Tutorial 8
NWEN241
Modular Programming

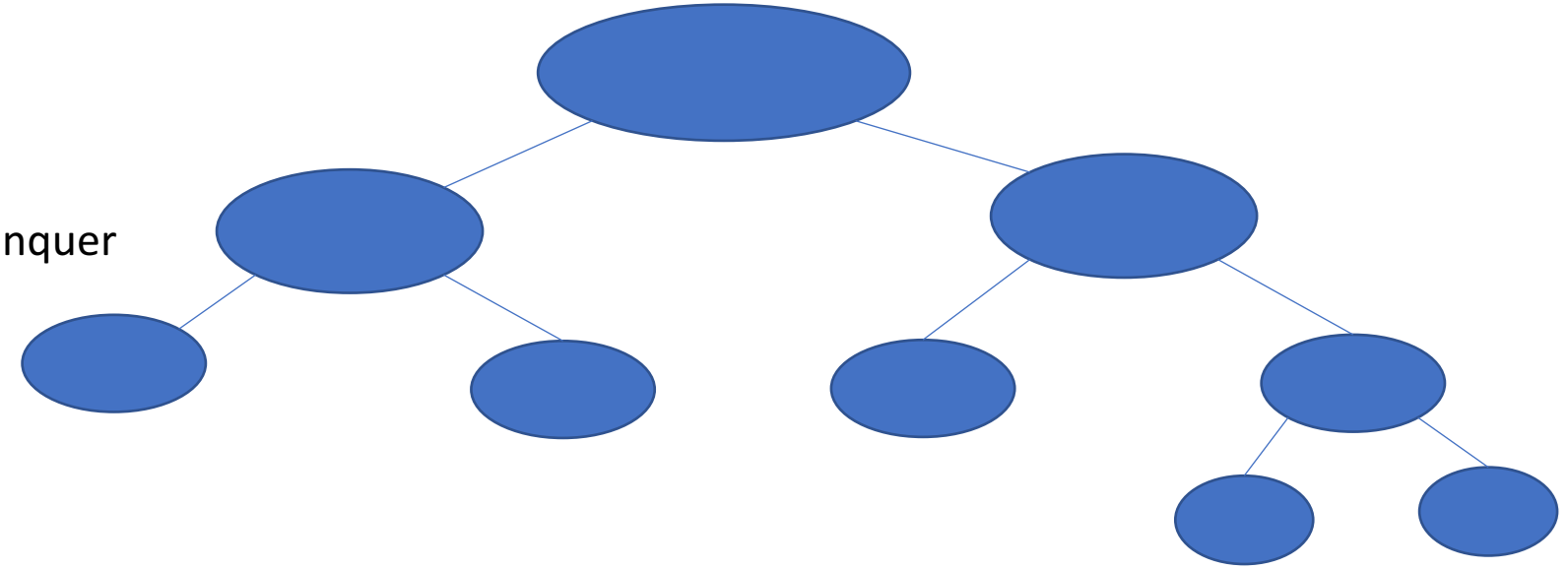
Kirita-Rose Escott

kirita-rose.escott@ecs.vuw.ac.nz

Content

- This tutorial is about writing large programs using **Modular Programming**
  - How do you solve a big/complex problem?
  - Divide it into small tasks and solve each task. Then, combine the solutions.

Divide and Conquer



  - In C, functions implement modules that perform specific tasks that we need in our solution.

Advantages of Using Modules

- Modules can be written and tested separately
- Modules can be reused
- Large projects can be developed in parallel
- Reduces length of program, making it more readable
- Promotes the concept of abstraction
  - A module hides details of a task
  - We just need to know what the module does
  - We do not need to know how it does it

C

Dividing Program into Multiple Files

- Each set of functions will go into a separate source file, eg. **foo.c**
- Each source file will have a matching header file – **foo.h**, which contains prototypes for the functions defined in **foo.c**
- Functions to be used ***only*** within **foo.c** ***should not*** be declared in **foo.h**
- **foo.h** will be included in each source file that needs to call a function defined in **foo.c**
- **foo.h** will also be included in **foo.c** so the compiler can check that the prototypes in **foo.h** match the definitions in **foo.c**

C++

Dividing Program into Multiple Files

- Each set of functions will go into a separate source file, eg. **foo.cc**
- Each source file will have a matching header file – **foo.hh**, which contains prototypes for the functions defined in **foo.cc**
- Functions to be used ***only*** within **foo.cc** ***should not*** be declared in **foo.hh**
- **foo.hh** will be included in each source file that needs to call a function defined in **foo.cc**
- **foo.hh** will also be included in **foo.cc** so the compiler can check that the prototypes in **foo.hh** match the definitions in **foo.cc**

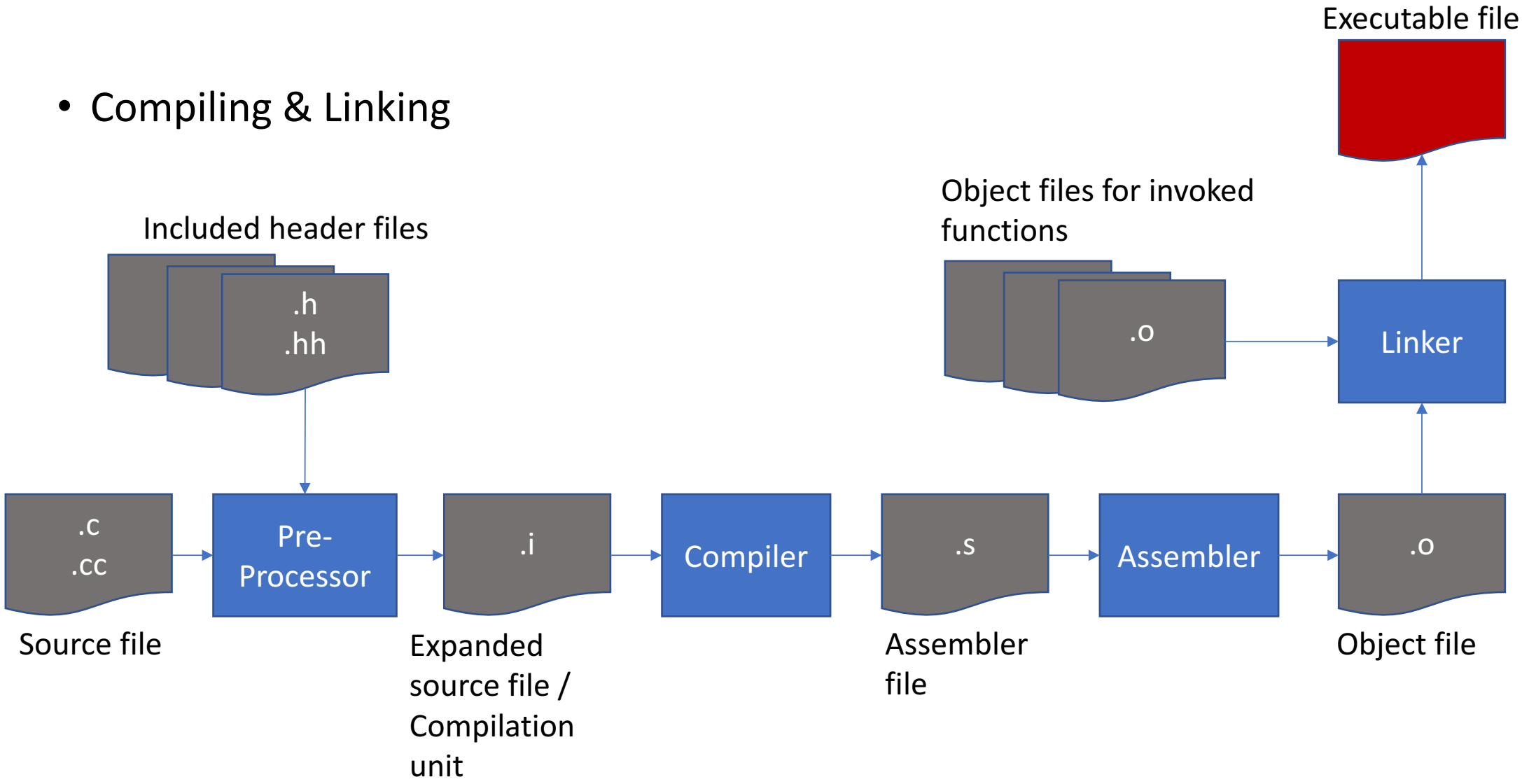
Dividing the Program into Multiple Files

- It is possible that there are other functions in the same file as **main**, as long as they are not called from other files in the program.

- Building a large program requires the same basic steps as building a small one:

Dividing the Program into Multiple Files

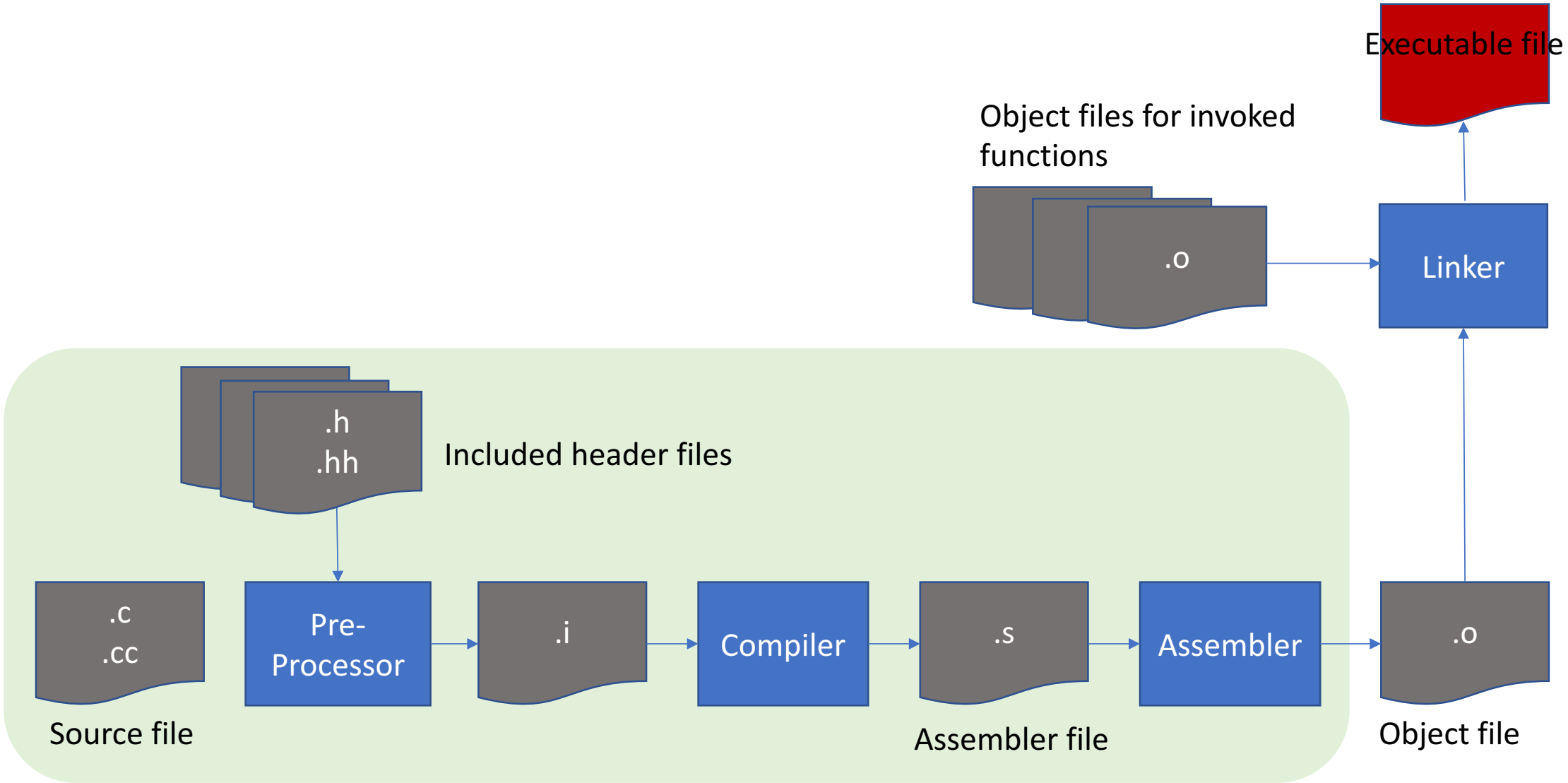
- Compiling & Linking



Building Multiple-File Program

- Each source file in the program must be compiled separately.
- Header (**.h**) files do not need to be compiled.
- A header file is automatically compiled whenever a source file that includes it is compiled.
- For each source file, the compiler generates a file containing object code, known as ***object files***; extension **.o** in UNIX and **.obj** in Windows.

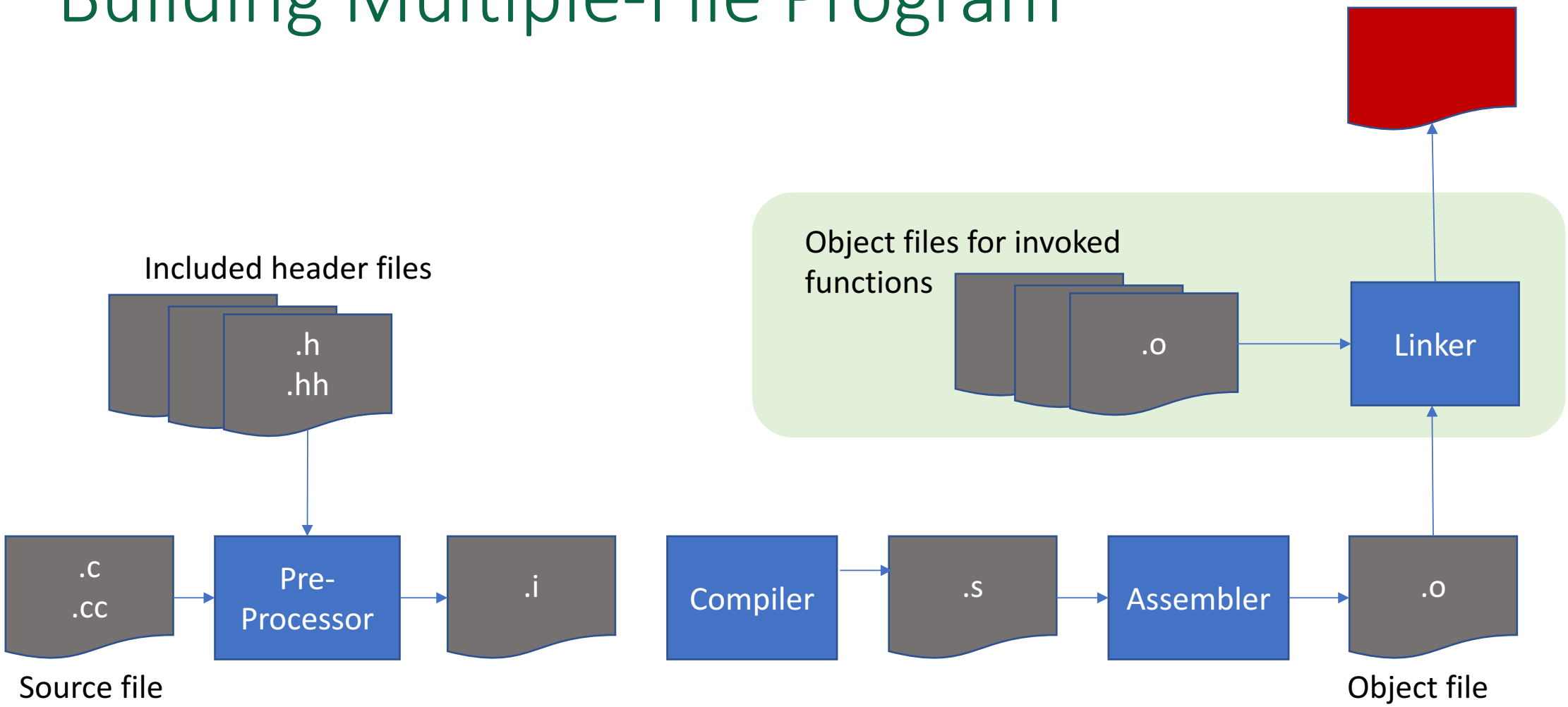
Building Multiple-File Program



Building Multiple-File Program

- The linker (ld) combines the object files created in the previous step – along with code for library functions – to produce an executable file.
- The linker is responsible for resolving external references left behind by the compiler.
- An external reference occurs when a function in a file calls a function defined in another file or accesses a variable defined in another file.

Building Multiple-File Program



Building Multiple-File Program

- Most compilers allow us to build a program in a single step.
- GCC command that builds **t8test:**

  gcc –o t8test t8test.c editor2.c

  The two source files are first compiled into object code.
- G++ command that builds **t10test:**

  g++ –o t10test t10test.cc editor2.cc

  The two source files are first compiled into object code.
- The object files are then automatically passed to the linker, which combines them into a single file.
- In the GCC command the **–o** option specifies that we want to name the executable file **t8test**.
- In the G++ command the **–o** option specifies that we want to name the executable file **t10test**.

Makefiles

- To make it easier to build large programs, UNIX originated the concept of the ***makefile.***

- A ***makefile*** not only lists the files that are part of the program, but also describes ***dependencies*** among the files.
  - Suppose that the file **foo.c** includes the file **bar.h**
  - Then **foo.c** “depends” on **bar.h**, because a change to **bar.h** will require us to recompile **foo.c**
  - Suppose that the file **foo.cc** includes the file **bar.hh**
  - Then **foo.cc** “depends” on **bar.hh**, because a change to **bar.hh** will require us to recompile **foo.cc**

C

Makefiles

- An example makefile for the ***t8test*** program from Assignment 2:

```
t8test: t8test.o editor2.o
	gcc -o t8test t8test.o editor2.o

t8test.o: t8test.c editor2.c editor2.h
	gcc -c t8test.c

editor2.o: editor2.c editor2.h
	gcc -c editor2.c
```

C++

Makefiles

- An example makefile for the ***t10test*** program from Assignment 2:

```
t10test: t10test.o editor2.o
        g++ -o t10test t10test.o editor2.o

t10test.o: t10test.cc editor2.cc editor2.hh
        g++ -c t10test.cc

editor2.o: editor2.cc editor2.hh
        g++ -c editor2.cc
```

C and C++

Makefiles

- There are three groups of lines; each group is known as a ***rule***, for example:

```
t8test: t8test.o editor2.o
TAB gcc -o t8test t8test.o editor2.o

t10test: t10test.o editor2.o
TAB g++ -o t10test t10test.o editor2.o
```

- The first line in each rule gives a ***target*** file, followed by the ***file on which it depends.***
- The second line is a ***command*** to be executed if the target should need to be rebuilt because of a change to one of its dependent files.

Makefiles

- When the **make** utility is used, it automatically checks the current directory for a file called **Makefile** or **makefile**.
- To invoke **make**, use the command

  make *target*

  where *target* (optional) is one of the targets listed in the **makefile.**
- If no target is specified when **make** is invoked, it will build the target of the **first rule.**
- Except for this special property of the first rule, the order of rules in a **makefile** is arbitrary.

Why use Makefile?

- During the development of a program, it is rare that we need to keep recompiling all its files.
- To save time, the rebuilding process should recompile only those files that might be affected by the latest change.
- Assume that a program has been designed with a header file for each source file.
- To see how many files will need to be recompiled after a change, only need to consider two possibilities:
  - Source file changed
  - Header file changed

Rebuild When Source File Changed

- What happens when source files are changed?

C and C++

Clean Target

- Cleaning, or removing of object and executable files can be done by adding a target called **clean**:

```
clean:
	rm –rf *.o t8test
```

```
clean:
	rm –rf *.o t10test
```

- To remove all object and executable files, just run

```
make clean
```