NWEN 241
Writing large programs

Winston Seah

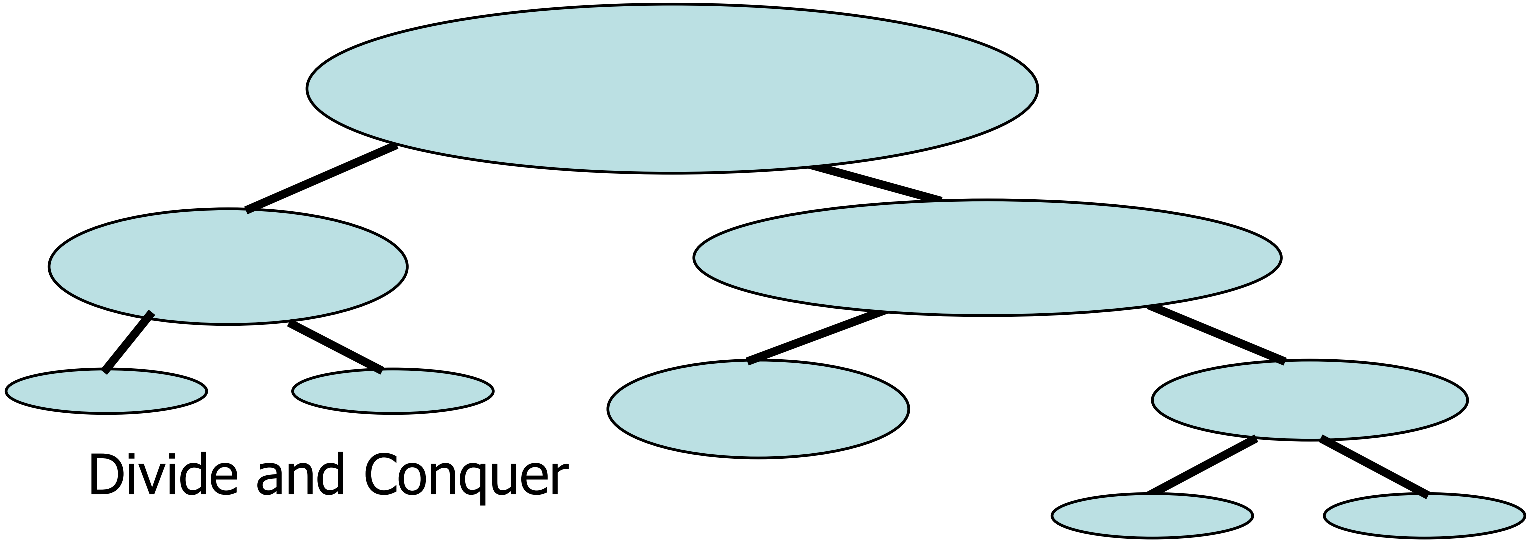
School of Engineering and Computer Science
Victoria University of Wellington

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui

CAPITAL CITY UNIVERSITY

Modular Programming

- How do you solve a big/complex problem?
- Divide it into small tasks and solve each task. Then, combine these solutions.



- 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

Abstraction

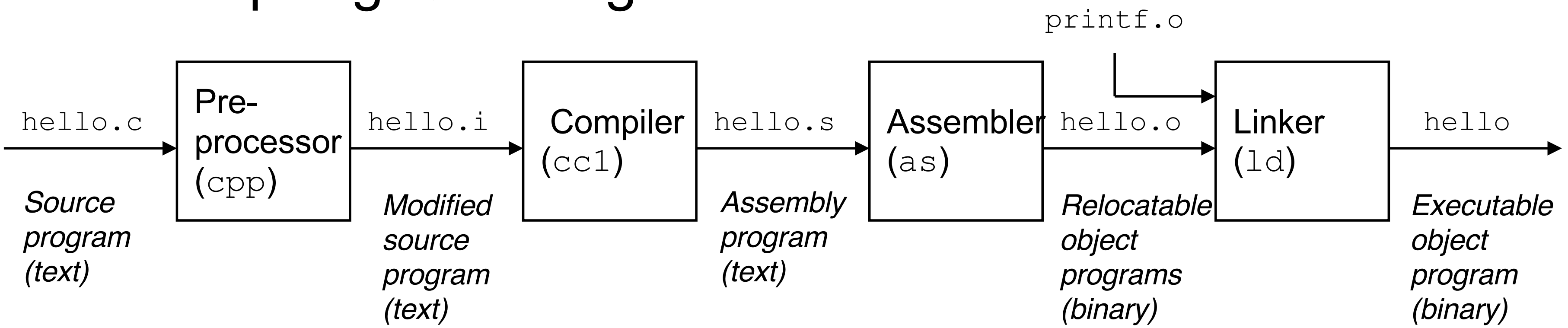
- procedural abstraction
  - separate what a function does from the details of how the function accomplishes its purpose
- data abstraction
  - separate the logical view of a data object (what is stored) from the physical view (how the information is stored)
- information hiding
  - protect the implementation details of a lower-level module from direct access by a higher-level module
- encapsulate
  - package a unit as a data object and its operators

Dividing program into multiple files

- Each set of functions will go into a separate source file, e.g. **`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`**.

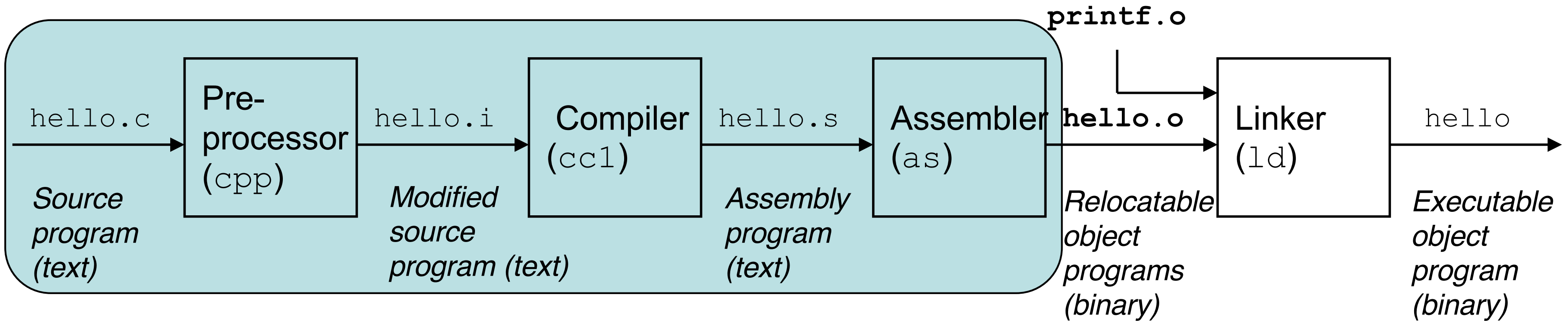
Dividing program into multiple files

- The **`main()`** function will go in a file whose name matches the name of the program.
- 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:
  - Compiling & Linking



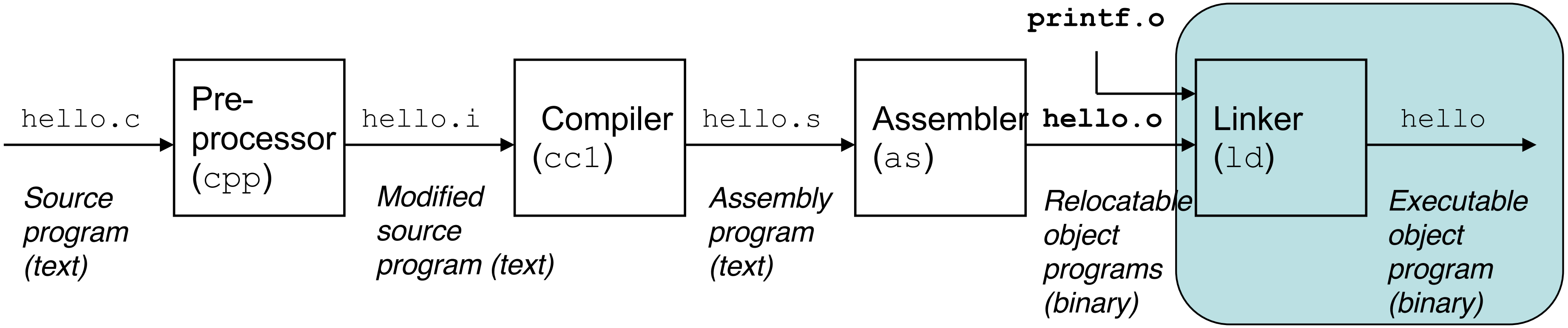
Building Multiple-file Program

- 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

- 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 also responsible for resolving external references left behind by the compiler.
- An external reference occurs when a function in one file calls a function defined in another file or accesses a variable defined in another file.



Building Multiple-file Program

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

```
gcc -o justify justify.c line.c word.c
```

- The three 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.
- The **`-o`** option specifies that we want to name executable file **`justify`**.

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`**.

Makefiles

A UNIX makefile for the **`justify`** program:

```
justify: justify.o word.o line.o
        gcc -o justify justify.o word.o line.o

justify.o: justify.c word.h line.h
        gcc -c justify.c

word.o: word.c word.h
        gcc -c word.c

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

Makefiles

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

```
justify: justify.o word.o line.o
        gcc -o justify justify.o word.o line.o
```

- The first line in each rule gives a ***target*** file, followed by the ***files 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.

- 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

- If a single source file, only recompile that file.
- Suppose that we decide to condense the **`read_char`** function in source file **`word.c`**:

```
int read_char(void)
{
  int ch = getchar();
  return (ch == '\n' || ch == '\t') ? ' ' : ch;
}
```

- This modification does not affect **`word.h`**, so we need only recompile **`word.c`** and relink the program.

Rebuild when header file changed

- Recompile all files that include the header file, since they could potentially be affected by the change. E.g. **`word.h`** is changed.

```
justify: justify.o word.o line.o
         gcc -o justify justify.o word.o line.o

justify.o: justify.c word.h line.h
         gcc -c justify.c

word.o: word.c word.h
         gcc -c word.c

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

Rebuild when function definition changed

- Suppose that we modify the function **`read_word()`** so that it returns the length of the word that it reads. Assume it previously returns nothing.
- First, we change the prototype of **`read_word`** in **`word.h`**:

```
int read_word(char *word, int len);
```

- Then change the definition (code) of the function in **`word.c`** (see next slide).

Rebuild when function definition changed

```
int read_word(char *word, int len)
  {
    int ch, pos = 0;

    while ((ch = read_char()) == ' ')
      ;
    while (ch != ' ' && ch != EOF) {
      if (pos < len)
        word[pos++] = ch;
      ch = read_char();
    }
    word[pos] = '\0';
    return pos;
  }
```

Rebuild when function definition changed

- Finally, we modify **`justify.c`** by changing **`main()`**:

```
int main(void)
{
  char word[MAX_WORD_LEN+2];
  int word_len;

  clear_line();
  for (;;) {
    word_len = read_word(word, MAX_WORD_LEN+1);
    …other codes…

  }
}
```

Rebuild when function definition changed

- Once changes have been done to **`justify.c`**, **`word.c`** and **`word.h`**, we can manually rebuild **`justify`** by recompiling **`word.c`** and **`justify.c`** and then relinking.
- The **`make`** utility does it automatically as follows:
  1. Build **`justify.o`** by compiling **`justify.c`** (because **`justify.c`** and **`word.h`** were changed).
  2. Build **`word.o`** by compiling **`word.c`** (because **`word.c`** and **`word.h`** were changed).
  3. Build **`justify`** by linking **`justify.o`**, **`word.o`**, and **`line.o`** (because **`justify.o`** and **`word.o`** were changed).