

C++ Programming - Basic

Author: Peter Tse ([mcreng](#))

This section is mainly intended for **non**-software members during the internal competition period.

C++ Code Structure

Welcome to C++. C++ is somewhat similar to C, and its syntax partly adapts from C. Here in this note, some C-style syntax would be introduced. To begin, we shall look at a script of C++ codes. Note that all C++ files are stored in `.cpp` files.

```
1  #include <iostream>
2
3  int main(){
4      // this is a comment
5      /*
6          This
7          is
8          a
9          multiline
10         comment
11     */
12     std::cout << "Hello World" << std::endl;
13     return 0;
14 }
```

This is a basic structure of C++ codes. Here is the dissection:

- `#include <iostream>` is to include the library `iostream`, which is provided by C++ compiler itself.
- `int main(){}` is the program entry point. It should always ends with `return 0` because this specifies to the compiler that the program runs without errors.
- The included library `iostream` is used to print the text `Hello World` in line `std::cout << "..." << std::endl;`. Replacing the text inside `"..."` can make it print anything else.

Variable Type

As seen from the example above, in C++ we have variable types such as `int`. Here is a table of variable types that you might find useful.

Type	Descriptions
<code>int</code>	integer, defaulted to be 32-bit signed integer
<code>bool</code>	boolean, can be either <code>true</code> or <code>false</code>
<code>char</code>	character, can store a single character such as <code>'a'</code>
<code>float</code>	float, single-precision floating point value
<code>double</code>	double, double-precision floating point value
<code>void</code>	void, an absence of type

The following types are also available if `<cstdint>` is included.

Type	Descriptions
<code>uint8_t</code>	unsigned 8-bit integer
<code>uint16_t</code>	unsigned 16-bit integer
<code>uint32_t</code>	unsigned 32-bit integer
<code>int8_t</code>	signed 8-bit integer
<code>int16_t</code>	signed 16-bit integer
<code>int32_t</code>	signed 32-bit integer

The type `string` is also available if `<string>` is included.

In C++, we can declare a variable in the following way. Note that the symbol `=` means 'assignment' in the following context.

```

1  #include <cstdint>
2  #include <string>
3
4  int main(){
5      int a; // declared an integer with name a
6      uint8_t b = 0; // declared an unsigned 8-bit integer with name b and value 0
7      bool c = true; // declared a boolean with name c and value true
8      char d = 'd'; // declared a character with name d and character d, note the single
        quotation
9      float e = 0.0; // declared a float with name e and value 0.0, note the decimal point
10     double f = 0.0; // declared a double with name f and value 0.0
11     std::string g = "hello"; // declared a string with name g and string hello, note the
        double quotation and std::
12     return 0;
13 }
```

It is **highly recommend** that whenever you declare a variable, you should provide it an initial value (*initialization*).

You can name your variables whatever names you want, but bare in minds the following points:

- All variable names should only either begin with an alphabet letter or an underscore (_)
- After the initial character, you can also contain numbers
- Uppercase differs from lowercase letters
- No C++ keywords (such as `int`) can be used

```
1  int main(){
2      int a; // ok
3      int 2; // not ok
4      int a2; // ok
5      int a#2; // not ok, contain special character
6      int _a2; // ok
7      int _A2; // ok, not the same as _a2
8      int float; // not ok
9      return 0;
10 }
```

When you would like to re-assign the variable, you can only call the variable name without the type. Otherwise the compiler would throw an error.

```
1  int main(){
2      int a = 0; // a is 0
3      a = 1; // now a is 1
4      int a = 2; // invalid
5      return 0;
6  }
```

You can also assign a variable with another variable.

```
1  int main(){
2      int a = 0;
3      int b = a; // assigned the value of a to b
4      return 0;
5  }
```

When an assignment of a variable consists of other variables, there might be variable casting, especially if the types of the variables are different.

```

1  int main(){
2      bool b = true;
3      int n = b; // compiler casts a boolean to an integer, n would be 1 because true -> 1;
                     false -> 0
4
5      char c = 'c';
6      n = c; // compiler casts a character to an integer, by using ASCII conversion, n is now 99
7
8      float f = 0.55;
9      n = f; // compiler casts a float to an integer, by truncating the floating point value, n
                     is now 0, same goes with double
10
11     uint8_t p = 6;
12     uint16_t q = p; // compiler casts a 8-bit to a 16-bit, since 16-bit can hold larger
                          values, q is also 6
13
14     q = 678;
15     p = q; // compiler casts a 16-bit to a 8-bit, since 8-bit cannot store a number this
                          large, it overflows and goes back to 166 (678 in binary with only last 8 bits)
16
17     int16_t j = -5;
18     uint16_t k = j; // compiler casts a signed to unsigned, since it cannot store a negative
                          number, it underflows and goes to 65531 (-5 in binary's two's compliment and converts to
                          unsigned)
19
20     return 0;
21 }

```

The type conversion above is an *implicit* casting, i.e. the compiler itself figures out the converted type. Usually, instead, we would prefer a safer approach, that we specify the converted type in our codes.

```

1  int main(){
2      float f = 2.5;
3      int n = (int)f; // C type conversion, n is now 2
4      n = float(f); // C++ type conversion, n is now 2
5
6      return 0;
7  }

```

It is **recommended** to use C++ type conversion, as you will see the reason in an upcoming section.

You may rename certain type for the sake of ease of reading. Taking examples from our basic assignment:

```

1  typedef uint8_t Byte; // typedef <known type> <new name>

```

This can help us clarify between the actual `Byte` that is meant to be used as binary numbers and `uint8_t` as a small number count.

Constants

Apart from variables (things that we can change), we can also specify things that cannot be changed (i.e. constants). There are two ways to define a constant.

1. Using `#define`, **not recommended**

```
1 #define CONSTANT_1 10
2
3 int main(){
4     int a = CONSTANT_1; // a is now 10
5     CONSTANT_1 = 20; // illegal
6     return 0;
7 }
```

2. Using `const`, **recommended**

```
1 int main(){
2     const int a = 10;
3     a = 20; // illegal
4     return 0;
5 }
```

Sometimes using `const` to specify certain items may prevent changing unchangeable values, making it safer.

Operators

For different variables and constants, there are several operators available to be used. These include:

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators
- Assignment operators

Arithmetic Operators

Here is the list of arithmetic operators and the sample code.

Arithmetic operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement

```

1  int main(){
2      int a = 10, b = 20; // note the use of comma here
3      int c = a + b; // 30
4      c = a - b; // -10
5      c = a * b; // 200
6      c = a / b; // 0, note that (int)/(int) -> (int)
7      float d = 2.0;
8      c = d / a; // 0.2, note that if one of the operands contain float, it returns float
9      /*
10         Remember if you wish to do division, always write
11         int a = 2 / 5.0;
12         instead of
13         int a = 2 / 5;
14         Otherwise, you would have repeated our team leader's mistake!!
15     */
16     c = a % b; // 10. a % b means finding the remainder of a div b
17     a++; // a = 11
18     ++a; // a = 12
19     a--; // a = 11
20     --a; // a = 10
21     // Here demonstrates the difference between prefix and postfix increment/decrement
22     c = a++; // c = 10, a = 10 -> 11; c keeps the value of a, then a increments
23     c = ++a; // a = 11 -> 12, c = 12; a increments, then c keeps the value of a
24     // the same goes with decrement
25     return 0;
26 }
```

Relational Operators

Here is the list of relational operators, note that they all return the boolean type.

Relational operator	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>></code>	Bigger than
<code><</code>	Smaller than
<code>>=</code>	Bigger than or equal to
<code><=</code>	Smaller than or equal to

```

1  int main(){
2      bool a;
3      a = (1 == 2); // false
4      a = (1 != 2); // true
5      a = (1 > 2); // false
6      a = (1 < 2); // true
7      a = (1 >= 2); // false
8      a = (1 <= 2); // true
9      return 0;
10 }
```

Logical Operators

Here is the list of logical operators, note that they all also return the boolean type.

Logical operator	Meaning
<code>&&</code>	and
<code> </code>	or
<code>!</code>	not

```

1  int main(){
2      bool _true = true;
3      bool _false = false;
4      bool result;
5      // and operator
6      result = _true && _true; // true
7      result = _true && _false; // false
8      result = _false && _true; // false
9      result = _false && _false; // false
10     // or operator
11     result = _true || _true; // true
12     result = _true || _false; // true
13     result = _false || _true; // true
14     result = _false || _false; // false
15     // not operator
16     result = !_true; // false
17     result = !_false; // true
18     return 0;
19 }

```

Bitwise Operators

Bitwise operators are used to deal with bit by bit arithmetic, really useful when interacting with different electronic modules through low level library.

Bitwise operator	Meaning
&	Bitwise and
	Bitwise or
~	Bitwise not
^	Bitwise xor
<<	Binary left shift
>>	Binary right shift


```

1  int main(){
2      int a = 0b1100; // this is a way to represent binary numbers in C++
3      // side note: you can also use something like 0xFF to define hexadecimal numbers
4      int b = 0b0110;
5      int c;
6      // bitwise and
7      c = a & b;
8      // a = 1100
9      // b = 0110
10     // c = 0100 (bit by bit and)
11
12     // bitwise or
13     c = a | b;
14     // a = 1100
15     // b = 0110
16     // c = 1110 (bit by bit or)
17
18     // bitwise not
19     c = ~a;
20     // a = 1100
21     // c = 0011 (bit by bit not)
22
23     // bitwise xor
24     c = a ^ b;
25     // a = 1100
26     // b = 0110
27     // c = 1010 (bit by bit xor, only one 1 and one 0 returns 1, otherwise 0)
28
29     // left shift, a >> n means a / (2^n)
30     c = a >> 2;
31     // a = 1100
32     // c = 11
33
34     // right shift, a << n means a * (2^n)
35     c = a << 2;
36     // a = 1100
37     // c = 110000
38
39     return 0;
40 }

```

Assignment Operators

Assignment operators are just the combination of assignment `=` and all the above operators.

Assignment operator	Meaning
<code>+=</code>	<code>a += b</code> means <code>a = a + b</code>
<code>-=</code>	<code>a -= b</code> means <code>a = a - b</code>
<code>*=</code>	<code>a *= b</code> means <code>a = a * b</code>
<code>/=</code>	<code>a /= b</code> means <code>a = a / b</code>
<code>%=</code>	<code>a %= b</code> means <code>a = a % b</code>
<code><<=</code>	<code>a <<= b</code> means <code>a = a << b</code>
<code>>>=</code>	<code>a >>= b</code> means <code>a = a >> b</code>
<code>&=</code>	<code>a &= b</code> means <code>a = a & b</code>
<code>^=</code>	<code>a ^= b</code> means <code>a = a ^ b</code>
<code> =</code>	<code>a = b</code> means <code>a = a b</code>

Casting revisited

In the previous section, we recommended the C++ style conversion. This is preferred due to the appearance of operators.

```

1  int main(){
2      char c = (char)23 + 5; // may cause confusion
3      char d = char(23 + 5); // seems nicer
4      float f = float(2) / 5; // however for floating point division, you still need to do this
5      float f = float(2/5); // since this would calculate 2/5 = 0, then casts 0 to 0.0
6      return 0;
7  }
```

Basic I/O

To test our programs here, we mostly would use the user input/output interfaces. Here we would only cover the output interface since input interface has a less usefulness when it comes to debugging our programs. In SmartCar, we do not use C++ I/O anyways, we normally use LCD/LED for outputs and buttons/joystick for inputs.

You have already seen the usage of C++ output interface in the first section, the `std::cout` one.

```

1  #include <iostream> // remember to include this to use std::cout
2
3  int main(){
4      std::cout << "Hello World" << std::endl; // it can output any string
5      int b = 5;
6      std::cout << b << std::endl; // or any variables with any type
7      float c = 0.2;
8      std::cout << b << " " << c << std::endl; // or multiple things at once
9      std::cout << b; // or without std::endl, which is to specify an end line
10     std::cout << " " << c << std::endl; // this is just the same as std::cout << b << " " << c
        << std::endl
11     return 0;
12 }

```

Scopes

Have you noticed the curly brackets `{ }` in our `main()` function? This is to define a *scope*, which you think of as a closure of declared variables and constants. In fact, you can define a variable outside `main()` and you would be still able to access it. It is called a *global variable*.

```

1  int outside = 10;
2  int main(){
3      int inside = outside; // valid
4      return 0;
5  }

```

You can even declare more scopes inside `main()` to separate different parts of your program.

```

1  int outside_main = 10;
2  int main(){
3      int inside_main = outside_main; // valid
4      {
5          int inside_main_scope = inside_main; // valid
6          inside_main_scope = outside_main; // valid
7      }
8      inside_main = inside_main_scope; // invalid, closure of scope!
9      return 0;
10 }

```

In fact, variable names can repeat in different scope.

```

1  #include <iostream>
2  int main(){
3      int n = 10;
4      {
5          int n = 5;
6          std::cout << n << std::endl; // prints 5
7      }
8      std::cout << n << std::endl; // prints 10
9      return 0;
10 }

```

But it refers to outside scope if the variable is not declared inside the scope.

```

1  #include <iostream>
2  int main(){
3      int n = 10;
4      {
5          n = 5;
6          std::cout << n << std::endl; // prints 5
7      }
8      std::cout << n << std::endl; // prints 5
9      return 0;
10 }

```

Controls

In C++, there are several control blocks that you can use to simplify your code. Note the use of `{ }` (scopes).

if-then-else

As the name specified, this can check whether certain conditions are met.

```

1  #include <iostream>
2  int main(){
3      int n = 5;
4      if (n > 7){
5          std::cout << "1" << std::endl;
6      } else if (n > 6){
7          std::cout << "2" << std::endl;
8      } else {
9          std::cout << "3" << std::endl;
10     } // expected output: 3
11     return 0;
12 }

```

If the code inside each scope contains only one sentence, you can omit the scope as it is not useful in this scenario.

```

1 #include <iostream>
2 int main(){
3     int n = 5;
4     if (n > 7) std::cout << "1" << std::endl;
5     else if (n > 6) std::cout << "2" << std::endl;
6     else std::cout << "3" << std::endl;
7     //expected output: 3
8     return 0;
9 }

```

There is a shorthand for `if` if you are using it to assign different values to a variable.

```

1 int main(){
2     bool _certain_req = true;
3     int n = _certain_req ? 2 : 5; // (condition ? true : false), n is now 2
4     int m = !_certain_req ? 2 : 5; // m is now 5
5     return 0;
6 }

```

switch-case

Switch-case is a convenient way to write out a bunch of if-then-else if necessary. Note the **lack** of scope in each `case`.

```

1 #include <iostream>
2 int main(){
3     int a = 2;
4     switch(a){
5         case 1:
6             std::cout << "a is 1" << std::endl;
7             break; // since switch-case is adapted from assembly, without 'break' the code would
                  // just continue to run through other cases, so it is necessary if you do not wish it to
                  // execute the codes in other cases
8         case 2:
9             std::cout << "a is 2" << std::endl;
10            break;
11        case 3:
12            std::cout << "a is 3" << std::endl;
13            break;
14        default: // other cases
15            // not handled
16            break;
17    } // expected output: "a is 2"
18    return 0;
19 }

```

If you do not include `break`:

```

1  #include <iostream>
2  int main(){
3      int a = 2;
4      switch(a){
5          case 1:
6              std::cout << "a is 1" << std::endl;
7          case 2:
8              std::cout << "a is 2" << std::endl;
9          case 3:
10             std::cout << "a is 3" << std::endl;
11     }
12     // expected output: "a is 2"
13                         "a is 3"
14     return 0;
15 }

```

You should add the curly brackets `{ }` if you wish to have separate scopes in each case.

```

1  #include <iostream>
2  int main(){
3      int a = 2;
4      switch(a){
5          case 1:{
6              std::cout << "a is 1" << std::endl;
7              break;
8          }
9          case 2:{
10             std::cout << "a is 2" << std::endl;
11             break;
12          }
13          case 3:{
14             std::cout << "a is 3" << std::endl;
15             break;
16          }
17     }
18     return 0;
19 }

```

while

The `while` block can repeat certain actions in the code when certain conditions are met.

```

1  int main(){
2      int i = 0;
3      while (i < 10){
4          i++;
5      }
6      // i is 10 when it leaves the while-loop
7      return 0;
8  }

```

Again, if the loop only contains one line of code, the scope can be eliminated.

```
1 int main(){
2     int i = 0;
3     while (i < 10) i++;
4     // i is 10
5     return 0;
6 }
```

do-while

The `do - while` block acts similar to the `while` block, the only difference is that for `while`, if the initial conditions are not met, the loop would not be executed, while for `do - while` block, the loop would be executed once no matter what.

```
1 int main(){
2     bool _false = false;
3     int i = 0;
4     do {
5         i++;
6     } while (_false); // the while condition is never met
7     // yet as it goes here i = 1
8     return 0;
9 }
```

```
1 int main(){
2     bool _false = false;
3     int i = 0;
4     do i++;
5     while (_false);
6     // i is 1
7     return 0;
8 }
```

for

The `for` block is also to repeat certain actions in the code, the original purpose for `for` is to loop the codes for certain amount of times, but as you alter the content, it can behave like a `while` loop.

```

1  #include <iostream>
2  int main(){
3      int i;
4      for (i = 0; i < 10; i++){ // (initial value of i; condition to be met; step of i), ++i and
// output: 0 1 2 3 4 5 6 7 8 9
5          std::cout << i << " ";
6      }
7      // output: 0 1 2 3 4 5 6 7 8 9
8
9      for (i = 0; i < 10; i++) std::cout << i << " "; //one line version
10     // output: 0 1 2 3 4 5 6 7 8 9
11
12     for (i = 10; i >= 0; i--){ //--i and i-- are also equivalent
13         std::cout << i << " ";
14     }
15     // output: 10 9 8 7 6 5 4 3 2 1 0
16
17     for (int j = 0; j < 10; j++){ // you can also declare a variable inside the for-loop scope
18         std::cout << j << " ";
19     }
20     // output: 0 1 2 3 4 5 6 7 8 9
21     j = 1; // invalid, closure of for-loop scope
22
23     i = 7;
24     for (; i < 10; i++){ // initial value left blank
25         std::cout << i << " ";
26     }
27     // output: 7 8 9
28
29     for (i = 7; i < 10; ){ // incremenet left blank
30         std::cout << i++ << " "; // Note that original i is printed out, then i is incremented
31     }
32     // output: 7 8 9
33
34     for ( ; ; ){ // all left blank, causing an eternal loop as the condition are always met
35         std::cout << "Hello";
36     }
37     // output: "HelloHelloHello...." foreverly repeats itself
38
39     return 0;
40 }

```

continue and break

In loops like `while`, `do-while` and `for`, you may use `continue` or `break` to alter the loop cycles.

- `continue` means to skip the current remaining cycle and start the next one immediately
- `break` means to completely terminate the current loop.


```

1  #include <iostream>
2  int main(){
3      for (int i = 0; i < 10; i++){
4          if (i%2) continue; // skips every odd number as i%2 is true when i is odd
5          if (i>6) break; // stops the loop if i > 6
6          std::cout << i << std::endl;
7      }
8      return 0;
9  }
10 // Output:
11 // 0 2 4 6

```

Static Variables

Still remember that for a variable in some certain scope, as the program leaves the scope, the values are gone and as the program enters the scope again, the variable is re-created with its original value? If you wish to keep the value as the program re-enters the scope and also preserves the scope safeness, you may use `static` specifier.

```

1  #include <iostream>
2  int main(){
3      for (int i = 0; i < 10; i++){
4          { // note this scope
5              int count = 0;
6              count++;
7              std::cout << count << " ";
8          }
9      }
10 // output: 1 1 1 1 1 1 1 1 1 1
11 return 0;
12 }

```

```

1  #include <iostream>
2  int main(){
3      for (int i = 0; i < 10; i++){
4          { // note this scope
5              static int count = 0; // 0 would be its initial value; as the program re-enters the
scope, it keeps the previous value rather than resetting it to 0
6              count++;
7              std::cout << count << " ";
8          }
9          // count is undefined outside the scope
10 }
11 // output: 1 2 3 4 5 6 7 8 9 10
12 return 0;
13 }

```

Functions

Since the beginning, we have been dealing with `int main()`. In fact, it is a function and you can create other functions apart from the `main` function. This can help you bundle up some codes for certain specific use, and you can re-use them whenever without any repetition of code.

```
1  int my_func(){ // just like main(), you need a type for the function, the name of the
    function and the () to specify it is a function
2      int i = 2;
3      return i; // you should always return the same type of variable
4  }
5
6  float my_func2(){
7      return 0.2;
8  }
9
10 int main(){
11     int i = my_func(); // my_func() returns 2, and assign it to i; note that the scope of both
    i are different
12     float j = my_func2(); // returns 0.2
13     return 0;
14 }
```

You may also recall that there is a `void` type that we have not talked about. It is mainly used in functions to specify that there is no specific type to be returned.

```
1  #include <iostream>
2  void my_func(){
3      std::cout << "Hello" << std::endl;
4      return; // this can be omitted
5  }
6
7  int main(){
8      my_func(); // prints "Hello"
9      return 0;
10 }
```

Sometimes, the function you created can allow certain inputs (parameters).

```
1  int my_add(int a, int b){ // a and b are the inputs of this function, with both type being
    int
2      return a+b;
3  }
4
5  int main(){
6      std::cout << my_add(2, 5) << std::endl; // prints 7
7      std::cout << my_add( my_add(4, 5) , 7) << std::endl; // prints 16
8      return 0;
9  }
```

It is possible for you to predefine the parameters of the functions.

```

1 int myFunc(int a = 2, b = 7){
2     return a + b;
3 }
4 int main(){
5     int i = myFunc(); // a = 2, b = 7, return 9
6     i = myFunc(1); // a = 1, b = 7, return 8
7     i = myFunc(4, 6) // a = 4, b = 6, return 10
8 }

```

Note that after the parameter with default value, any other parameters following it must have a default value as well.

```

1 int myFunc1(int a = 2, b){ // invalid
2     //...
3 }
4 int myFunc2(int a, b = 2){ // valid
5     //...
6 }

```

Within a function, you can call another function. (*What happens if the function calls itself?*)

```

1 int my_func2(){
2     return 2;
3 }
4 int my_func(){
5     return my_func2();
6 }
7 int main(){
8     int i = my_func(); // return 2
9     return 0;
10 }

```

This demonstrates the effect on a function calling itself.

```

1 int f(int n){
2     if (n == 1) return 1;
3     else return n * f(n - 1);
4 }

```

This is an implementation of factorial ($n!$). Note that for most of the times, recursion is a worse implementation method of certain algorithms comparing with other implementations.

C++ codes run from the top to the bottom, so any functions that you would like to call you be defined first (placed at the top) before you call them.

```
1 int main(){
2     my_func(); // error, there is no my_func() upper than this line
3     return 0;
4 }
5 void my_func(){
6     return;
7 }
```

To avoid these kinds of trouble, it is **recommended** for you to define the function first at the top, and leaves your implementation at the bottom. These are what we call *prototypes*.

```
1 void my_func(); // prototype of my_func()
2
3 int main(){
4     my_func(); // ok
5     return 0;
6 }
7 void my_func(){ // implementation of my_func()
8     return;
9 }
```

Logical Operators revisited

When boolean functions are used with logical operators, one might discover one interesting phenomenon.

```

1  #include <iostream>
2  bool _true(){
3      std::cout << "_true() called." << std::endl;
4      return true;
5  }
6
7  bool _false(){
8      std::cout << "_false() called." << std::endl;
9      return false;
10 }
11
12 int main(){
13     _false() && _true(); // only _false() would be called since it must be false no matter the
                           // result of _true()
14     std::cout << std::endl;
15     _true() && _false(); // both called
16     std::cout << std::endl;
17     _true() || _false(); // only _true() would be called since it must be true no matter the
                           // result of _false()
18     std::cout << std::endl;
19     _false() || _true(); // both called
20     return 0;
21 }
22
23 // Expected output:
24 // _false() called.
25 //
26 // _true() called.
27 // _false() called.
28 //
29 // _true() called.
30 //
31 // _false() called.
32 // _true() called.

```

Header

For most cases, you will be working with multiple C++ files since it is best for you to separate stuff that are meant to have different functionalities. To communicate among different C++ files, you need header files (those with `.h` file type). Inside these header files, there are three major things that you will put it.

- Function prototypes that other files can access
- Type declaration that other files can access (will be mentioned in *Intermediate* tutorial)
- Global variables that other files can access

You can think of the header file as a list of things that it is willing to share with other files.

```

1  // sample header file with name file1.h, the implementation of the following functions would be
   // in file1.cpp
2  int my_func();
3  float my_func2();
4  extern int glob_int; // global variables for other files, need extern specifier

```

```
1 // this is file1.cpp
2 #include "file1.h" // need include self header file for the prototypes
3 int glob_int = 5; // define global variable
4 int my_func(){ // define my_func()
5     return 0;
6 }
7 float my_func2(){ // define my_func2()
8     return 3.14;
9 }
```

```
1 // this is another file, file2.cpp
2 // if wish to use the functions in file1.cpp, you must
3 #include "file1.h" // includes the header file for file1.cpp
4
5 int main(){
6     int i = glob_int; // ok
7     int j = my_func(); // ok
8     float k = my_func2(); // ok
9     return 0;
10 }
```

Note that there should only be one `int main()` function as this is the program entry point. Multiple `int main()` function may cause the compiler/linker to return an error.