

The problem of single source file

Lecture 2

Separate compilation: Multiple source and object files

The main task of this course is to partition a large program into smaller, more manageable, logically separated pieces.

This involves two parts: physically separating the source file, and conceptually separating the functions.

This and the next few lectures offer advice for the first step.

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.

PMOOP(0396A)

PMOOP(0396A)-2.1

Main targets to separate files

What are the primary concerns when we separate the source files?

- Each file should be **small enough**.
- It should **reduce the amount of code that needs recompilation** when some code changes.
- It should **respect the division of labour** of the project. E.g., if two different developers are responsible for two functionalities of the program, they should be in different files.
- Each source file should have a **simple interface** that is used by other programs. I.e., it should be easy to understand what is provided by each source file, without going through the whole source file.
More about this later in the course: OOP really helps here.
- It should be possible to **reuse** some source files in another project.

PMOOP(0396A)-2.2

The problem of too much separation

On the other hand, separating source files should not be done too deeply:

- Each separated source file **requires some code** to glue them up. Too much separation will cause such gluing code to dominate the project.
- Compiling each source file has some **fixed cost**, so too much separation causes slower rather than faster compilation.
- It does not simplify the design. It can be very **difficult to get the overall picture of the program**.
We only see the trees, and can't see the forest.
- **Finding the dependencies** of the files become very time-consuming.
What are dependencies? We will see it soon.

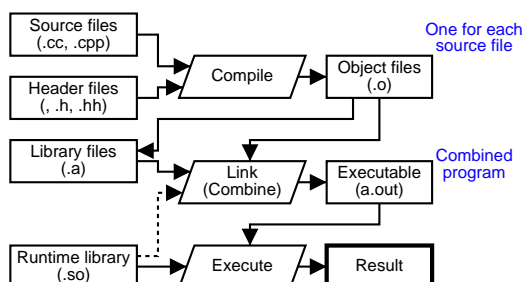
So separation should be done on basis of needs.

It is probably a bad idea to writing each function (or even each class, as we will talk about later) in a separate file.

PMOOP(0396A)-2.3

C++ source, object and executable

Let's see how multiple files in C++ works together.



Our previous working mode simply tells the compiler that the source files we specified are the only ones, so automatically link up the program and delete the intermediate .o files.

PMOOP(0396A)-2.4

The C++ #include directive

First let's solve one mystery about the language: what is `#include`.

- The `#include` directive is generally used to **get some objects defined**. E.g., if you want to use the standard C++ I/O stream library, you use `#include <iostream>`.
- The directive does a very simple thing: to **include the specified file into the source file**, as it were in the file.
- The file is searched in **standard path** if `<>` is used (as above), and in **current directory** if `" "` is used (like `#include "hi.h"`).
Some standard path to search: `/usr/include`, `/usr/include/g++-3`, etc.
- You can **add new search path** by `-I`, e.g.,

```
g++ -I /usr/X11R6/include myxprog.cc.
```

- Typically, the header file declare objects and functions contained in the library. We will soon see how to write such files.

PMOOP(0396A)-2.5

A simple example

Let's see a simple toy program and see how to break it up.

```
#include <iostream>
#include <cmath>
using namespace std;

int i = 2; // Used only by g()
int g() {
    return ++i;
}

float f(int a, int b) {
    float sum = 0;
    for (int i = 1; i < a; ++i)
        sum += sqrt(i);
    return sum;
}

int main() {
    cout << f(g(), g()) << endl;
}
```

When we break up our program, we usually consider one part of the program the **user** program, and the others as **library** routines.

Suppose we want to break up the functions *f()* and *g()* into its own source file `myfuncs.cc` as library. The function *main()* constitute the user program that uses the library.

PMOOP(0396A)-2.6

First try: let's just break it

Our first try: just break it, and use `#include` to glue them up.

```
// main.cc
#include <iostream>
#include <cmath>
#include "myfuncs.cc"
using namespace std;

int main() {
    cout << f(g(), g())
        << endl;
}

// myfuncs.cc
int i = 2; // Used only by g()
int g() {
    return ++i;
}

float f(int a, int b) {
    float sum = 0;
    for (int i = 1; i < a; ++i)
        sum += sqrt(i);
    return sum;
}
```

This works, but it is no good because (1) **every line of code needs re-compilation** for each modification, and (2) we don't know **what functions supposed to be used outside myfuncs.cc**.

PMOOP(0396A)-2.7

Second try: let's just declare them

- What if we **don't include myfuncs.cc**? Then the compiler don't know about the functions *f()* and *g()*.
- Solution: we add the declaration of the functions in *main.cc*.

```
// main.cc (user source)
#include <iostream>
using namespace std;

int g();
float f(int, int);

int main() {
    cout << f(g(), g()) << endl;
}

// myfuncs.cc (library source)
#include <cmath>
int i = 2; // Used only by g()
int g() {
    return ++i;
}

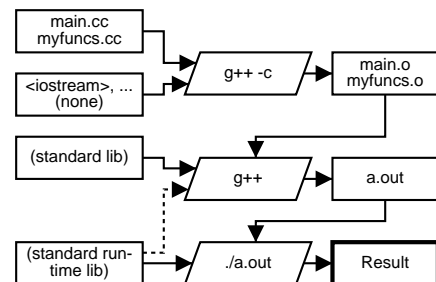
float f(int a, int b) {
    float sum = 0;
    for (int i = 1; i < a; ++i)
        sum += std::sqrt(i);
    return sum;
}
```

Compiling: `g++ -c main.cc`, `g++ -c myfuncs.cc`, `g++ main.o myfuncs.o`

PMOOP(0396A)-2.8

The process

Let's compare the process with what we have said.



We need more compilations, but the amount of code is the about same. Except that we have some more prototypes to compile. This is the overheads.

PMOOP(0396A)-2.9

Insiders' view

What is the magic that lets the object files combine?

- In *main.cc*, the functions *f()* and *g()* are declared, but not defined. So are `cout`, the `<<` operator, etc.
- They are said to be **external references** of the function. The object file *main.o* contains only the **object main**. External references must be fixed before a program is produced.
- When we compile *myfuncs.cc*, the functions *f()* and *g()*, and the variable *i*, are defined. By default, all functions and global variables are **exported**.
- So the object file `myfuncs.o` contains three objects: *i*, *f* and *g*. But it **contains no main**, so it alone cannot be compiled to a program.
- When we link, all the external references are **resolved** using exported object in the specified object files, or object files in the library.
- If multiple object files export the same object, linking fails.

PMOOP(0396A)-2.10

Comments of the current settings

Goods:

- If we only change the implementation of *f* or *g*, we don't need to modify or compile *main.cc*. Likewise, if changing *main* does not necessitate recompilation of *myfuncs.cc*.
How to know which file needs recompilation? Next lecture...

Bad:

- The private variable *i* is exported, which can clash with other symbols. Recall that one symbols can't be exported twice.
- The declaration of *f* and *g* needs to be written in every source file using them.
- How the hell can the writer of *main.cc* (i.e., user program) knows what functions are exported in *myfuncs.cc*?
Especially if the *myfuncs.cc* is hidden and only *myfuncs.o* is provided.

PMOOP(0396A)-2.11

Header files

To solve the problem of having to rewrite declaration every time, and to have a place for the library writer to tell what is exported, we introduce a **header file** to be included by both *main.cc* and *myfuncs.cc*. E.g.,

```
// myfuncs.hh
#ifndef MYFUNCS_HH
#define MYFUNCS_HH
int g();
float f(int, int);
// extern int i; // uncomment this if we want to export i
#endif
```

Now we can add `#include "myfuncs.hh"` rather than write out all declarations both *main.cc* and *myfuncs.cc*.

Note the `#ifndef` directives. It makes sure that **the same header will not be included twice** for one source file.

Why a file can be included twice? Consider what will happen if two header files use the same header file!

PMOOP(0396A)-2.12

Preventing export

To specify that `i` should not be exported, one can either use a **nameless namespace**, or use **static**.

static is deprecated in C++, but still in common use. It's the only choice for C.

```
// myfuncs.cc (nameless namespace) // myfuncs.cc (static)
#include "myfuncs.hh"               #include "myfuncs.hh"
#include <cmath>                     #include <cmath>
namespace {                          static int i = 2;
    int i = 2;                       int g() {
}                                      return ++i;
int g() {                              }
    return ++i;                       float f(int a, int b) {
}                                      float sum = 0;
float f(int a, int b) {               for (int i = 1; i < a; ++i)
    float sum = 0;                    sum += std::sqrt(i);
    for (int i = 1; i < a; ++i)       return sum;
        sum += std::sqrt(i);
    return sum;                       }
}                                      }
```

PMOOP(0396A)-2.13

What header file should contain

If something in the header file change, the user program must be recompiled. This is called **dependencies** of header files.

In general we want to **minimize** such dependencies. On the other hand, the header file must contain certain things. Without them, the compiler cannot generate the code for the source file.

- Declaration of all functions (i.e., prototype) and global variables (i.e., extern variables) that user programs may use.
- Definition of types (i.e., **struct**'s, **typedef**'s, **enum**'s, etc) needed in the interface (e.g., as parameters of functions).
If you need a type only in implementation, don't make it into the header file.
- Definition of all constants that may be used by the source files.
- Definition of all inline functions in the library that user programs can call. (What are inline functions?)

PMOOP(0396A)-2.14

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 we write `++n` in `f()`, it is much more efficient.

But then it would be difficult to modify it, e.g., to change all `++n` to `n+=2`.

PMOOP(0396A)-2.15

How to inline

You can suggest the compiler to use inlining for a function, by prefixing the function definition by **inline**:

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

It is **only a suggestion** to the compiler. The compiler has the final decision whether to inline or not.

For g++, the function is **inlined** only if you asks the compiler to "**optimize**", with the `-O` (or `-O2`) flag. E.g., `g++ -Wall -O myprog.cc`.

This is because g++ tries to make debugging easy without `-O`. Rearranged code would make debugging more difficult.

If a header file contains a function that needs to be inlined, its **definition** (not just prototype) **must appear in the header**.

Otherwise, how the compiler can inline the function?

PMOOP(0396A)-2.16

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 -c -O myprog.cc`, run it like `g++ -Wall -S -O myprog.cc`. The compiler will generate assembly code instead of object file.
Do this only to improve your understanding of the compiler. Don't get too distracted by the assembly!
- A file with the extension `.s` will be created, and you can read the generated code there.
- If you want even more internal details, start the program in the debugger and use the 'disassemble' command (e.g., "disassemble main" to disassemble the code of the main function).
Do this even less!

PMOOP(0396A)-2.17

The cost of inline

- 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 code size large**.
- If this is large enough, it can even make the program slow.
Your program used up fast memory (cache) and starts using slow memory.
- This is not a problem in our previous example, since `incr()` is short.
Actually, it is shorter than the original function calling!

General guidelines: small code size overhead if the function is **short**, or if the function is **called only a few times** throughout the whole program.

At higher optimization level (e.g., `-O3` in `g++`), the compiler will automatically inline for small functions.

PMOOP(0396A)-2.18

Multiple definition: what must not be in headers

Anything that creates an exported object in the object file must not reside in a header file.

Otherwise, when multiple source file include it, each object file will have such an exported object, and the linking will fail due to name clash.

In other words, the header file cannot contain:

- A global variable, except those specified as **const**, **extern** or **static**.
- A definition of function, except an **inline** or **static** function.

Since a header file is for **communication** between the user and library source, most of the time you won't need **static** variables or functions within the header either.

Later in this course we will see that **template function** is an exception to this rule.

PMOOP(0396A)-2.19

#include in headers

- **Question:** If a library uses another library, should I `#include` it in the header?
- **Answer:** it depends on whether we use it in the header.
- If we use the other library in the header, we should `#include` the library from the header. E.g.,

```
#include <string> // Needed!
std::string reverse(std::string);
```

- **Rationale:** Otherwise, the user program must remember to `#include` the header themselves, which will get very troublesome when the number of headers to include becomes large.
Don't worry about including twice: it should be handled by `#ifdef` already.

PMOOP(0396A)-2.20

#include in headers (cont'd)

- If we need the only in the implementation, we should `#include` the library only from the library source. E.g.,

```
// header file           // source file
int reverse(int);       #include <string>
                        int reverse(int x) {
                            std::string s; ...
                        }
```

- **Rationale:** the decision to use the library is a purely implementation issue. Not declaring it in the header allow us to **modify the library source** to use another library, without affecting user programs.
- Using `#include` only in implementation also reduces the problem of header dependencies, so that less programs will need to be recompiled if the included file is modified.

PMOOP(0396A)-2.21

Reducing header dependencies

- Many times we want to **avoid header dependencies** like this.
This is especially important if it is a header we write, and is usually modified.
- One technique is to avoid "using" the type, but just "referencing" the type in the function declaration.
- The idea is simple: we don't need exact knowledge of a type if we **just use a pointer or reference of it**. It is because the size of pointer is fixed.
- We just need to let the compiler know that it is a type. Example:

```
// Don't need #include even if the following appears in our header
class mystring;
mystring *reverse(mystring *);
```

- This is useful (only) if `mystring` is a blackbox in our library.

PMOOP(0396A)-2.22

Advices

- Break up a program once it becomes difficult to manage, either because of large file size, or complicated program structure.
- Break up your program into source files in such a way that each file is minimally independent on other parts.
This might not be possible, but at least this must be a target.
- Inline small functions to speed up the program. To make it possible, write the inlined function in the header, not the source file.
- Strike to understand the basics well, but think in terms of larger concepts. More about it when we introduce OOP.

Exercise

- Find one of your previously written program and separate it into two or more source files. Make sure it still compiles correctly.

PMOOP(0396A)-2.23