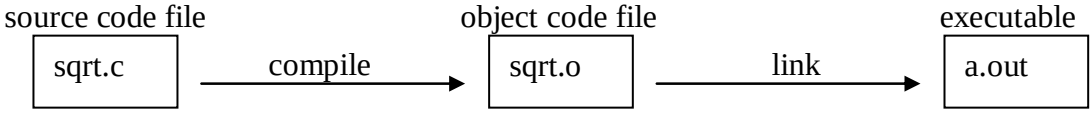
ECE 222 System Programming Concepts
Lecture notes – Building a program

What are the steps in building a program?



Source code is the C program you write. Running gcc on it compiles it into object code. To demonstrate:

```
gcc -c sqrt.c
```

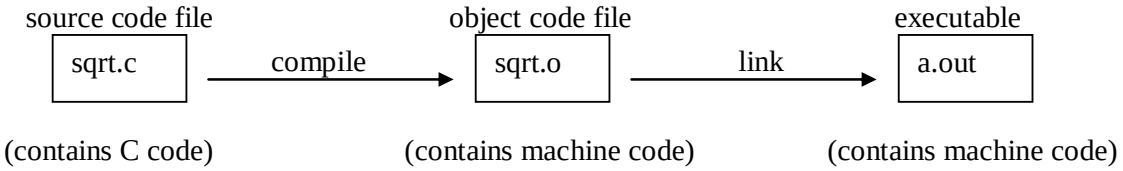
This tells gcc to stop after the compile stage, and do not proceed to linking.

An object code file contains machine code (sometimes also called object code), instructions that can run on the processor. But an object code file cannot be directly executed. Object code files must be linked to become an executable program that can be run. Linking is the process of bringing together multiple pieces of object code, and arranging them into one executable. Object code can come from multiple source code files, each compiled into its own object code file. To demonstrate:

```
gcc -c sqrt.c
gcc -c main.c
gcc sqrt.o main.o
```

Notice that without both object codes, we cannot create an executable. To demonstrate:

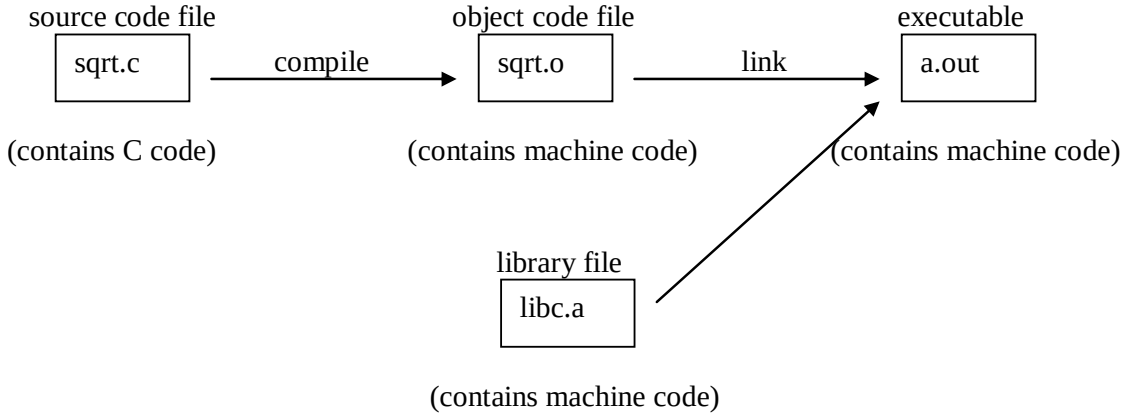
```
gcc sqrt.o          [no main() function found]
gcc main.o          [cannot find object code for SquareRoot() function]
```



An executable contains machine code, just like an object code file. However, it has a known starting instruction, whereas an object code files does not.

Note that object code files can be created from any source language, not just C. It is not uncommon to link object code files that were compiled from combinations of C, C++, java, pascal, and fortran, as well as files originally written directly in machine language.

When compiling, unless otherwise instructed, gcc will automatically proceed to linking, and then remove any object code files it created. When compiling multiple files, it is often convenient to instruct the compiler to keep object code files, so that only changed source code files need to be recompiled. Everything else only needs to be linked.



Linking also brings in object code from library files. The most commonly linked library file is /usr/lib/libc.a. It contains all the object code for the standard functions like printf(), fopen(), and strcmp(). To see what's inside:

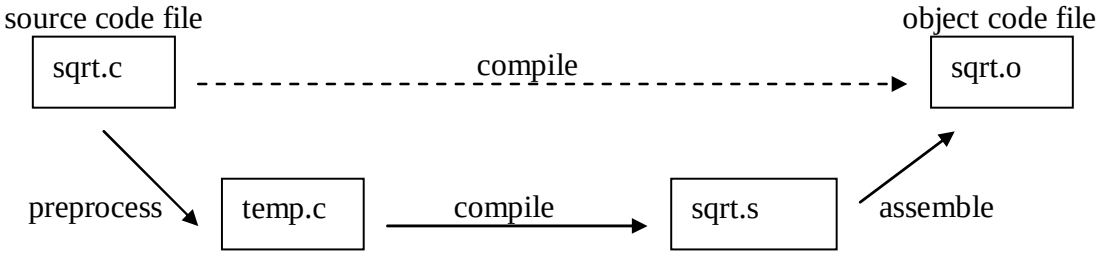
```
ls -al /usr/lib/libc.a
ar t /usr/lib/libc.a                [takes a while because it's so big]
ar t /usr/lib/libc.a | grep print   [to see printf.o and others]
ar t /usr/lib/libc.a | grep string  [to see strcmp.o and others]
```

Since these object codes are used over and over, they are permanently stored in library files, where they do not need to be recompiled every time. Library files are also handled differently during linking. Normally, they are dynamically linked into the executable, which means that the object code is not copied into the executable. Instead, when the program is run, if machine code from a library file is needed, it is loaded into memory directly from the library file. To demonstrate:

```
gcc main.c sqrt.c
ls -al a.out
gcc -static main.c sqrt.c
ls -al a.out
```

Notice how much larger the executable is when the library file is linked statically. This is because the object code from the library file is copied into the executable file.

What are the other implications of dynamic versus static linking? [Make class think about this.] Dynamically linked executables can run slower, because multiple library files may need to be accessed. Statically linked executables take up more space, and have redundant copies of the same exact machine instructions (object code) inside them.



Looking a little deeper into the compile process, it actually consists of three steps. The first step is preprocessing. Preprocessing provides mechanisms to support text substitutions. For example, it is often convenient to "name" a value that is used often during a program.

```
#define PI 3.14
```
[show example pre1.c]

When preprocessed, every occurrence of the string PI is replaced with 3.14. We can tell gcc to stop after preprocessing, and show us the result:

```
gcc -E pre1.c
```

Note that the occurrences of PI within quotes are not affected. We can save this to a file using the –o option we have already seen:

```
gcc -E pre1.c -o temp.c
```

Note that we can continue the compiling process from this point, by running gcc on temp.c, and everything still turns out the same.

What is the advantage to this sort of string substitution? Why not use a variable? [Make class think about it.] It saves storage space. A #define is NOT a variable, it takes up zero bytes of (data) memory.

Another common use for preprocessing is to copy source code that is needed over and over, usually from an include file.

```
#include "globals.h"
```
[show example pre2.c]

This copies the file globals.h, line for line, exactly in place of the #include line. To demonstrate:

```
gcc -E pre2.c
```

Any statement that begins with a # character is a preprocessor directive, NOT C code. Other common preprocessor directives are #if, #else, #endif, #ifdef and #ifndef to allow for control of what string substitutions are performed. For example, a common method to include or exclude debugging code in a program is to use preprocessing:

```
#define DEBUG 1              [show example pre3.c]
#ifdef DEBUG …
```

To demonstrate:

```
gcc -E pre3.c
edit pre3.c      [change 1 to 0 for DEBUG]
gcc -E pre3.c
```

The second step in our "zoomed-in" look at compiling is the actual compiling, which converts the C source code into assembly code. Once again, gcc can be instructed to stop here. For example:

```
gcc -S pre3.c
edit pre3.s
```

Assembly code is a human-readable version of machine code, much like ASCII symbols are a human-readable version of byte values. (Imagine trying to read text looking at the raw byte values.) Each machine code instruction is actually a set of byte values telling the processor what to do. The assembly instruction is a human-readable translation of those byte values.

The third step in our "zoomed-in" look at compiling is called assembling. During this step, assembly code is translated into machine code, to get it ready to run.

To summarize:

What are the stages in building a program?
What is preprocessing? Why is it helpful?
What is the difference between an object code file, a library file, and an executable?