklnoble$ nm main.o | grep sum
U __Unwind_Resume
0000000100001ce0 T _sum
```

Now the `C++` compiler expects the `sum` function to be a `C` function, and now the linker will easily be able to find `_sum` in the object files.

```
patricklnoble$ g++ main.o -Ldir -lsum.o -o main
patricklnoble$ ./main
10 + 12 = 22
```

Introduction to Libraries and GNU make

The previous section has shown that properly compiling and linking a program can be a lot of work. It is also very error prone - if a change was made in one of the `.c` source code files, did we remember to compile it into a `.o` object file and link everything again? In a large project with many thousands (or more) source files this is quite a task. `C/C++` deal with this in two ways - **libraries** and **GNU make**.

- **Libraries** collect groups of object files together into a single package, so that instead of having to explicitely link every source file needed we can just link against the package containing the object code we need. There are two types of libraries - **static** and **dynamic** - we will discuss both in the next section.
- **GNU make** is a program that is included as part of the **GNU toolchain** that automates the whole compiling and linking process (including creating and using libraries). For very small projects `make` can be overkill, but as the size of projects grow it can be indespensible.

Libraries

A library is a collection of object files. Using libraries is conveniant because it saves us from having to link against every single object file needed by our code. There are also memory savings for dynamic libraries (to be discussed). We will start with static libraries because our use of `sum.o` in

the previous section was essentially identical to linking `main.c` with a static library containing one object file `sum.o`. We will then look at dynamic linking, and see why dynamic linking is the default library type for `gcc/g++`.

These examples are in a new directory `CompilingAndLinking2` which itself contains two directories `include` and `libs`. This is not an optimal/standard project organisation but it will illustrate the issues we need to dealt with. Our headers will go in `include` and source code and object files will go in `libs`. If we were 'selling' our software we would provide users with the header files and compiled libraries. We *do not* provide users with the `.c` source code - this remains our intellectual property!

Static Libraries

A static library is a collection of object files that get completely linked into our executable program at the time the program is itself created. This means that all the object code is baked into the executable program, and we can run the program at anytime without ever needing any of the static library code again. This is conveniant and portable, but it can make the program binary file large in size.

To make the project structure more interesting, we will define addition and subtraction functions taking two and three arguments respectively. The project directory structure will look as follows:

```
- CompilingAndLinking2/
   - main
   - include/
       - binary/
           - sum.h
           - subtract.h
       - ternary/
           - sum.h
           - subtract.h
   - libs/
       - binary_src/
           - sum.c
           - subtract.c
       - ternary_src/
           - sum.c
           - subtract.c
       - libbinary.a
       - libternary.a
```

We will provide the user of our code the ability to add and subtract two numbers (binary operations) and to add and subtract three numbers (ternary operations). For the user to incorporate our addition/subtraction library into their code, they need

- **Headers** for them to compile their code.
- **Libraires** which contain the actual implementation of our addition/subtraction code.

We don't want the users to know anything about *how* we do our addition/subtraction (that would give away our IP, and mean that the user no longer needs to buy our software!), which is why we would *not* ship the two directories `binary_src` and `ternary_src`.

First we compile our four source files. For example, the ternary sum looks like this

```
// libs/ternary_src/sum.c
#include "ternary/sum.h"
int sum(int x, int y, int z)
{
    return x + y + z;
}
```

Here we assume that the compiler will have the project directory `CompilingAndLinking2/include` added to the header search path `-I`, so that when looking for the header referred to above the

compiler will open the directory `CompilingAndLinking2/include`, see the directory `ternary`, and then inside that find `sum.h`. Since we will use the directory `CompilingAndLinking2` regularly, lets define

```
patricklnoble$ export CL2=/Users/patricklnoble/Documents/Projects/
    ↪ CppProjects/Viswanath/CompilingAndLinking2
patricklnoble$ echo $CL2
/Users/patricklnoble/Documents/Projects/CppProjects/Viswanath/
    ↪ CompilingAndLinking2
```

so that the compiler/linker instructions are a little shorter. Some definitions like this will be provided to library users so that they can put the library code wherever they like and our compiler instructions will still be correct.

```
patricklnoble$ g++ -c sum.c -o sum.o -I${CL2}/include
```

Do the same for `subtract.c`. This completes the object code for the first of our two libraries - the library for binary addition and subtraction. To package up the library we use the GNU archive program `ar`. `ar` is so named because the collection of object files (here `subtract.o` and `sum.o` are). Now create the archive with `ar`

```
patricklnoble$ ar -rv libbinary.a sum.o subtract.o
ar: creating archive libbinary.a
a - sum.o
a - subtract.o
```

The `-r` flag adds an object file to an archive, and if the archive doesn't exist it creates a new one (and `-v` is verbose). The static library naming convention is is always `libXXX.a` where `XXX` is the desired name of the library. The line above is creating functions for binary operations so the `XXX` is `binary`. The contents of the archive are interesting:

```
patricklnoble$ nm libbinary.a
libbinary.a(sum.o):
0000000000000000 T __Z3sumii

libbinary.a(subtract.o):
0000000000000000 T __Z8subtractii
```

Move into the the `ternary_src` directory and create a similar archive

```
patricklnoble$ ar -rv libternary.a subtract.o sum.o
ar: creating archive libternary.a
a - subtract.o
a - sum.o
```

Now we have the following directory structure and libraries

```
patricklnoble$ ll CompilingAndLinking2/libs/*/*.a
  1520 Mar 30 20:08 CompilingAndLinking2/libs/binary_src/libbinary.a
  1536 Mar 30 20:14 CompilingAndLinking2/libs/ternary_src/libternary.a
```

Now inside `$CL2/main` write a `main.` program that uses the libraries.

```
Patricks-MacBook-Pro:main patricklnoble$ cat main.c
#include <iostream>
#include <binary/sum.h>
#include <ternary/sum.h>

int main()
{
int x = 12;
int y = 9;
int z = 21;

int binarySum  = sum(x,y);
int ternarySum = sum(x,y,z);
std::cout << "sum(" << x << "," << y << ") = " << binarySum << std::endl;
std::cout << "sum(" << x << "," << y << "," << z << ") = " << ternarySum <<
    ↪  std::endl;
}
```

To compile and link it, first compile

```
patricklnoble$ g++ -c main.c -I${CL2}/include
```

Remember, the compiler only needs the headers. We now have to link `main.o` with our two libraries to create the executable program. Now link as follows

```
Patricks-MacBook-Pro:main patricklnoble$ g++ main.o -o main -L${CL2}/libs/
    ↪ binary_src -lbinary -L${CL2}/libs/ternary_src -lternary
```

and look in the symbol table

```
patricklnoble$ nm main | grep sum
U __Unwind_Resume
0000000100001cb0 T __Z3sumii
0000000100001cd0 T __Z3sumiii
```

Note that the two `sum` functions are resolved inside the executable `main`. This is how a static library works. The object code inside our two libraries has been linking *into* the executable `main`. The code in the libraries is never needed again. Running the program gives

```
Patricks-MacBook-Pro:main patricklnoble$ ./main
sum(12,9) = 21
sum(12,9,21) = 42
```

And we are done!

Dynamic Libraries

A dynamic library is a collection of object code in the same way as a shared library. It diff

Compile the source code for binary and ternary functions into shareable objects

```
patricklnoble$ g++ -shared -I${CL2}/include -fPIC sum.c subtract.o -o
    ↪ libternary.so
patricklnoble$ g++ -shared -I${CL2}/include -fPIC sum.c subtract.o -o
    ↪ libbinary.so
```

Look inside one of the new shared libraries

```
Patricks-MacBook-Pro:ternary_src patricklnoble$ nm libternary.so
0000000000000f70 T __Z3sumiii
0000000000000f90 T __Z8subtractiii
U dyld_stub_binder
```

Now `main.c` is compiled as before

```
patricklnoble$ g++ main.c -c -I${CL2}/include
```

Note the `sum` functions are still undefined

```
Patricks-MacBook-Pro:main patricklnoble$ nm main.o | grep sum
U __Unwind_Resume
U __Z3sumii
U __Z3sumiii
```

Link `main.o` against the dynamic/shared libraries

```
patricklnoble$ g++ main.o -o main -L${CL2}/libs/binary_src -L${CL2}/libs/
    ↪ ternary_src -lbinary -lternary
```

Note that the `sum` functions are still undefined in the executable

```
Patricks-MacBook-Pro:main patricklnoble$ nm main | grep sum
U __Unwind_Resume
U __Z3sumii
U __Z3sumii
```

Alas, when we try to run `main` we crash at run time

```
patricklnoble$ ./main
dyld: Library not loaded: libbinary.so
Referenced from: /Users/patricklnoble/Documents/Projects/CppProjects/
    ↪ Viswanath/CompilingAndLinking2/main/./main
Reason: image not found
Abort trap: 6
```

On Mac we use the program `otool` to see what dynamic libraries an executable expects

```
patricklnoble$ otool -L main
main:
libbinary.so (compatibility version 0.0.0, current version 0.0.0)
libternary.so (compatibility version 0.0.0, current version 0.0.0)
/usr/lib/libc++.1.dylib (compatibility version 1.0.0, current version
    ↪ 400.9.0)
/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
    ↪ 1252.0.0
```

On linux this is done by `ldd`. We see both the `binary` and `ternary` libraries are expected. The error happens because the dynamic linker is unable to find the `binary` library (same goes for `ternary`). To instruct the dynamic linker where the library is we need to define the variable `DYLD_LIBRARY_PATH`

```
patricklnoble$ export DYLD_LIBRARY_PATH=${CL2}/libs/binary_src:${CL2}/libs/
    ↪ ternary_src
```

After doing this `main` runs as before.

```
Patricks-MacBook-Pro:main patricklnoble$ ./main
sum(12,9) = 21
sum(12,9,21) = 4
```

make

`make` is a GNU program that automates the process of building and compiling libraries and programs. The strength of `make` is in capturing the dependencies that exists between a library full of headers and source files, and determining whether files need to be recompiled or not.

A `makefile` consists of a set of rules made up of targets, prerequisites and commands as follows `bash target: [prereq1 prereq2 ...] [command1 command2 command3 ...]` A rule is evaluated in the following way: * If a target is a file, `make` checks if the target file is more current than all of the prerequisits. If it is more current than all prerequisits, stop before running any commands. * If a prerequisit is older than a target `make` should search for a rule where the the prerequisit **is** the target. Then `make` should run all the commands in the rule.

Scope, Lifetime and Linkage

Linkage

A common build case is where symbols are shared between different object files. To see how to do this, we need a few definitions.

Translation Unit

A **translation unit** or **compilation unit** is the basic piece of code that the compiler works with. It consists of a single source file (ie a `.c`,`.cc` etc) file after it has been processed by the preprocessor. It will include all headers, substituted macros, flags/switches (ie all `#include`,`#ifndef`,`#define` etc). It

is the job of the compiler to turn a translation unit into a object file. After different translation units have been turned into multiple object files, it is the job of the **linker** to peice them together.

Scope

A symbol name always refers to a particular entity - be it a variable, function, class, namespace etc etc. In different parts of a program however, the same name can refer to different entities. The region of the program where a particular name refers to a particular entity is referred to as a *scope*. Scopes are usually delimited by braces `{...}` in control statements, functions, classes, namespaces and so on. All symbols declared outside of *any* particular scope are implicitely in the unnamed *global scope*. If one scope is inside another scope (ie is nested), then the symbol in the outside scope is available in the inside scope as well.

Definition vs Declaration

`C/C++` makes a clear distinction between **definitions** and **declarations**. A declaration informs the compiler that a name exists, and it has a certain type. No memory for the object referred to is actually allocated at this point. This means the compiler does not need to know a lot about the name - ie what it does/how it works. Below are some declarations.

```
1 int function1(double, int);
2 void function2(float*, float*);
3 class TestClass;
4 extern int;
```

The last - `extern int;` is interesting. Variables are often declared and defined at the same time. For instance

```
1 double function1(double x, double y) { return x + y; }
2 class TestClass { };
```

Essentially, if there are braces `{}` there is both declaration and definition happening. Variables confuse things a little things, because a statement which appears as a declaration for a class is *both* a definition and a declaration for a variable eg

```
1 class TestClass;  // declaration
2 int x;            // declaration AND definition
```

Confusing. It is possible to *only* declare a variable however - it requires the use of the `extern` keyword as in the below code

```
1 extern int x;
```

Functions can also be declared using `extern`, but it is redundant since functions are `extern` by default. The exception is, as we have seen previously, we want to refer to `C` code defined in a library somewhere. There we use `extern` (along with a directive to the compiler that it should not `C++` mangle the name of the function). But here, it is the name wrangling the necessitated the `extern` keyword. Classes are not declared `extern` - we will see how to deal with them.

A critical rule for compilers and linkers is the *one definition rule*. You can declare a symbol as many times as you like. However, it can only be *defined once*. The following code is perfectly legal

```
1 int func(int,int);    // decl 1
2 int func(int,int);    // decl 2
3 int func(int,int);    // decl 3
4 int func(int x ,int y) { return x + y; }  // def 1
```

however the following is *very* illegal

```
int func(int x, int y) { return x + y; }
int func(int x, int y) [ return x + y; }
```

where the compiler throws the following errors

```
patricklnoble$ g++14 test.cc -c
test.cc:4:5: error: redefinition of 'func'
int func(int x, int y) { return x + y; }
    ^
test.cc:3:5: note: previous definition is here
int func(int x, int y) { return x + y; }
    ^
1 error generated.
```

This is pretty obvious - don't define a symbol twice. However, this can lead to quite subtle errors when we try to link multiple object files together. Consider the following trivial example. The header `header.h` contains the following function *definition*

```
// header.h
int func(int x, int y) { return x + y; }
```

which is included into two source files which intend to use this function, the first is a ridiculous fused multiple add

```
// source1.cc
#include "header.h"
#include "source1.h"
int fmadd(int a, int x, int y)
{
    return func(a*x,y);
}
```

and the second a fused divide add

```
// source2.cc
#include "header.h"
#include "source2.h"
int fdadd(int a, int x, int y)
{
    return func(a/x,y);
}
```

Both compile with no problems

```
patricklnoble$ g++ -c source1.cc source2.cc
```

into `source1.o` and `source2.o`. Are functions are constructed properly

```
patricklnoble$ g++ -c source1.cc source2.cc
patricklnoble$ nm -C source1.o
0000000000000000 T func(int, int)
0000000000000020 T fmadd(int, int, int)
patricklnoble$ nm -C source2.o
0000000000000000 T func(int, int)
0000000000000020 T fdadd(int, int, int)
```

Now, lets make a program which actually uses these. We include both `source1.h` and `source2.h` to get the *declarations* of `fmadd` and `fdadd`, as is good practice. This also compiles

```
patricklnoble$ g++ -c test.cc
```

However, when we try to link together, we get the following errors:

```
patricklnoble$ g++ source1.o source2.o test.o
duplicate symbol __Z4funcii in:
    source1.o
    source2.o
ld: 1 duplicate symbol for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see
    ↪ invocation
```

The dreaded *duplicate symbol* error. How did this happen - we didnt define anything twice, did we? Yes we did! Both `source1.cc` and `source2.cc` included `header.h`, which contained the definition of `func`. The name `int func(int,int)` is now **defined** in two translation units - the one belonging to `source1.cc` and once belonging to `source2.cc`. Each compiled without problem, since they are compiled seperately. But the linker choked - which `func(int,int)` should it use? See in the `nm -C` output - `func(int,int)` is defined identically twice! This is the reason that we should generally put all declarations in headers, and definitions in sources. Very often we need the *definition* of names in multiple places, but we want to use only one *declaration*.

Sharing names between translation units

Sometimes we may want to share data between two translation units. Here is a contrived example. Here is a `main`, where we increment a variable `counter` which is in the global namespace. The function `logCounter` logs the current value of `counter`.

```
1 #include <iostream>
2 #include "logCounter.h"
3 int counter = 0;
4 int main()
5 {
6   for (int i=0; i<5; ++i)
7   {
8     logCounter();
9     ++counter;
10  }
11 }
```

The function `logCounter` function is declared and defined in `logCounter.[cc|h]`

```
1 patricklnoble$ cat logCounter.cc
2 #include "logCounter.h"
3 void logCounter(void)
4 {
5   std::cout << counter << std::endl;
6 }
```

However, the compiler cannot compile this - what is `counter`?

```
1 patricklnoble$ g++14 -c logCounter.cc
2 logCounter.cc:4:16: error: use of undeclared identifier 'counter';
```

The solution is to instruct the compiler that `counter` exists - that it will be *defined somewhere else*. They way to do this is to *declare* `counter` using `extern`.

```
1 // logCounter.cc
2 #include "logCounter.h"
3 void logCounter(void)
4 {
5   extern int counter;  // declare counter here.
6                        // linker will deal with finding counter later
7   std::cout << counter << std::endl;
8 }
```

Now everything compiles and link properly!

```
1 patricklnoble$ g++ -c logCounter.cc test.cc
2 patricklnoble$ g++ logCounter.o test.o -o test
3 patricklnoble$ ./test
4 0
5 1
6 2
7 3
8 4
```

The variable `counter` is said to have **external linkage**. That is its name (ie `counter` is known across multiple translation units. Importantly, `counter` in `logCounter.cc` and `counter` in `main.cc` are

referring to the same peice of memory - this is the crux of external linkage. If I print `counter`'s address in both `logCounter.cc` and `test.c` this is apparent. Remember that this is the work of the linker - the compilation of each translation unit happens completely independantly.

```
// logCounter.cc
#include "logCounter.h"
void logCounter(void)
{
  extern int counter;
  std::cout << counter << "," << &counter << std::endl;
}
// test.cc
#include <iostream>
#include "logCounter.h"
int counter = 0;
int main()
{
  for (int i=0; i<5; ++i)
  {
    std::cout << "&counter = " << &counter << std::endl;
    logCounter();
    ++counter;
  }
}
```

and recompiling/linking

```
patricklnoble$ g++ -c logCounter.cc test.cc
patricklnoble$ g++ logCounter.o test.o -o test
patricklnoble$ ./test
&counter = 0x101ffd0e8
0,0x101ffd0e8
&counter = 0x101ffd0e8
1,0x101ffd0e8
... etc
```

This process is quite dangerous - the writer of `test.cc` may not expect that the call to `logCounter()` will be able to modify their counter variable! As such, often global variables are hidden in a `namespace`. Note that the above procedure works inside a general namespace though. If `counter` is hidden in namespace `ns`, it can still be referred to as follows:

```
// test.cc
#include <iostream>
#include "logCounter.h"
namespace ns { int counter = 0; }  // hidden in namespace ns
int main()
{
...

// logCounter.cc
#include "logCounter.h"
namespace ns { extern int counter; }  // just refer to counter in ns
void logCounter(void)
{
  std::cout << ns::counter << "," << &ns::counter << std::endl;
}
```

The variable does need to be in global namespace for the linker to be able to resolve the name. Returning the code back to its original form, except define `counter` inside `main` - ie not in global namespace

```
// test..cc
#include <iostream>
#include "logCounter.h"
int main()
{
  int counter = 0;  // counter is NOT in global namespace
  for (int i=0; i<5; ++i)
  {
    logCounter();
    ++counter;
  }
}
```

both files are still perfectly legal to the compiler, but the linker will not be able to find the variable `counter` referred to in `logCounter.cc`. This is the dreaded `undefined symbol` linker error.

```
patricklnoble$ g++ -c logCounter.cc test.cc
patricklnoble$ g++ logCounter.o test.o -o test
Undefined symbols for architecture x86_64:
  "_counter", referenced from:
      logCounter() in logCounter.o
```

This all begs the question - what if I don't want a variable in my program - ie like `counter`, to be visible in any other translation unit? This gives me more certainty that some other peice of code I've never seen (eg `logCounter.cc`) is not doing something nefarious with my `counter`!. What is needed is **internal linkage**. That is, you want the name `counter` to only be known inside your translation unit, and nowhere else. This will stop the *linker* from being able to find your variable when it is linking together your code and other code written elsewhere. There are two ways to do this, using the *static keyword* or *anonymous namespaces*. The former, using `static` is now deprecated in favour of the later. That said, seeing `static` in code is very common so important to know. To use `static` in this context, define `counter`

```
#include <iostream>
#include "logCounter.h"
static int counter = 0;
int main()
{
  for (int i=0; i<5; ++i)
  {
    logCounter();
    ++counter;
  }
}
```

while leaving `logCounter.cc` unchanged, so that it uses the declaration `extern int counter;`. Both files compile with no problems, but now the linker will complain that `counter` is unresolved.

```
patricklnoble$ g++ -c logCounter.cc test.cc
patricklnoble$ g++ logCounter.o test.o -o test
Undefined symbols for architecture x86_64:
  "_counter", referenced from:
      logCounter() in logCounter.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see
   ↪ invocation
```

This is the effect of giving `counter` *internal linkage*. It stops the linker from being able to *see* `counter` from a different translation unit. The other option is to hide `counter` in a anonymous (unnamed) namespace like below:

```
patricklnoble$ cat test.cc
#include <iostream>
#include "logCounter.h"
namespace { int counter = 0; }
...
```

Again, compiling works but linking fails as above. In this case, it is very clear *why* the failure happened by looking at the symbol table:

```
patricklnoble$ nm test.o
                 U __Z10logCounterv
00000000000000b0 b __ZN12_GLOBAL__N_17counterE
0000000000000000 T _main
```

Different compilers will handle this different - but see that symbol for `counter` has `_GLOBAL__N_` appended to the front - this is due to the anonymous namespace, and is how the compiler stops the linker from finding the symbol. In `gcc` it is even more explicit, where the entry looks like

```
patricklnoble$ nm test.o
... entries
.... b (anonymous namespace)::counter
```

Lifetime

The period of the program's execution that a symbol refers to and owns a peice of memory is the lifetime of the name. Global variables are are created when the program starts running, and are not destroyed until the programs ends. Variables local to a particular scope (ie a function, or class etc) have different lifetime rules depending on their definition.

Most variables defined in some local scope have *automatic lifetime* - that is they are created when program execution hits that scope (ie this function is called etc) and are destroyed automatically after the scope ends. There are some exceptions to this idea - *static* lifetimes, and memory dynamically allocated. In the latter case, we know that the symbol referring to the dynamic memory will be destroyed, but the memory it pointed to is not deallocated. In this sense the variable has *automatic* lifetime, because the symbol is destroyed, but in another sense it does not, because the memory it points to still remains.

Local static objects are created *before* the first pass through the object's definition, and are not destroyed until after the program terminates. They are persistant. Note that their *definition* actually only occurs once - even if the definition appears like it is encountered many times. For example, this program:

```
#include <iostream>
void func(void)
{
  static int x = 0;  // this only occurs once!
  std::cout << "x = " << x << std::endl;
  x += 1;
}

int main()
{
  for (size_t i=0; i<5; ++i)
  {
      func();
  }
}
```

print the following:

```
Patricks-MacBook-Pro-24560:CompilingAndLinking3 patricklnoble$ ./test
x = 0
x = 1
x = 2
x = 3
x = 4
```

Classes can have **static** members and member functions. These members/member functions belong to the class itself, and not any particular instance of the class. Static methods can only refer to static members - since they do not belong to a particular class instance, they do not get a `this` pointer when called.

There are some subtleties to declaring and defining static class members. Since a static data member does not belong to a particular instance of the class, they are not defined when we create an instance of the class and are not intialised by the class's constructor. As such, they need to be *defined outside of the class scope*. To refer to static members or member functions, one can use the class namespace and access directly, or they can be referred to using an instance of the class. The following snipped demonstrates these issues.

```
#include <iostream>
class A
{
public:
  static int data;   // is a declaration only.  a definition here is
      ↪ illegal!
  static int f(int x) { std::cout << x << std::endl;}
};
```

```
8
9  int A::data = 10;     // definition must be outside of class
10 int main()
11 {
12     A a;
13     std::cout << A::data << std::endl;  // use class scope to refer to data
14     std::cout << a.data << std::endl    // use instance to refer to data.
15     std::cout << A::f(20) << std::endl; // use class scope to refer to f
16     std::cout << a.f(20) << std::endl;  // use instance to refer to f
17 }
```

Note that in the above, the static member `A::data` was defined *once* when the program executes, and was never defined again - regardless of how many instances of `A` were created during the running of the program. It is not destroyed until the program itself terminates. It does not matter if *any* instances of `A` ever existed in the first place.

Further, objects can themselves be declared **static**. As opposed to local/automatic variables, static objects are not destroyed until the end of the entire program. `GeeksForGeeks` has a great example. In the following, an instance of the class `A` is constructed inside the call to the function `func`.

```
1  class A
2  {
3  public:
4    A(void) { std::cout << " A constructor " << std::endl; }
5    ~A(void) { std::cout << " A destructor " << std::endl; }
6  };
7  void func(void)
8  {
9    std::cout << " func called. " << std::endl;
10   A a;
11   std::cout << " func exited. " << std::endl;
12 }
13
14 int main()
15 {
16   std::cout << " main() called " << std::endl;
17   func();
18   std::cout << " main() exited " << std::endl;
19 }
```

In the following output, we see that the object `a` of type `A` is created inside the function `func`. As a local variable inside the scope of the function, it has automatic storage/lifetime and is destroyed when the function terminates.

```
1 Patricks-MacBook-Pro-24560:CompilingAndLinking3 patricknoble$ ./test
2  main() called
3  func called.
4  A constructor
5  func exited.
6  A destructor
7  main() exited
```

Now, we change the storage class to static, and observe the difference. Now `func` is

```
1 void func(void)
2 {
3   std::cout << " func called. " << std::endl;
4   static A a;
5   std::cout << " func exited. " << std::endl;
6 }
```

and when the program is recompiled and run, we see that `a` is *not* destroyed at the end of `func`, but instead is destroyed after `main` terminates - ie the end of the program!

```
1 patricknoble$ ./test
2  main() called
3  func called.
4  A constructor
5  func exited.
6  main() exited
7  A destructor
```

Compiling and Linking MKL

BLAS and LAPACK

BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra Package) are names given to numerical linear algebra packages. In these notes we use two different implementations:

- **Eigen C++** : Eigen is very expressive high performance open source library for linear algebra.
- **Intel MKL**: the Intel MKL is a super high performance library optimised for Intel hardware. It is much less expressive and more difficult to work with the Eigen, but cannot be beat in terms of raw performance.

BLAS is usually described as having three parts * **Level 1**: vector operations * **Level 2**: matrix - vector operations * **Level 3**: matrix - matrix operations

These notes continue those in *CompilingAndLinking1* and look at how to install, compile and link programs using the Intel MKL.

MKL directories

In the documentation it refers to `<Composer XD directory>` and `<mkl directory>` directories. I think these are

```
/opt/intel
/opt/intel/mkl
```

on my system So we have the relevant installs. Now we need to set a number of environment variables which is done using scripts in blist{}

```
/opt/intel/mkl/bin
```

To use the `IA-64` architecture we source the following file

```
patricklnoble$ source /opt/intel/mkl/bin/mklvars.sh intel64
```

which appears to set a number of import environment variables like `MKLROOT` and `DYLD_LIBRARY_PATH`. Note that the docs suggest that directories

```
/opt/intel/mkl/bash/[intel32/intel64]
```

exist but I can't seem to see them. The download also clearly knows we are on a mac - note where `MKLROOT` points

```
Patricks-MacBook-Pro:examples patricklnoble$ echo $MKLROOT
/opt/intel/compilers_and_libraries_2018.1.126/mac/mkl
```

It looks like the headers required are all in `$MKLROOT/include`, and we can see them all being included by the header `$MKLROOT/include/mkl.h`

```
patricklnoble$ grep '#include' $MKLROOT/include/mkl.h | head -n 5
#include "mkl_version.h"
#include "mkl_types.h"
#include "mkl_blas.h"
#include "mkl_trans.h"
#include "mkl_cblas.h"
```

The libraries themselves are in `$MKLROOT/bin`.

Intel installed everything as roolt. This means you can't uncompress the examples. Go to `/` and change the owner of all the directories to `patricklnoble`

```
sudo chown -R opt patricklnoble
```

Decompress all the `C` examples in `$MKLROOT/examples`

```
tar -xzf examples_core_c.tgz
```

I went to the `solverc` examples in `$MKLROOT/examples/solverc` and use the `makefile` there to build the static 64-bit examples

```
make libintel64 compiler=gnu
```

Have a look at the ouput below - this might help us compile and link our own versions.

```
----- Compiling gnu_lp64_omp_intel64_lib ----- cg_mrhs_stop_crt_c
gcc -m64  -w -I"/opt/intel/compilers_and_libraries_2018.1.126/mac/mkl/
    ↪ include" \
./source/cg_mrhs_stop_crt_c.c \
"/opt/intel/compilers_and_libraries_2018.1.126/mac/mkl/lib/
    ↪ libmkl_intel_lp64.a" \
"/opt/intel/compilers_and_libraries_2018.1.126/mac/mkl/lib/
    ↪ libmkl_intel_thread.a" \
"/opt/intel/compilers_and_libraries_2018.1.126/mac/mkl/lib/libmkl_core.a" \
-Wl,-rpath,/opt/intel/compilers_and_libraries_2018.1.126/mac/mkl/lib -Wl,-
    ↪ rpath,/opt/intel/compilers_and_libraries_2018.1.126/mac/mkl/../
    ↪ compiler/lib -Wl,-rpath,/opt/intel/compilers_and_libraries_2018
    ↪ .1.126/mac/mkl/../tbb/lib \
-L"/opt/intel/compilers_and_libraries_2018.1.126/mac/mkl/../compiler/lib" -
    ↪ liomp5 -lpthread -lm -o _results/gnu_lp64_omp_intel64_lib/
    ↪ cg_mrhs_stop_crt_c.out
----- Execution gnu_lp64_omp_intel64_lib ----- cg_mrhs_stop_crt_c
_results/gnu_lp64_omp_intel64_lib/cg_mrhs_stop_crt_c.out > _results/
    ↪ gnu_lp64_omp_intel64_lib/cg_mrhs_stop_crt_c.res
```

This wrote all the executables and output files here

```
/opt/intel/mkl/examples/solverc/_results/gnu_lp64_omp_intel64_lib
```

Looking in the `*.res` files, it looks like we were able to compile and run these examples! For example, looking inside `dss_sym_c.res`

```
patricknoble$ cat /opt/intel/mkl/examples/solverc/_results/
    ↪ gnu_lp64_omp_intel64_lib/dss_sym_c.res
determinant power is 0
determinant base is 2.25
Determinant is 2.25
Solution array:  -326.333 983 163.417 398 61.
```

it isnt obvious what the example was but it certainly did something.

Lets try to use a routine ourselves. A really simple call is `cblas_dscalx` which implements vector-scalar multplication. An intel example using it is here `/opt/intel/mkl/examples/cblas/source/`
↪ `cblas_dscalx.c`. I have written a simple `main.cpp` to call it.

```
#include <iostream>
#include "mkl.h"
int main()
{
    std::cout << "Compile MKL" << std::endl;
    MKL_INT n = 10;           // size of vector
    MKL_INT incx = 1;         // increment
    double a = 3.70;          // scalar
    double* x = (double *)mkl_calloc(10, sizeof(double),64);

    //  write a test array
    for (int i=0; i<n; i++)
    {
        x[i] = i;
        std::cout << "x["<< i << "] = " << x[i] << std::endl;
    }
    // call mkl cblas_dscal
    cblas_dscal(n, a, x, incx);

    // print output
    for (int i=0; i<n; i++)
    {
        std::cout << "x[" << i << "] = " << x[i] << std::endl;
```

```
    }
}
```

I compile it like this

```
g++ main.cpp -I$MKLROOT/include ${MKLROOT}/lib/libmkl_intel_ilp64.a ${
    ↪ MKLROOT}/lib/libmkl_sequential.a ${MKLROOT}/lib/libmkl_core.a -
    ↪ lpthread -lm -ldl -o main
```

which produces the correct result!

```
patricklnoble$ ./main
Compile MKL
x[0] = 0
x[1] = 1
x[2] = 2
x[3] = 3
x[4] = 4
x[5] = 5
x[6] = 6
x[7] = 7
x[8] = 8
x[9] = 9
x[0] = 0
x[1] = 3.7
x[2] = 7.4
x[3] = 11.1
x[4] = 14.8
x[5] = 18.5
x[6] = 22.2
x[7] = 25.9
x[8] = 29.6
x[9] = 33.3
```

Installing Tools on Mac

GNU Toolchain

What is a *toolchain*?

When you download `XCode`, Apple installs a whole heap of tools for building `C++` (and other languages). This is nice, but as with all Apple products, if you dont use them in exactly the way that Apple foresaw, or you want more fine control over what is going on, things can be difficult (in my opinion).

After downloading `XCode`, you might expect that using `g++` at the command line would be using the `GNU C++` compiler - but actually it is using `clang++` - see the output below

```
patricklnoble$ g++ --version
Configured with: --prefix=/Library/Developer/CommandLineTools/usr --with-
    ↪ gxx-include-dir=/usr/include/c++/4.2.1
Apple LLVM version 9.1.0 (clang-902.0.39.1)
Target: x86_64-apple-darwin17.3.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

My favourite thing about `clang` is that it (in my opinion) often gives really helpful errors messages when you make errors in your `C++`. However, `GNU` is so ubiquitous in industry that I think from a practical point of view it is best to get used to working with `GNU` compilers. I dont have enough knowledge about performance/features etc - this is just a statement about which compiler/developer tools seem to me to be used most often. In addition, unless you are at the bleeding edge of language features, or really pushing performance hard, you are unlikely to notice much of a difference between the compilers.

`GNU` is the default on linux, but not on `mac`. So as with other tools, we will use `homebrew` to install it. I have installed `gcc6`, which is pretty recent - in industry, I wouldnt be suprised if many companies

were not yet using `gcc6`. In addition, I installed `gcc7` - it has essentially all `C++17` features - I don't think we will be using them, but if we can be future proof at no obvious cost, we might as well. To get both execute

```
1 patricklnoble$ brew install gcc@6
2 patricklnoble$ brew install gcc@7
```

where I havent shown all the output. This installed both (as well as some other dependencies [1]). As usual, `brew` installs `symlinks` in `/usr/local/bin` which all point to `brew`'s standard installation directory `/usr/loca/Cellar`. To use these compilers, we use `gcc-6`,`gcc-7` for the `C` compilers and `g++-6`,`g++-7` for the `C++` compilers.

[1] see https://formulae.brew.sh/formula/gcc to see what they are