VE280 2022FA RC2

L3: Developing Programs

Compilation Process

Compilation process in Linux contains three parts:

- **Preprocessing**: The codes with `#` starts will be implemented.
  - such as: `#define`, `#include`, `#ifdef`
- **Compiler**: Compiles the `.c`/`.cpp` file into object code.
  - The `.c`/`.cpp` files will be compiled to `.o`.
- **Linker**: Links object files into an executable.
  - `.o` files will be linked to an executable file.

Use g++ to Compile Multiple Resources

Three files `class.cpp`, `function.cpp` and `main.cpp` in your directory.

The simplest way to compile them is:\
`g++ -o [name] class.cpp function.cpp main.cpp`,
where `[name]` can be replaced by any name you want.

The complete compile process should be:

`g++ -c class.cpp` ⇒ Compile `class.cpp` to `class.o`

`g++ -c function.cpp` ⇒ Compile `function.cpp` to `function.o`

`g++ -c main.cpp` ⇒ Compile `main.cpp` to `main.o`

`g++ -o [name] class.o function.o main.o` ⇒ Link `.o` files to an executable file named `[name]`.

Remark: The **preprocessing** part is implemented automatically by `g++`.

Question: Files existing in current directory?

Comment: The **benefit** of dividing the compile process apart is that if the project is very large and only a small fraction of the codes are changed, we do not have to compile them again. We only have to recompile the files which are not change and will save a lot of time and resources.

Header File and Header Guard

- **Header file**: used to contain the **function** or **class** declarations.
  Remark: You **don't** need to add the header files in the compiling commands. This is because the header files all already **included** in the preprocessing part by `#include`.

How to solve? :When we develop a large project where some header files are included for many times in many files.

For example, if `#include "class.h"` in both `main.cpp` and `function.cpp`, the header file `class.h` is included twice.

This may cause multiple definitions of the classes or functions defined in the header file, which will lead to tough problems.

- **Header guard**: used to avoid the above situation.

`class.h`:

```
#ifndef CLASS_H
#define CLASS_H

CODE BODY...

#endif
```

Remark: `#ifndef VAR` is a conditional diretive. Like the conditional statement in c++, the directive will test whether `VAR` is defined in the environment. If not, the body code will be implemented until the `#endif`. Different from the conditional statement, `#ifndef` and `#ifdef` are always implemented in the preprocessing part.

Question: How to name `VAR` for different header files?
Comment:
For the **first** time when `class.h` is included, the `#ifndef CLASS_H` will return true and the environmental variable `CLASS_H` will be defined. Then the body codes will be implemented until the `#endif`.

For the **second** time when the `class.h` is included, since the variable `CLASS_H` already exists in the environment, `#ifndef CLASS_H` will return false and the body codes will not be implemented twice.

You should **always** write the header guard when you write your own header files.

Makefile

Makefile: used to write all the commands during the compile process together in a file.

`Makefile`:

```
main: main.o class.o function.o
    g++ -o main main.o class.o function.o

class.o: class.cpp
    g++ -c class.cpp

main.o: main.cpp
    g++ -c main.cpp

function.o: function.cpp
    g++ -c function.cpp

clean:
    rm -f main *.o
```

How the makefile constructed:

- Use `:` to link the demand file and dependent files.
- Use a `<tab>` start command to create the demand file from the dependent files.
- Always switch the line between two demands.
- You can add environmental variables in front of a makefile (optional).

How to use a makefile:

- Type `make` to implement the first demand of makefile.
- Type `make [demand name]` to implement a specific demand

L4 Review of C++ Basics

Basic Concepts

- Built-in data types:
  `int`, `double`, `float` , `char`, `string`.

Question: How many memories does an `int` variable take? How many doese a `char`?

- Input and output by "stream":
  `cout<<"hello world"<<endl`, `cin>>[variables]`
- Operators:
  - Arithmetic: `+`, `-`, `*`, `/`
  - Comparison: `>=`, `==`
  - `x++` or `++x`
  - **Flow**: `>>`, `<<`
- Branch:
  - if/else
  - switch/case
- Loop:
  - while
  - for

lvalue and rvalue

- ***lvalue***: An expression which may appear as **either the left-hand or right-hand side** of an assignment.
- ***rvalue***: An expression which may appear on the **right- but not left-hand** side of an assignment
  - Common lvalues: local variables, return type of "++x", *ptr, ptr[index].
  - Common rvalues: constant, (x+y), return type of "x++" .
  - ***Question***: What is the result of `x`?

```
x = 3;
++x = x;
//x++ = x;
```

Function declaration and definition

- **Declaration**: should appear before the function is called.

Syntax:

```
Return_Type Function_Name(Parameter_List);
//comment
```

- **Definition**: can appear after the function is called.

Syntax:

```
Return_Type Function_Name(Parameter_List)
{
    //function body
}
//comment
```

Reference

- ***Reference***: an important feature of c++.

We can define a variable as a **reference** of an **existing** variable. For example:

```
int a = 1;
int &b = a;
```

Comment: Reference is just like the pointer, which means if we change the value of `b`, the value of `a` will also be changed.

Question: Are the following codes correct? What are the values of `a`, `b` and `ref`.

```
int a = 3,b = 1;
int &ref = a;
&ref = b;
```

```
int &ref = 3;
```

```
int a = 3,b = 1;
int &ref = a;
ref = b;
```

pass the value by reference to a function, like:

```
void f(int &a){
    a*=2;
}
```

If we call the function `f(b)`, the function will define `a` as the reference of `b`. If `a` is changed in the function, the value of `b` will also be changed. You should notice that iF `b` is the name of an array and `f` is written as:

```
void f(int a[])
```

The total array is passed by reference to the function and will be changed by the function.

Pointers

- Some functions of pointer can be replaced by reference.
- Still very important in the dynamic memory allocation.

Structs

- A set of variables.
- What is the total memory of a stuct variable?
- How to declare and define a struct? How to create a struct variable?
- How to access a struct pointer's member attributes?

Exercises

Write a struct Complex in file `complex.h`, which contains two `int` called `real` and `imag`.

Write 2 functions.
`Complex complexAdd(Complex a, Complex b)`, which return the addition of two complex numbers.
`void complexIncre(Complex &a)`, which add the real and imaginary part of `a` by 1.

You should write the function declaration and definition apart, in header file `function.h` and cpp file `function.cpp` respectivley. Remember to write header guard.

Then, write `main.cpp` which can scan the four integers by user and form 2 complex variables `a` and `b`. Then, output `complexAdd(a, b)` and `complexIncre(a)`. For example, the user input "1 2 3 4", the output should be "4+6i 2+3i".

Then, write a makefile and obtain the executable file.