

## C++ in Unix: multiple source files

### Lecture 8

#### Separate compilation: Multiple object files

We now have a program that contains different parts, some part is written by the library writer, some part is written by the library user.

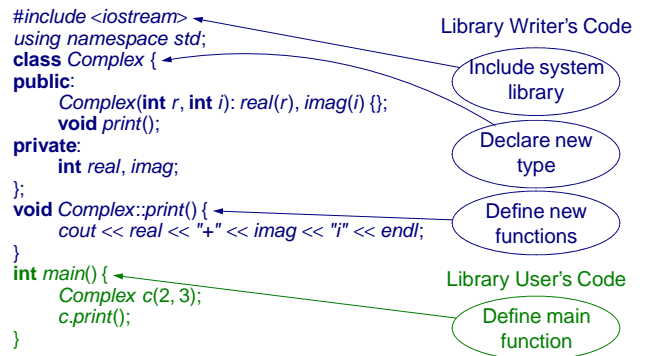
Now we need a way for the whole program to be broken down into multiple files, in such a way that each can be separately edited and compiled.

References:

- 3rd Ed. Sect. 2.4 (esp. Sect. 2.4.1), Sect 7.1.1, Ch 9
- Info page of (gcc), esp. **Invoking GCC** and **Link options** within it
- Reusable C++ book: Sect. 4.4.1 (the art of inlining)

PMOOP(0396A)

So far, we have been writing programs like this:



PMOOP(0396A)-8.1

## Programs with multiple source files

#### Two problems:

- When the program gets larger and larger, it becomes more and more difficult to maintain the program. Any small change requires a complete recompile.
- When the library user starts working, the library writer must stop: two people cannot work on the same file at the same time!

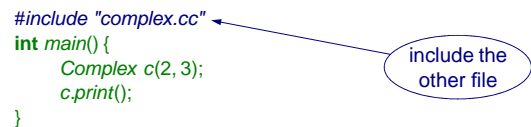
#### Our aim:

- We should be able to break a program into many portions and compile each independently.
- Each portion should be more or less self-contained, so that it is possible to reuse a part of a program without a lot of modifications.

PMOOP(0396A)-8.2

## First try: just break it

- Just break our program into two files.
- We have a file `comp.cc` containing just the main function, and a file `complex.cc` containing other functions and declaration. We add a line at the beginning of the `comp.cc` file, so that it looks like:



#### Problem

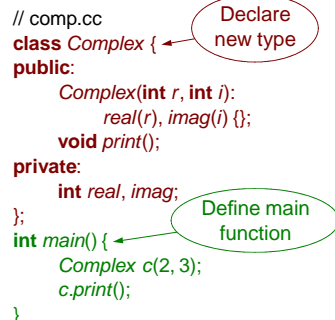
- We still need to compile the whole large program whenever anything is changed. The compilation process is not really changed at all.

PMOOP(0396A)-8.3

## Second try

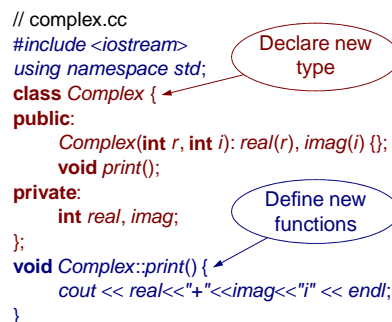
#### Separate compilation

- How about **not including** `complex.cc`? Then the compiler don't know about the `Complex` type, and don't know the member function `print`.
- Solution: we include the declaration in both `complex.cc` and `comp.cc`. The compiler will know the type `Complex` when we compile `comp.cc`, but will not need to compile everything else.



PMOOP(0396A)-8.4

## Making object files



To compile the program:

```
>g++ -c complex.cc
>g++ -c comp.cc
>g++ -o comp comp.cc \
complex.cc
>./comp
2+3i
```

The commands create `complex.o`, `main.o` and `comp` respectively.

- If we change a function in `complex.cc`, we don't need to recompile `comp.cc`, and vice-versa.

PMOOP(0396A)-8.5

### The need of headers

There is one problem of our previous scheme: what if we need to change the *complex* class? We need to make changes in **both** files! Worse yet, if we forget one, the compile won't even warn!

Better if we separate it into a third file, *complex.h*, and let both cc files include them. Then changes are need only for the complex.h file.

```
// complex.h
class Complex {
public:
    Complex(int r, int i): real(r), imag(i) {};
    void print();
private:
    int real, imag;
};
```

Only declare new type, no function definition

### The new cc files

Now we don't need to have the class declaration in the source cc files. Instead, just use `#include`.

```
// complex.cc
#include "complex.h"
#include <iostream>
using namespace std;
void Complex::print() {
    cout << real << "+" << imag << "i" << endl;
}

// comp.cc
#include "complex.h"
int main() {
    Complex c(2, 3);
    c.print();
}
```

Include header file: get Complex declaration

Note that user code is in *comp.cc*, while library code is split to common declarations in *complex.h* and implementation in *complex.cc*.

PMOOP(0396A)-8.6

PMOOP(0396A)-8.7

### What header file should contain

If something in the header file change, the user program must be recompiled. So we want to minimize the number of things in headers.

On the other hand, the header file must contain certain things. Without them, the compiler cannot generate the code for the user programs.

- Definition of public classes and their base-classes in the library.
- Definition of all library constants that user programs may use.
- Definition of all inline functions in the library that user programs can call. (What are inline functions?)
- Declaration of all functions and global variables of the library that user programs may use. (What are declarations?)
- All enumerated types of the library that user programs may use.

PMOOP(0396A)-8.8

### Sidenote: Inline functions

Some functions are so small that it is always better to include the code into the program rather than to call the function. For example:

```
void incr(int &x) { // Such function is usually seen in classes
    ++x;
}
void f() {
    int n = 10;
    incr(n); incr(n); ...
}
```

To call the function, the caller needs to allocate x in memory and push the address of n onto stack; while the function must increase the value in memory, remove the address from stack and then return back to the caller.

If the function is not called but directly embedded into f(), all that is needed is to increment n, and n can be in a register rather than in memory. It can even turn the two increments to one addition.

PMOOP(0396A)-8.9

### How to inline

You can suggest the compiler to use inlining for a function. There are two ways doing it. You can prefix the function definition by **inline**:

```
inline void incr(int &x) {
    ++x;
}
```

For member functions, you can write out the definition within the class:

```
class XXX {
public:
    void incr() { ++n; };
private:
    int n;
}
```

In either case, it is only a suggestion to the compiler. The compiler has the final decision whether to inline or not.

PMOOP(0396A)-8.10

### See it in action: get your hand dirty

If you want to know what has actually happened when you add inline, you can look at the assembly code generated by the compiler.

- Instead of running the compiler like `g++ -Wall inputfile`, run it like `g++ -Wall -S inputfile`. The compiler will generate assembly code instead of object file.
- By default, g++ do nothing on inline. But if you optimize your code, you'll see the difference. So run the compiler like `g++ -Wall -S -O inputfile` instead.
- A file with the extension `.s` will be created, and you can read the generated code there.
- If you want even more internal details, run the program in the debugger and use the 'disassemble' command.

PMOOP(0396A)-8.11

### The cost of inline

- If the function body is long, the code size increases on inline.
- This is not a problem in our previous example, since `incr()` is really short. It is so short that it is shorter than the corresponding function calling.
- However, if the function body is long (e.g., more than a dozen of lines), putting a copy of the function in the caller a lot of times will make the executable file large.
- If this is large enough, it can even make the program slow.

General guidelines: if the function is very short, or if the function is called only a few times throughout the whole program, the space overhead will not be large.

If the compiler find that it won't hurt, it uses inline automatically at higher optimization level.

PMOOP(0396A)-8.12

### Multiple definition: what must not be in headers

Variable definition and ordinary (i.e., non-inline) function definitions cannot appear more than once in the whole program. The following is **not** allowed in header files (more than one cc file include it).

```
void foo() { cout << l; }
int a;
```

Instead, use this:

**Header file:**

```
void foo();
extern int a;
```

**Implementation cc file:**

```
void foo() { cout << l; }
int a;
```

**Declares** the function and variable without compiling the functions or reserving space for the variable.

These actually generate code for the function and reserve space for the variable: **definition**.

PMOOP(0396A)-8.13

### Exactly once per compilation

We have seen ordinary function definition and variable definition can only appear once per program, i.e., summing over all compilation.

Something that can appear only once per compilation, i.e., per .o file:

- Each class can be defined only once per compilation.
- An inline function can be defined only once per compilation.
- A constant can be defined only once per compilation.
- An enumerated type can be defined only once per compilation.

Other things can appear more than once in the code. (e.g., typedef, extern variable declaration, etc.)

It might seem stupid to have them twice in the same compilation. But...

PMOOP(0396A)-8.14

### #include in headers

Suppose we want a library that uses `complex.h` in some case, e.g., a mathematical library that support complex numbers. Again, we want to separate the implementation, say to `mathlib.cc`.

We thus have two more files: `mathlib.cc` and `mathlib.h`. Question: should `mathlib.h` include `complex.h`?

- If `mathlib.h` does not include `complex.h`, then the user of `mathlib.h` must always remember to `#include "complex.h"` even if it does not use `Complex`.
- If `mathlib.h` does include `complex.h`, then the user of `mathlib.h` must remember **not** to `#include "complex.h"` even if it **does** use `Complex`!

Solution: `complex.h` should make sure that its definitions will only be compiled **once per compilation**.

PMOOP(0396A)-8.15

### Conditional compilation

One feature of C++ will help us to make sure something is never compiled twice: the preprocessor directives.

```
// complex.h
```

```
#ifndef _COMPLEX_H
#define _COMPLEX_H
```

Conditional: only if `_COMPLEX_H` is not defined

```
class Complex {
public:
    Complex(int r, int i): real(r), imag(i) {};
    void print();
private:
    int real, imag;
};
```

Define `_COMPLEX_H`, so that the declaration of `Complex` won't be compiled twice.

End conditional part

```
#endif
```

Now `mathlib.h` can safely `#include "complex.h"`.

PMOOP(0396A)-8.16

### Summary

- To make it easy to reuse functions and classes of a program, it is essential to make sure that each part can be separately compiled.
- The other part of the program sees only the header file. Whatever not mentioned in the header file is not for the main program to use.
- The header file should be wrapped within `#ifndef` so that other headers is safe to `#include` the header file.
- By adding "-c" flag to `g++`, it only compile the program to an object file, to be combined later to an executable later in another run of `g++`.

### Exercise

- Try to use separate compilation for our `WAVFile` class. Make sure it still compiles correctly.

PMOOP(0396A)-8.17