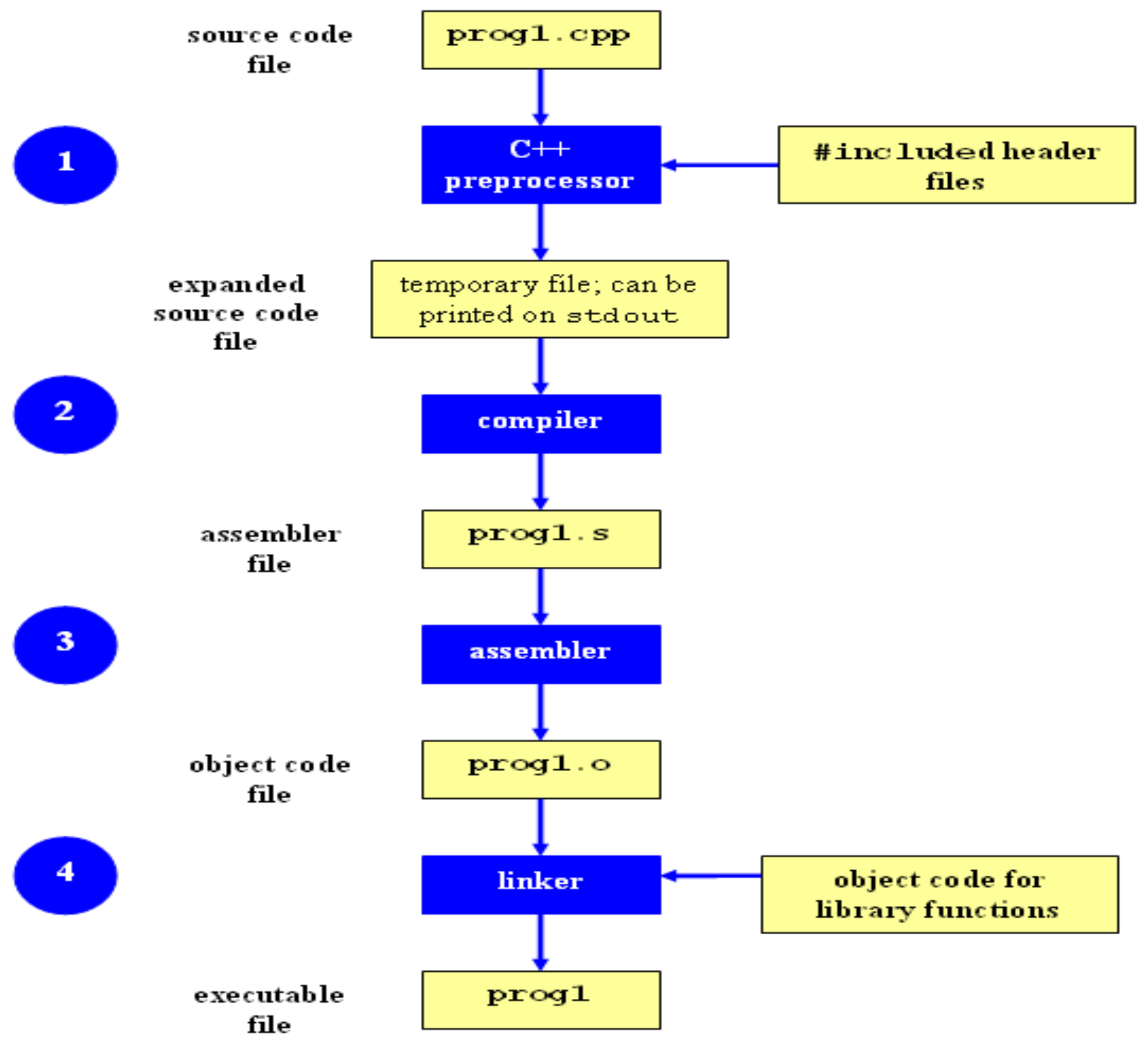
Makefile



source code file
prog1.cpp
1
C++ preprocessor
#included header files
expanded source code file
temporary file; can be printed on stdout
2
compiler
assembler file
prog1.s
3
assembler
object code file
prog1.o
4
linker
object code for library functions
executable file
prog1

Makefile

- **Separate Compilation**
  - What does the compiler do when it sees `#include` of a `.hpp` file in a `.cpp` file?
  - When you have many `.cpp` files, you will end up creating a corresponding `.o` file. In UNIX C++ compilers this is done with the `-c` option.
  - Any implementations of classes that are NOT part of the `.cpp` file, but appear in the `#include` files are not compiled

https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/makefile.html

Makefile

- Not necessary to recompile all the files when you make changes.
- You only need to recompile a small subset of the files.

https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/makefile.html

Makefile

Each entry has:

- a target (usually a file)
- the dependencies (files which the target depends on)
- and commands to run, based on the target and dependencies

https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

Makefile

Example:

```
Movie.o: Movie.cpp Movie.hpp Vector.hpp
	g++ -Wall -c Movie.cpp
```

The basic syntax of an entry looks like:

```
<target>: [ <dependency > ]*
	[ <TAB> <command> <endl> ]+
```

- The entry tells how to construct a target, and more importantly, *when* to construct the target

https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

Makefile

Exectuables as targets

The purpose of the **Makefile** is to create an executable. That target is often the first target in the **Makefile**.

Example:

```
p1 : MovieList.o Movie.o NameList.o Name.o
     g++ -Wall MoveList.o Movie.o NameList.o Name.o -o p1
```

Makefile

Executables as targets

If **p1** is the first target in your Makefile, then when you type "make", it will run the commands for **p1**.

- *Notice that the command to compile will create an executable called p1*
- *It uses the **-o** option to create an executable with a name other than **a.out***

Makefile

Macro definitions

```
OBJS = MovieList.o Movie.o NameList.o Name.o
CC = g++
DEBUG = -g
CFLAGS = -c -Wall -Werror -std=c++11 -pedantic
LFLAGS = -lsfml-graphics -lsfml-audio -lsfml-window -lsfml-system

p1 : $(OBJS)
    $(CC) $(LFLAGS) $(OBJS) -o p1
```

Macros are usually put at the beginning of a Makefile.

A macro has the following syntax:

```
<macro_name> = <macro_string>
```

Makefile

make `clean`

```
clean:
      \rm *.o *~ p1
```

- The backslash prevents "`rm`" from complaining
- Normally, you remove all `.o` files, all files ending in `~` (which are emacs backup files), and the name of the executable

Makefile

make `all`

```
all: p1 p2 p3

p1: Foo.o main1.o
	g++ -Wall Foo.o main1.o -o p1

p2: Bar.o main2.o
	g++ -Wall Bar.o main2.o -o p2

p3: Baz.o main3.o
	g++ -Wall Baz.o main3.o -o p3
```

Makefile

http://www.gnu.org/software/make/manual/make.html#Simple-Makefile