
22.5HV2

SOFTWARE ENGINEERING II

Separate compilation and make



Aims

○ In this unit we will consider the following topics:

- Modular programming
- Separate compilation
- `extern` and `static` modifiers
- Header files
- `make`
- Creating libraries using `ar`



Modular programming

- **Most of the programs that we have considered so far have been small. In practice, most programs that you will write will be much bigger.**
- **As programs grow in size, it becomes sensible to split them up into different sections called *modules*, where each module is stored as a separate file.**
 - **This idea follows from the practice of splitting the code itself into a number of functions.**
 - **The modules will contain different functions, together with any declarations of structures and shared variables.**
 - **It is possible to solve any programming problem by a single file program containing no functions other than `main()`. However, this program will be difficult to write as a team effort, maintain and update, and reuse in later applications.**



Modules

- **A module is a collection of functions or user defined data types (structures and classes) that perform *related* functions.**
 - **If a team of programmers is required to work on a problem, then each programmer will be assigned to a particular part of the problem and will create his/her own module(s).**
 - **This approach ensures independence between different modules and each programmer only has to worry about the internal details of their own code.**
 - **The programmer can test and document their code independently.**
 - **Once completed and tested, each module can be *used* by other programmers.**
 - **As many modules are applicable in different applications, this approach encourages software reuse.**
-
-



Public and private parts of modules

- This description of modules outlines a particular feature of this approach:
 - The details of a module's *implementation* are only of concern to the programmer responsible for the module, whilst the only the *operation* of the module is of interest to the programmer using it.
 - As we shall see, this principle leads us to have *public* and *private* parts to a module.
 - Before we discuss this, we will consider the idea of separate compilation.



Separate compilation

- **As we will be dividing our program over a number of different modules, it becomes necessary to be able to compile the modules individually.**
 - **Although we may have a number of modules, we will have only a single executable file, that we use to "run" the program.**
 - **When we compile the modules separately, we compile them to *object code*.**
 - **The final executable file is constructed by *linking* these object files together.**
 - **To illustrate this, let us look at a very simple example:**



A separately compiled program

- Consider the following example (adapted from Oualline p414):

```
#include <iostream.h>
extern int counter;
extern void inc_counter(void);
void main()
{
    int i;
    for (i=0;i<10;i++) {
        inc_counter();
        cout << counter << endl;
    }
}
```

main.cc

```
int counter=0;
void inc_counter()
{
    ++counter;
}
```

count.cc



Separate compilation

- To compile this program, we must do the following:

```
nannu% g++ -c count.cc  
nannu% g++ -c main.cc  
nannu% g++ -o exec main.o count.o
```

- The compilation option `-c` takes a C++ source file and creates the object code file. Object files are given a `.o` extension.
- The option `-o` takes the object code files `count.o` and `main.o` and links them to create the executable file `exec`.
- It is also possible to compile and link a multi-file program in one command as follows:

```
nannu% g++ -o exec main.cc count.cc
```



The extern modifier

- The file `main.cc` contained a couple of declarations before the function `main()`:

```
extern int counter;  
extern void inc_counter(void);
```

- These declarations are necessary as the integer `counter` and the function `inc_counter()` are defined in file `count.cc`, and the compiler must be told about them before their use in `main.cc`.
- The modifier `extern` tells the compiler that the definitions of these are not contained in this file.
- Remember the definition of `counter` will allocate storage space for its value and the definition of `inc_counter` will specify the function code.



The extern modifier

- Without the `extern` modifier, the declaration of an integer called `counter` in `main.cc` would have *defined* a second variable of that name.
 - This problem would be identified on linking and would result in a "multiple definition of 'counter'" error message.
- In fact we can get away with leaving out the `extern` modifier for the function `inc_counter()` as the lack of a modifier in `count.cc` means that this function can be used by other files.
 - The rules for this are given as . . .



Access modifiers

Modifier	Meaning
<code>extern</code>	Variable/function is defined in another file.
<code><blank></code>	Variable/function is defined in this file and can be used in other files.
<code>static</code>	Variable/function is local to this file.

- As we shall see later, the `static` keyword has more meanings than shown here. In this context, it allows us to create *private* code, that cannot be accessed outside its module.
- Conversely the `<blank>` (i.e. no specified modifier) modifier will implement a *public* interface, by allowing use outside the module.



The static modifier

- The `static` modifier can be demonstrated as follows:

```
#include <iostream.h>
void inc_counter(void);
void disp_counter(void);
void main()
{
    int i;
    for (i=0;i<10;i++) {
        inc_counter();
        disp_counter();
    }
}
```

main.cc

```
#include <iostream.h>
static int counter = 0;
void inc_counter()
{
    ++counter;
}
void disp_counter()
{
    cout << counter << endl;
}
```

count.cc



The static modifier

- By declaring the variable `counter` in `count.cc` as `static`, we have made it *private* to this file only.
 - Hence we have had to provide a function `disp_counter()` in `count.cc` that we can use to display `counter`'s value.
 - As we are using `cout` and `endl` in `count.cc`, we now have to include the header file `iostream.h`.

```
#include <iostream.h>
void inc_counter(void);
void disp_counter(void);
void main()
{
    int i;
    for (i=0;i<10;i++) {
        inc_counter();
        disp_counter();
    }
}
```

main.cc

```
#include <iostream.h>
static int counter = 0;
void inc_counter()
{
    ++counter;
}
void disp_counter()
{
    cout << counter << endl;
}
```

count.cc



22.5

The `static` modifier

- A consequence of declaring `counter` in file `count.cc` as being `static`, is that we can only use this variable as the programmer who wrote `count.cc` intends.
 - In other words we can only use the public functions `inc_counter()` and `disp_counter()` to increment and display `counter`'s value.
 - Any attempt to assign a value to `counter` in `main.cc` will be disallowed by the compiler.
 - This is an important principle, and we shall discuss it later when we look at classes.



Header files

- In practice it is best to place information that is to be shared between modules in a *header file*. The convention is that header files use a `.h` file extension.
- The header should contain all the public information:
 - Comments indicating the the module function and available functions/variables.
 - Public structures (and as we shall see, classes).
 - Common constants.
 - Prototypes of public functions.
 - `extern` declarations of public variables.



Example: A rational module

- Consider the following header file for the rational number example. For simplicity, we have restricted the implementation to arithmetic operators and output:

```
// rational.h by Stuart Clarke Nov 1998
// Declarations required to implement manipulation
// of rational numbers.
struct rational {
    int num, den;
};
rational operator+(const rational&, const rational&);
rational operator-(const rational&, const rational&);
rational operator*(const rational&, const rational&);
rational operator/(const rational&, const rational&);
ostream& operator<<(ostream&, const rational&);
```



The header file `rational.h`

- In this case, the header file contains the structure declaration for the new type `rational`, and the function *prototypes*.
 - This represents the *interface* to the module. It outlines what is available to the user of the module. In this case a user defined data type, `rational`, and a number of functions (overloaded operators).
- The implementation of the module is contained in the file `rational.cc`.
 - This file contains the function *definitions*.



rational.cc

```
#include <iostream.h>
#include <stdlib.h>
#include "rational.h"
rational operator+(const rational& a, const rational& b) {
    rational c={a.num*b.den+b.num*a.den, a.den*b.den};
    return c;
}
rational operator-(const rational& a, const rational& b) {
    rational c={a.num*b.den-b.num*a.den, a.den*b.den};
    return c;
}
rational operator*(const rational& a, const rational& b) {
    rational c={a.num*b.num, a.den*b.den};
    return c;
}
rational operator/(const rational& a, const rational& b) {
    rational c={a.num*b.den, a.den*b.num};
    return c;
}
ostream& operator<<(ostream& out, const rational& a) {
    if ( a.num*a.den < 0.0 ) out << "-";
    out << abs(a.num) << "/" << abs(a.den);
    return out;
}
```



The implementation file `rational.cc`

- This file includes the header files `iostream.h` and `stdlib.h`, due to the use of `cout` and `abs()`. It also includes the header file `rational.h` to get the declaration of the type `rational`.
- `rational.h` is placed in quotes to tell the compiler to look in the current directory.
- We have overloaded the stream output operator `<<` in this file - more of this later when we look at classes.

```
#include <iostream.h>
#include <stdlib.h>
#include "rational.h"
rational operator+(const rational& a,const rational& b) {
    rational c={a.num*b.den+b.num*a.den,a.den*b.den};
    return c;
}
rational operator-(const rational& a,const rational& b) {
    rational c={a.num*b.den-b.num*a.den,a.den*b.den};
    return c;
}
rational operator*(const rational& a,const rational& b) {
    rational c={a.num*b.num,a.den*b.den};
    return c;
}
rational operator/(const rational& a,const rational& b) {
    rational c={a.num*b.den,a.den*b.num};
    return c;
}
ostream& operator<<(ostream& out, const rational& a) {
    if ( a.num*a.den < 0.0 ) out << "-";
    out << abs(a.num) << "/" << abs(a.den);
    return out;
}
```

`rational.cc`



The driver program `ex1.cc`

- We can use the code contained in the files `rational.h` and `rational.cc` in a program `ex1.cc`.

```
#include <iostream.h>
#include "rational.h"
void main()
{
    rational    a={22,7}, b={1,3};
    cout << a << " + " << b << " = " << a+b << endl;
    cout << a << " - " << b << " = " << a-b << endl;
    cout << a << " * " << b << " = " << a*b << endl;
    cout << a << " / " << b << " = " << a/b << endl;
}
```



Compiling the pieces

- The following commands can be used to compile and execute the program:

```
nannu% g++ -c rational.cc
nannu% g++ -c ex1.cc
nannu% g++ -o ex1 ex1.o rational.o
nannu% ex1
22/7 + 1/3 = 73/21
22/7 - 1/3 = 59/21
22/7 * 1/3 = 22/21
22/7 / 1/3 = 66/7
nannu%
```

- Suppose that we change the details of the code in `ex1.c` . . .



ex1.cc

```
#include <iostream.h>
#include "rational.h"
void main()
{
    rational  a, b, c;
    char  op;
    cout << "Enter two rational numbers as ";
    cout << "<num1> <den1> <num2> <den2>: ";
    cin >> a.num >> a.den >> b.num >> b.den;
    do {
        cout << "Enter an operation: <+> <-> <*> </>: ";
        cin >> op;
    } while (op!='+' && op!='-' && op!='*' && op!='/');
    switch (op) {
        case '+': c = a+b; break;
        case '-': c = a-b; break;
        case '*': c = a*b; break;
        case '/': c = a/b; break;
    }
    cout << a << " " << op << " " << b << " = " << c << endl;
}
```

Changing the application

- As the code in `ex1.c` has changed, we need to recompile it.
 - However, the code in `rational.c` has not changed, so we need only link in the previously compiled code:

```
nannu% g++ -c ex1.cc
nannu% g++ -o ex1 ex1.o rational.o
nannu% ex1
Enter two rational numbers as <num1> <den1> <num2> <den2>:
22 7 1 3
Now enter an operation: <+> <-> <*> </>: +
22/7 + 1/3 = 73/21
nannu%
```



Separate compilation in practice

- In simple applications like this one, it is easy to see which files require compilation after a modification.
 - Practical applications can consist of 10's, 100's or even thousands of different files, and the implications of changing a single file may be difficult to determine.
 - In the past, programmers used shell files (UNIX) or batch files (DOS) containing the sequence of compilation commands.
 - However, this only saved the programmer some typing, the computer still had to recompile all the files. Sometimes, this took hours or even days.
 - This prompted the development of a way of specifying which files needed compilation after a modification - a method called `make`:
-
-



make

- **Make is a UNIX utility (copied by some windows compiler vendors) for specifying the rules of compiling an application involving several files.**
 - **These rules are contained in a file called `Makefile`.**
- **The `Makefile` contains the following sections:**
 - **Comments**
 - **Macros**
 - **Explicit rules**
 - **Default rules**



Makefile

○ Comments:

- ❑ Any line beginning with a # is a comment.

○ Macros:

- ❑ A macro has the format:

```
name = data
```

- ❑ *data* is the text that is substituted whenever make sees \$ (*name*).
- ❑ These macros are used to specify compile flags and pathnames.
- ❑ The idea of macros follows that of using constants in C++ programs - modifications need only be made once at the top of the Makefile.



Makefile

○ Macros . . .

- Typical uses of macros are to specify the compiler command and compile options:

```
CX = g++  
CFLAGS = -O
```

- The macro `CX` has been declared to be the `g++` compiler.
- The macro `CFLAGS` is specified as `-O` which will cause the compiler to optimise the code for speed.
- We can use these macros in the `Makefile` as follows:

```
$ (CX) $ (CFLAGS) -o ex1 ex1.o rational.o
```

- If we wish to change the compiler, say to use the native UNIX C++ compiler `CC`, or to change the flags to use the debugger option `-g`, we need only edit these macros and not each compile command.



Makefile

○ Explicit rules:

- ❑ Explicit rules tell make which commands are required to make the program. The usual form is:

```
target: source [source2] [source3]  
    command  
    [command]  
    . . .
```

- ❑ *target* is the name of the file to be created. It is made out the *source* file(s) on the right of the colon.
- ❑ The *command*(s) used to create *target* are listed (one per line) below. Each line must start with a <tab> character.



Makefile

○ Explicit rules . . .

- If we chose as our target, the executable file `ex1`, then the files upon which this depends would be `ex1.o` and `rational.o`. The command to generate this file (using macros) would be:

```
ex1: ex1.o rational.o
    $(CX) $(CFLAGS) -o ex1 ex1.o rational.o
```

- The object files `ex1.o` and `rational.o` would themselves be targets as:

```
ex1.o: rational.h ex1.cc
    $(CX) $(CFLAGS) -c ex1.cc

rational.o: rational.h rational.cc
    $(CX) $(CFLAGS) -c rational.cc
```



Makefile

○ Explicit rules ...

- The complete Makefile for our rational number example is:

```
# Makefile for rational example
CX = g++
CFLAGS = -O

ex1: ex1.o rational.o
    $(CX) $(CFLAGS) -o ex1 ex1.o rational.o

ex1.o: rational.h ex1.cc
    $(CX) $(CFLAGS) -c ex1.cc

rational.o: rational.h rational.cc
    $(CX) $(CFLAGS) -c rational.cc
```



Makefile

○ Explicit rules . . .

- Typing `make` will only compile any of the targets if it is necessary.
- This decision is determined by the relative *ages* of the files.
- In this example, if everything is "*up to date*", then typing `make` will do nothing.
- The correct order of ages for this to happen is:

<code>rational.h</code>	<code>oldest</code>
<code>rational.cc, ex1.cc</code>	<code>old</code>
<code>rational.o, ex1.o</code>	<code>oldish</code>
<code>ex1</code>	<code>newest</code>

- Any other order of ages will lead to the necessary compilation.
- Note that the age of the `.cc` files is implicit - a change in the header will necessitate modifying the source code.



Makefile

○ Explicit rules . . .

- The UNIX command `touch` can be used to fool the operating system into thinking that a file has just been modified:

```
nannu% touch rational.h
nannu% make
g++ -O -c ex1.cc
g++ -O -c rational.cc
g++ -O -o ex1 ex1.o rational.o
nannu%
```

- . . . and again:

```
nannu% touch ex1.cc
nannu% make
g++ -O -c ex1.cc
g++ -O -o ex1 ex1.o rational.o
nannu%
```



Makefile

○ Default rules:

- ❑ Make knows how to create certain files using built in rules:

```
ex1.o: rational.h
```

- ❑ make knows to create `ex1.o` from `ex1.cc` using the command:

```
g++ -c ex1.cc -o ex1.o
```

- ❑ It will only do this if `ex1.o` is *older* than `rational.h` or `ex1.cc`.
- ❑ Note that the default rule does not pick up the `CFLAGS` macro.



Makefile

- We can update our Makefile, to use the default rules and add a couple of other features:

```
# Makefile for rational example
CX = g++
CFLAGS = -O
OBJ = ex1.o rational.o

all: ex1

ex1: ex1.o rational.o
    $(CX) $(CFLAGS) -o ex1 ex1.o rational.o

ex1.o: rational.h

rational.o: rational.h

clean:
    rm ex1 $(OBJ)
```



Makefile

- We have provided a macro OBJ to record all the object files.
 - This macro is used to specify the dependencies for `ex1`, and is used in the `rm` command following the target `clean`.

- We have added two new targets, `all` and `clean`.
 - The target `all` allows us to have a `Makefile` for a number of programs. By placing the executable file name as a dependency for `all`, typing "`make all`" will cause all files to be compiled if necessary.
 - The target `clean`, is used to remove all the object and executable files. This is achieved by typing "`make clean`".



Makefile

- If we had a second example (`ex2.cc`) involving the rational code, we could include it in our Makefile:
- We could make a single program by typing:

```
nannu% make ex1
```

- Or make both by typing:

```
nannu% make
```

```
# Makefile for rational example
CX = g++
CFLAGS = -O
OBJ = ex1.o ex2.o rational.o

all: ex1 ex2

ex1: ex1.o rational.o
    $(CX) $(CFLAGS) -o ex1 ex1.o rational.o

ex1.o: rational.h

ex2: ex2.o rational.o
    $(CX) $(CFLAGS) -o ex2 ex2.o rational.o

ex2.o: rational.h

rational.o: rational.h

clean:
    rm ex1 ex2 $(OBJ)
```



Creating a library with ar

- The UNIX archive command `ar` allows us to build a library that can be used to collect a number of object files together.
- We can create a library for the rational number example as follows:

```
ar rv librational.a rational.o
```

- The optional flags `r` and `v` cause files to be inserted into the archive with replacement, and Verbose reporting respectively.
- A convention is that the library file must start with `lib` and end in the file extension `.a` (or `.so` for *dynamic* libraries).



Creating a library with ar

○ To make our rational library available for use by others, we need to place the header file `rational.h` and the archive file `librational.a` in appropriate places.

□ This will be the directories `/usr/include` and `/usr/lib` assuming that we have root permission.

□ If this is the case, we can use the rational library by including the header file `rational.h` in our source code, using the `<>` brackets:

```
#include <rational.h>
```

□ We can compile our code using the flag `-l` to include the rational library:

```
g++ -c ex1 ex1.cc -lrational
```



Creating a library with ar

- If we do not have root user privileges, but still wish to create a library of functions, we can use the flags `-I` and `-L` to specify the location of the include and library files respectively.

- If they are placed in the current directory ("`.`" in UNIX), the compile command would be:

```
g++ -c ex1 ex1.cc -I. -L. -lrational
```

- Otherwise, we could create the directories `include` and `lib` in our home directory (`$HOME` in UNIX) and use the compile command:

```
g++ -c ex1 ex1.cc -I$HOME/include -L$HOME/lib -lrational
```

- Some systems may provide environment variables to specify these paths.

