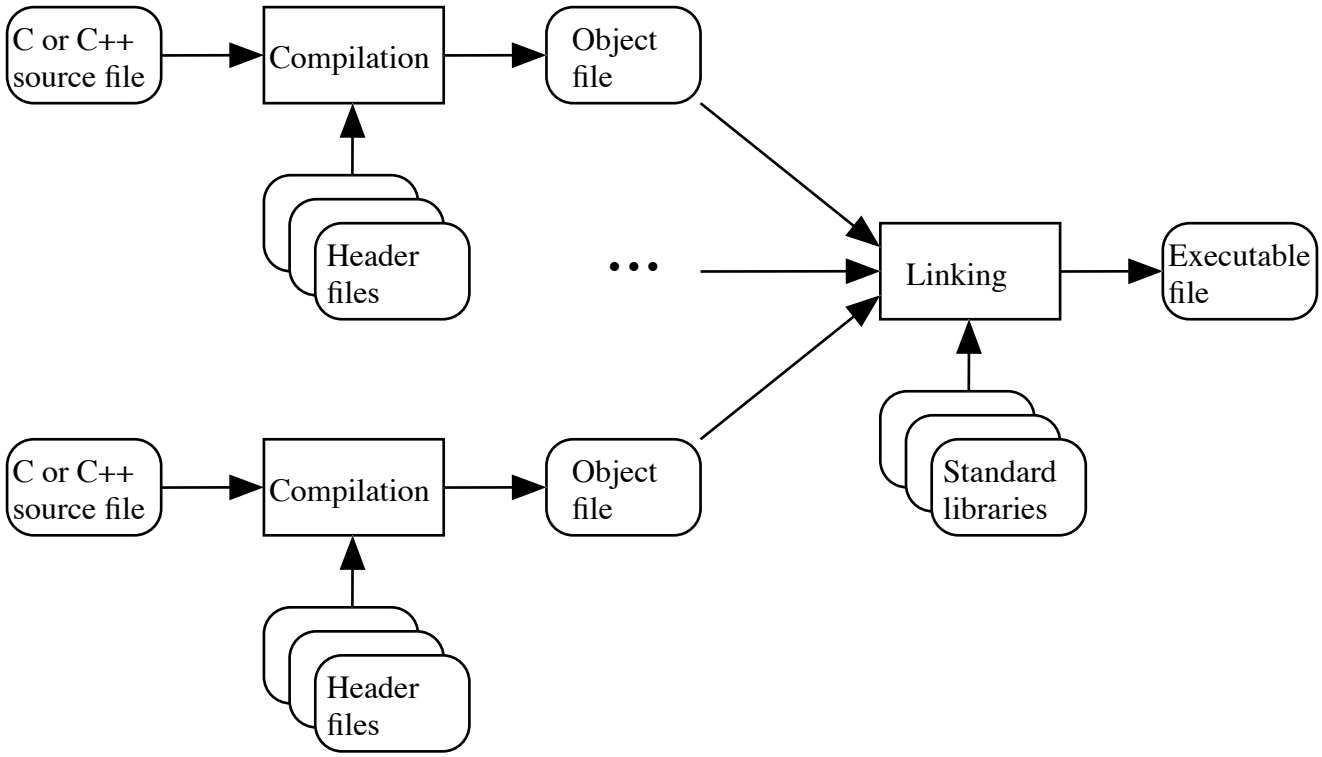
CPS222 - DATA STRUCTURES AND ALGORITHMS

Creating and Compiling C/C++ programs on a Unix/Linux System

A program consists of one or more source files that pass through a series of processing steps before becoming an executable program. The following diagram shows this process:



Compilation translates a *source file* (written in C or C++) into a binary *object file* for a specific platform (e.g. Power PC or Pentium or MIPS or ...). Header file code specified by #include directives is included into the file at the specified point during compilation. (Actually, compilation involves three distinct steps: preprocessing, compilation proper, and assembly; but ordinarily all three steps are done as part of one process)

Linking combines one or more object files with code from the standard libraries to produce a single *executable file* for a specific platform.

File Types

By convention, the final characters of the file's name (the suffix) specify what it contains.

1. Source files. The suffix indicates what language it is written in.

   ".c" - the file is a source file written in C
   ".cc" - the file is a source file written in C++.

   (Note: Depending on the platform, C++ source files may also have names ending in ".C" or ".cxx" or ".cpp" or ".c++". But ".cc" is the most commonly-used suffix).

2. Header files. Traditionally, the names of header files have ended in ".h". The new standard for C++ library headers is for the filename to have no suffix - e.g. <string> is the name of a standard C++ header that defines the string class. However, this is not yet consistently followed. User written headers and C library headers always have names that end in ".h".

3. Object files. The names of object files always end in ".o".

4. Executable file. By default, the linking process produces an executable file whose name is "a.out". This default behavior is frequently over-ridden by using the "-o" option on the command line to specify a more meaningful name, which - by convention - does not have any suffix.

Creating/Modifying Source (and Header) files

Source (and header) files are text files, and can be created or modified by any text editor. (However, a word processor should *not* be used to create source/header files, since the compiler does not understand the formatting codes word processors use.)

For example, the Gnome editor gedit can be used to create or modify a source/header file by using a command like the following:

```
gedit foo.cc
```

(If creating a new file, gedit will prompt to confirm that a new file is being created.)

Compilation and Linking

On Unix/Linux systems, a single command is used to invoke a "driver program" that, in turn, invokes the compiler and/or the linker as appropriate. Thus, the same basic command can be used to compile a source file or files, to link an object file or files, or to do both. A different command is used when the source files are written in C than is used when they are written in C++, both to invoke the appropriate compiler and to specify the use of the appropriate libraries when linking.

Traditionally, the C driver has been invoked by using the command-line command "cc" (compile C) and the C++ driver has been invoked by a command like "c++" or (less commonly) "cxx". On systems having the Gnu open-source versions of the compilers, the commands "gcc" and "g++" are used to invoke the Gnu versions of the C and C++ compilers (respectively), instead of the standard compilers furnished by the operating-system vendor. Since Linux systems use the Gnu compilers as their standard compilers, on our workstations, both "cc" and "gcc" are equivalent, and invoke the Gnu C compiler, and "c++" and "g++" are equivalent, and invoke the Gnu C++ compiler.

1. The following command could be used to translate the source file foo.cc (and any headers it includes) into an executable called a.out, by first compiling foo.cc and then linking the resulting object file.

   ```
   g++ foo.cc
   ```

   (Note: the compilation process produces an object file called foo.o; however, this file is automatically deleted when linking is complete.)

2. Since the name "a.out" is not very meaningful, it is possible to specify a different name by using the "`-o`" option. For example, the following command could be used to translate foo.cc into an executable called foo:

```
g++ foo.cc -o foo
```

3. If there is more than one source file involved, they can all be compiled and linked together with a single command. For example, the following command would compile foo.cc, bar.cc, and baz.cc, and would then link the object files to produce an executable called foo:

```
g++ foo.cc bar.cc baz.cc -o foo
```

   (Note: object files foo.o, bar.o, and baz.o are created by compilation, but they are automatically deleted when linking is complete.)

4. It is also possible to mix source and (previously-compiled) object files on the same command line. The driver uses the suffixes of the files to determine whether a file needs to be compiled or simply needs to be included in the final link. For example, the following command could be used to create an executable called foo by compiling the source files foo.cc and bar.cc, and then linking the resulting object files with the previously compiled object file baz.o:

```
g++ foo.cc bar.cc baz.o -o foo
```

   (Note: the object files foo.o and bar.o created by the compilation are automatically deleted after linking; however, baz.o is not deleted since it was specified in object form on the command line.)

5. Finally, there may be times when it is desired to compile a source file into an object file *without* going to the next step of linking to create an executable file. In this case, the "`-c`" option is used. For example, the following command could be used to compile the source file baz.cc into the object file baz.o:

```
g++ -c baz.cc
```

   (Note: in this case, of course, baz.o is not deleted!)

Executing the Translated Program

The translated program is executed by giving its name as a command - e.g. (for the above examples)

```
a.out
```

or

```
foo
```

If the current working directory is not part of the command path, it will be necessary to specify the current directory explicitly in the command - e.g.

```
./a.out
```

or

```
./foo
```