Separate Compilation

Rob Hackman

Winter 2025

University of Alberta

Table of Contents

Declarations and Definitions

We've created variables, functions, and structures. Everytime we've created one of these so far we've always both *declared* and *defined* it. There is a difference between definition and a declaration.

Declarations and Definitions

We've created variables, functions, and structures. Everytime we've created one of these so far we've always both *declared* and *defined* it. There is a difference between definition and a declaration.

- A *declaration* only asserts that an entity of a given type and name exists. You can declare an entity as many times as you like.
- A *definition* is the full details of an entity, and what it comprises. For variables and functions space is allocated. An entity can only be defined once.

Forward declarations, a necessity

Declarations allow us to make a promise to the compiler that some entity exists, so the compiler should allow us to refer to it in our code.

Consider the following two functions:

```c
void foo(int x) {
  x = x/2;
  printf("%d\n", x);
  if (x % 2 == 0) foo(x);
  else bar(x);
}
```

```c
void bar(int x) {
  if (x == 1) return;
  x = 3*x + 1;
  printf("%d\n", x);
  if (x % 2 == 0) foo(x);
  else bar(x);
}
```

Which function should we place first in our program? If we place `foo` first then it isn't aware `bar` exists. If we place `bar` first then it isn't aware `foo` exists.

Function calls — Where to go

When the compiler generates code for a function call, it doesn't need to know *where* that function's code is yet (the definition of that function). The compiler can generate placeholder code for jumping to the location in memory where that code resides. The *linker* fills in the placeholder so the code jumps to the right location.

The compiler, however, must be promised that such a function exists. This can be done by declaring the function ahead of time. We call this a *forward declaration*.

Fixing `foo` and `bar` with forward declarations

```
1  // Forward declaring bar
2  void bar(int x);
3
4  void foo(int x) {
5    x = x/2;
6    printf("%d\n", x);
7    if (x % 2 == 0) foo(x);
8    else bar(x);
9  }
```

```
10 void bar(int x) {
11   if (x == 1) return;
12   x = 3*x + 1;
13   printf("%d\n", x);
14   if (x % 2 == 0) foo(x);
15   else bar(x);
16 }
```

By forward declaring `bar`, the code for `foo` now knows that the function it calls exists.

The four steps of C/C++ compilation

So far when we've invoked gcc there have actually been four steps to building our executable, only one of them is really "compilation".

- The *C preprocessor* translates the text file into another text file, following any *preprocessor directives* in the code.
- The *compiler* translates the preprocessed C code into *assembly* code.
- The *assembler* translates the assembly code into binary *machine* code, producing a binary *object* file.
- The *linker* combines object files into a single executable.

Table of Contents

Preprocessor directives

Preprocessor directives are the lines of our code that begin with a `#`. They are not actual code for your program to execute, but rather instructions for how the preprocessor should translate your text file.

We've already seen the `#include` directive. All the `#include` directive does is copy and paste the contents of the specified file directly in place.

`#include <stdio.h>` finds the file `stdio.h` and copies its contents into your program.

Preprocessor directives — Definitions

We can define identifiers for the preprocessors use:

```
// Defines the preprocessor value MAX_SIZE
#define MAX_SIZE 1000
// All future occurrences (not in strings) of MAX_SIZE
// will be replaced literally with the text 1000

// Defines a preprocessor flag - used when
// the existence of the flag is enough
#define FLAG
```

Defining preprocessor constants with the compiler

We can also define preprocessor constants with compiler flags. Consider the following program, which is in the file `define.c`

```
int main() {
  printf("%d\n", PREX);
}
```

When compiled with the command `gcc define.c` a compilation error is produced that says `PREX` is not undeclared.

We can specify preprocessor constants with the `-D` flag, which takes the form `-D<identifier>` for simple definitions (we call these flags), or `-D<identifier>=<value>` for providing a value for the identifier.

Defining preprocessor constants with the compiler

We can also define preprocessor constants with compiler flags. Consider the following program, which is in the file `define.c`

```
int main() {
  printf("%d\n", PREX);
}
```

Defining preprocessor constants with the compiler

We can also define preprocessor constants with compiler flags. Consider the following program, which is in the file `define.c`

```
int main() {
  printf("%d\n", PREX);
}
```

If we compile this program with the command `gcc -DPREX=5 define.c` then the preprocessor constant `PREX` is defined with the value 5.

Defining preprocessor constants with the compiler

We can also define preprocessor constants with compiler flags. Consider the following program, which is in the file `define.c`

```
int main() {
  printf("%d\n", 5);
}
```

If we compile this program with the command `gcc -DPREX=5 define.c` then the preprocessor constant `PREX` is defined with the value 5.

Once the preprocessor has processed this file, all instances of `PREX` are replaced literally with 5.

`ifdef` and `ifndef`

We can also write simple conditionals that check if a preprocesor flag has been defined (or not defined).

These are written in the form:

```
#ifdef IDENTIFIER
<code to include only if IDENTIFIER is defined>
#endif
```

When the preprocessor sees this directive, if `IDENTIFIER` is a defined preprocessor flag then the code is included in the file the compiler will see. If it is *not* defined then all of the code between the `#ifdef` and `#endif` is omitted.

`#ifndef` does the opposite, only including the text if the identifier is *not* defined, and not including the text if it is.

`ifdef` example

Consider the following program, `debug.c`:

```
int main() {
  int x = 1, y = 2000;
  while (x < y) {
    #ifdef DEBUG
    printf("x is now: %d\n", x);
    printf("y is now: %d\n", y);
    #endif
    x = x*2;
    y = y+50;
  }
}
```

If compiled just with the command `gcc debug.c` the generated executable will not print anything, as the print statements were omitted by the preprocessor since the flag `DEBUG` was not defined.

`ifdef` example

Consider the following program, `debug.c`:

```
int main() {
  int x = 1, y = 2000;
  while (x < y) {
    #ifdef DEBUG
    printf("x is now: %d\n", x);
    printf("y is now: %d\n", y);
    #endif
    x = x*2;
    y = y+50;
  }
}
```

If compiled, instead, with `gcc -DDEBUG debug.c` then the `DEBUG` flag is defined, and the calls to `printf` are included.

Table of Contents

Large programs — a reality

The programs we've written in this class have been fairly small, and it hasn't been much of a problem to write them all in one file.

But real-world programs are often large, and having all the code in one giant file would be an unorganized mess.

Additionally, there are often components of a program that are reusable in other programs — like our `List` ADT.

So how do we break up a program into multiple files in C?

printArray

We've often printed out arrays to see their values to demonstrate C code.

Let's place our `printArray` function into its own file: `printArray.c`

```c
#include <stdio.h>
void printArray(int *arr, size_t len) {
  for (size_t i = 0; i < len; ++i) {
    printf("%d ", arr[i]);
  }
}
```

Compiling `printArray.c`

If we try to compile `printArray.c` with the command `gcc printArray.c` we get an error something like the following:

```
/usr/lib/gcc/.../Scrt1.o: In function `_start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

This is note a compilation error. This is a *linker* error. It's because when we invoke `gcc` by default we are running all four steps of building an executable, including linking.

The linker tries to find the `main` function, which an executable needs as its entry point. Since our code has no `main` function the linker produces an error.

Compiling without linking

A C file that has been compiled into machine code without linking it into a final executable is called an *object* file. An object file is your C code compiled into machine code, along with a bit of additional information for the linker which combines object files.

We can specify to `gcc` that we would like to only compile a file, and not link it, to generate an object file. We do so with the `-c` flag. So we compile our `printArray.c` file with the command

```
gcc -c printArray.c
```

Which produces a file `printArray.o`, the generated object file.

Writing a `main` that uses `printArray`

Now, we must write a C file that contains a `main` to build an executable. Let's write `main.c`

```c
int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  printArray(arr, sizeof(arr)/sizeof(int));
}
```

We try to compile the program into an object file with `gcc -c main.c`, but we get a warning

```
main.c: In function ‘main’:
main.c:4:3: warning: implicit declaration of function
      ‘printArray’ [-Wimplicit-function-declaration]
    4 |   printArray(arr, sizeof(arr)/sizeof(int));
      |   ^~~~~~~~~~
```

Declare before use

We should declare any functions we use before we use them (when it comes to variables we *must* declare them before use).

We don't need the full definition though, that can be found by the linker later. So we update our program

```
void printArray(int *, size_t);

int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  printArray(arr, sizeof(arr)/sizeof(int));
}
```

And we can generate our object file without warning. `gcc -c main.c` produces the object file `main.o`

Combining object files

To combine object files into a final exectuable we must invoke the linker. As `gcc` by default invokes the linker we can do so by running

```
gcc main.o printArray.o -o builtProgram
```

When `gcc` sees a file that ends with `.o` it assumes it is an object file and treats it as such. It runs only the linker to combine these two object files into a final executable named `builtProgram`

Building a module

So we take the code for our `List` ADT and place it in a C file `list.c` so that we can reuse our ADT in any program.

Our code for `List` contains many definitions, it contains definitions for the `List` structure itself and eight functions we wrote to use our ADT.

Does that mean that every file that wants to use our `List` ADT must declare all of these entities manually? Have we had to declare `printf` manually before we use it?

Table of Contents

Header files

Header files are files that include the declarations of entities our module provides. The files `stdio.h` and `stdlib.h` are header files.

We can write a header file for our `List` ADT. The contents of `List.h` would look something like this:

```
struct List;
size_t len(struct List*);
struct List *createList();
struct List *addToFront(struct List *, int);
struct List *removeItem(struct List *, size_t);
int ith(struct List *, size_t);
void setElem(struct List *, size_t, int);
void deleteList(struct List *);
```

Header files — providing the interface

Heade files don't need to declare *everything* in the module, only the things the client programmer needs.

Our `List.h` header for example omits the `struct Node` declaration and `deleteNode` function, since the client programmer does not need to know about the implementation of our `List` or about the helper function `deleteNode`.

In this way a header file should provide the `interface` we expect our client programmer to use. The client programmer should be able to use our module by using only the details provided to them in our header file. Our header file can include comments to describe the behaviour of function.

Including headers

We learned earlier that the `#include` directive simply instructs the preprocessor to copy and paste a files contents into your C program. This is how we can get access to the entities the module designer intended our program to have, by including the header.

When including header files for standard libraries we've used a preprocessor directive of the form

```
#include <stdio.h>
```

The angle brackets tell the preprocesor to look for this file in a location defined by the compiler, as the compiler is the one who provides the standard library implementations.

Including custom headers

Headers we write for our own modules will not be located in the same directories the compiler will search by default[1], so we include them a different way.
Including a file with double quotes instead of angle brackets tells the compiler to look in the current working directory, we can even write a relative or absolute path in the quotes.

```
#include "List.h"
```

Generally, we will use double quotes to include our own program's header files.

[1] We can include our own directories or headers in the include path that is searched. Read up on the `CPATH` and `C_INCLUDE_PATH` environment variables if you're curious

Using our custom module

Let's use our custom module. We move all the code for our `List` ADT into a file `List.c` that contains no `main` function, and we've written `List.h` to provide the interface.

Let's write a program `printLargest.c` that reads every integer from the input stream and once EOF has been reached prints out the largest number read in.

The printLargest program

```c
#include "list.h"
#include <stdio.h>

int main() {
  int x = 0;
  struct List *l = createList();
  while (scanf("%d", &x) == 1) {
    addToFront(l, x);
  }
  int maxSoFar = ith(l, 0);
  size_t maxInd = 0;
  for (size_t i = 0; i < len(l); ++i) {
    if (ith(l, i) > maxSoFar) {
      maxSoFar = ith(l, i);
      maxInd = i;
    }
  }
  printf("The largest number entered was %d\n", maxSoFar);
  deleteList(l);
}
```

Creating a `List` on the stack

What if we wanted to create a `List` on our stack? We might try writing a program, `onStack.c`, like so:

```
1  #include "list.h"
2  int main() {
3    struct List l;
4    addToFront(&l, 1);
5  }
```

But when we try to generate an object file with the command `gcc stackList.c` we get a compilation error.

```
stackList.c: In function ‘main’:
stackList.c:3:15: error: storage size of ‘l’ isn’t known
    3 |   struct List l;
      |               ^
```

Incomplete types

At the point of compiling `stackList.c` the type `struct List` is known as an *incomplete* type. The compiler has been promised that the type `struct List` exists, but knows no details about it — including its size.

The code for `stackList.c` tries to place a `struct List` varible on the stack, for all stack variables the compiler must generate the code that allocates the appropriate space on the stack. How can the compiler generate the code to make space for our variable if it doesn't know how many bytes it is?

It can't. But in this case that's a good thing! We don't want the user creating their own `struct List` entities. They should only be created with our `createList` function. That way we know the client programmer can't create an invalid `struct List`.

Incomplete types — How to use them

Our code for `printLargest.c` works because while we use `struct List` as at type, we never ask the compiler to allocate one!

We only ever placed a pointer to a `struct List` on the stack, and we only ever created that through the `createList` function that the compiler was promised exists. The compiler always knows how big a pointer is, so it has no problem making space for our pointer on the stack.

At no point in the code for `printLargest.c` did the compiler need to know the details of the details of a `struct List`.

Any code that requires the definition of `struct List` cannot compile as our header does not include the definition!

Encapsulation

The concept of *encapsulation* is that our ADTs should be black boxes which the client programmer does not need to know any details about, nor can they interfere with the implementation of our ADT.

We noticed before that the client programmer being able to modify the fields of a `List` structure was a problem. The example given was the client programmer modifying the `next` pointers inside our `Nodes` to point in a cyclical fashion, or even to point at `Nodes` stored on the stack!

If the `Nodes` in our `List` form a cycle our functions would cease to work. Also, if any pointers to `Nodes` stored in our `List` are stored on the stack then our `deleteList` function will crash.

Invariants and Encapsulation

In programming we often talk about *invariants*. An *invariant* is a property we require to hold true in order for our code to work properly. It is very hard to write programs of meaningful size without relying on invariants. It is a programmers job to ensure their code never violates their invariants.

Our `List` implementation has many invariants, one is that each `Node` pointer stored in our `List` is a pointer to a valid heap allocated `Node` or `NULL` (if the end of our list), another is that the `len` variable matches the length of our list. If the client programmer can modify the individual fields of our ADT then they may break our invariants.

Incomplete types for encapsulation

By providing only the declaration of `struct List` in our `List.h` header file we also stop the client programmer from being able to interact with our ADT *except* through the interface provided. Consider code that tries to modify our data type directly, it won't compile anymore! Consider:

```
#include "list.h"
int main() {
  struct List *l = createList();
  l->len = 10;
}
```

This code won't compile, as the compiler has no idea at this point what the fields of `struct List` are! It only knows the type exists!

Types that should be complete

What if we want to create a data type that the user *can* put on the stack. A simple data type that does not have any invariants, for example a data type to represent a point on a 2D euclidean plane.

```
struct Point {
  int x, y;
};
```

In order to be able to put a data type on the stack the client *must* have access to its definition. So for any type we want the client to be able to create freely, we must provide the definition by placing it in the header file.

Point.h

Consider the header file `Point.h` shown here

```
struct Point {
  int x, int y;
};

float euclidDist(struct Point, struct Point);
int manhattanDist(struct Point, struct Point);
```

Providing type definitions to the client programmer

In C++ it will actually be more common than not to provide the type definition in the header file for the client to access. There is good reason for this in C++.

Which brings us to...