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.

```cpp
#include <iostream>

int main(){
  // this is a comment
  /*
     This
     is
     a
     multiline
     comment
  */
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

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

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

Type	Descriptions
`uint8_t`	unsigned 8-bit integer
`uint16_t`	unsigned 16-bit integer
`uint32_t`	unsigned 32-bit integer
`int8_t`	signed 8-bit integer
`int16_t`	signed 16-bit integer
`int32_t`	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.

```cpp
#include <cstdint>
#include <string>

int main(){
  int a; // declared an integer with name d
  uint8_t b = 0; // declared an unsigned 8-bit integer with name b and value 0
  bool c = true; // declared a boolean with name c and value true
  char d = 'd'; // declared a character with name d and character d, note the single quotation
  float e = 0.0; // declared a float with name e and value 0.0, note the decimal point
  double f = 0.0; // declared a double with name f and value 0.0
  std::string g = "hello"; // declared a string with name g and string hello, note the double quotation and std::
  return 0;
}
```

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

```cpp
int main(){
  int a; // ok
  int 2; // not ok
  int a2; // ok
  int a#2; // not ok, contain special character
  int _a2; // ok
  int _A2; // ok, not the same as _a2
  int float; // not ok
  return 0;
}
```

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.

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

You can also assign a variable with another variable.

```
int main(){
  int a = 0;
  int b = a; // assigned the value of a to b
  return 0;
}
```

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

```cpp
int main(){
  bool b = true;
  int n = b; // compiler casts a boolean to an integer, n would be 1 because true -> 1; false -> 0

  char c = 'c';
  n = c; // compiler casts a character to an integer, by using ASCII conversion, n is now 99

  float f = 0.55;
  n = f; // compiler casts a float to an integer, by truncating the floating point value, n is now 0, same goes with double

  uint8_t p = 6;
  uint16_t q = p; // compiler casts a 8-bit to a 16-bit, since 16-bit can hold larger values, q is also 6

  q = 678;
  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)

  int16_t j = -5;
  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)

  return 0;
}
```

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.

```cpp
int main(){
  float f = 2.5;
  int n = (int)f; // C type conversion, n is now 2
  n = float(f); // C++ type conversion, n is now 2

  return 0;
}
```

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:

```c
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**

```c
#define CONSTANT_1 10

int main(){
  int a = CONSTANT_1; // a is now 10
  CONSTANT_1 = 20; // illegal
  return 0;
}
```

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

```c
int main(){
  const int a = 10;
  a = 20; // illegal
  return 0;
}
```

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

```
int main(){
  int a = 10, b = 20; // note the use of comma here
  int c = a + b; // 30
  c = a - b; // -10
  c = a * b; // 200
  c = a / b; // 0, note that (int)/(int) -> (int)
  float d = 2.0;
  c = d / a; // 0.2, note that if one of the operands contain float, it returns float
  /*
     Remember if you wish to do division, always write
     int a = 2 / 5.0;
     instead of
     int a = 2 / 5;
     Otherwise, you would have repeated our team leader's mistake!!
  */
  c = a % b; // 10. a % b means finding the remainder of a div b
  a++; // a = 11
  ++a; // a = 12
  a--; // a = 11
  --a; // a = 10
  // Here demostrates the difference between prefix and postfix increment/decrement
  c = a++; // c = 10, a = 10 -> 11; c keeps the value of a, then a increments
  c = ++a; // a = 11 -> 12, c = 12; a increments, then c keeps the value of a
  // the same goes with decrement
  return 0;
}
```

Relational Operators

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

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

```
int main(){
  bool a;
  a = (1 == 2); // false
  a = (1 != 2); // true
  a = (1 > 2); // false
  a = (1 < 2); // true
  a = (1 >= 2); // false
  a = (1 <= 2); // true
  return 0;
}
```

Logical Operators

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

Logical operator	Meaning
`&&`	and
`||`	or
`!`	not

```
int main(){
  bool _true = true;
  bool _false = false;
  bool result;
  // and operator
  result = _true && _true; // true
  result = _true && _false; // false
  result = _false && _true; // false
  result = _false && _false; // false
  // or operator
  result = _true || _true; // true
  result = _true || _false; // true
  result = _false || _true; // true
  result = _false || _false; // false
  // not operator
  result = !_true; // false
  result = !_false; // true
  return 0;
}
```

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

```cpp
int main(){
  int a = 0b1100; // this is a way to represent binary numbers in C++
  // side note: you can also use something like 0xFF to define hexadecimal numbers
  int b = 0b0110;
  int c;
  // bitwise and
  c = a & b;
  // a = 1100
  // b = 0110
  // c = 0100 (bit by bit and)

  // bitwise or
  c = a | b;
  // a = 1100
  // b = 0110
  // c = 1110 (bit by bit or)

  // bitwise not
  c = ~a;
  // a = 1100
  // c = 0011 (bit by bit not)

  // bitwise xor
  c = a ^ b;
  // a = 1100
  // b = 0110
  // c = 1010 (bit by bit xor, only one 1 and one 0 returns 1, otherwise 0)

  // left shift, a >> n means a / (2^n)
  c = a >> 2;
  // a = 1100
  // c = 11

  // right shift, a << n means a * (2^n)
  c = a << 2;
  // a = 1100
  // c = 110000

  return 0;
}
```

Assignment Operators

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

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

Casting revisited

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

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

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.

```
#include <iostream> // remember to include this to use std::cout

int main(){
  std::cout << "Hello World" << std::endl; // it can output any string
  int b = 5;
  std::cout << b << std::endl; // or any variables with any type
  float c = 0.2;
  std::cout << b << " " << c << std::endl; // or multiple things at once
  std::cout << b; // or without std::endl, which is to specify an end line
  std::cout << " " << c << std::endl; // this is just the same as std::cout << b << " " << c << std::endl
  return 0;
}
```

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*.

```
int outside = 10;
int main(){
  int inside = outside; // valid
  return 0;
}
```

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

```
int outside_main = 10;
int main(){
  int inside_main = outside_main; // valid
  {
    int inside_main_scope = inside_main; // valid
    inside_main_scope = outside_main; // valid
  }
  inside_main = inside_main_scope; // invalid, closure of scope!
  return 0;
}
```

In fact, variable names can repeat in different scope.

```cpp
#include <iostream>
int main(){
  int n = 10;
  {
    int n = 5;
    std::cout << n << std::endl; // prints 5
  }
  std::cout << n << std::endl; // prints 10
  return 0;
}
```

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

```cpp
#include <iostream>
int main(){
  int n = 10;
  {
    n = 5;
    std::cout << n << std::endl; // prints 5
  }
  std::cout << n << std::endl; // prints 5
  return 0;
}
```

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.

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

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

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

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

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

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`.

```cpp
#include <iostream>
int main(){
  int a = 2;
  switch(a){
    case 1:
      std::cout << "a is 1" << std::endl;
      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
    case 2:
      std::cout << "a is 2" << std::endl;
      break;
    case 3:
      std::cout << "a is 3" << std::endl;
      break;
  } // expected output: "a is 2"
  return 0;
}
```

If you do not include `break`:

```cpp
#include <iostream>
int main(){
  int a = 2;
  switch(a){
    case 1:
      std::cout << "a is 1" << std::endl;
    case 2:
      std::cout << "a is 2" << std::endl;
    case 3:
      std::cout << "a is 3" << std::endl;
  }
  // expected output: "a is 2"
                      "a is 3"
  return 0;
}
```

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

```cpp
#include <iostream>
int main(){
  int a = 2;
  switch(a){
    case 1:{
      std::cout << "a is 1" << std::endl;
      break;
    }
    case 2:{
      std::cout << "a is 2" << std::endl;
      break;
    }
    case 3:{
      std::cout << "a is 3" << std::endl;
      break;
    }
  }
  return 0;
}
```

while

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

```cpp
int main(){
  int i = 0;
  while (i < 10){
    i++;
  }
  // i is 10 when it leaves the while-loop
  return 0;
}
```

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

```cpp
int main(){
  int i = 0;
  while (i < 10) i++;
  // i is 10
  return 0;
}
```

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.

```
int main(){
  bool _false = false;
  int i = 0;
  do {
    i++;
  } while (_false); // the while condition is never met
  // yet as it goes here i = 1
  return 0;
}
```

```
int main(){
  bool _false = false;
  int i = 0;
  do i++;
  while (_false);
  // i is 1
  return 0;
}
```

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.

```
#include <iostream>
int main(){
  int i;
  for (i = 0; i < 10; i++){ // (initial value of i; condition to be met; step of i), ++i and i++ are equivalent
    std::cout << i << " ";
  }
  // output: 0 1 2 3 4 5 6 7 8 9

  for (i = 0; i < 10; i++) std::cout << i << " "; //one line version
  // output: 0 1 2 3 4 5 6 7 8 9

  for (i = 10; i >= 0; i--){ //--i and i-- are also equivalent
    std::cout << i << " ";
  }
  // output: 10 9 8 7 6 5 4 3 2 1 0

  for (int j = 0; j < 10; j++){ // you can also declare a variable inside the for-loop scope
    std::cout << j << " ";
  }
  // output: 0 1 2 3 4 5 6 7 8 9
  j = 1; // invalid, closure of for-loop scope

  i = 7;
  for (; i < 10; i++){ // initial value left blank
    std::cout << i << " ";
  }
  // output: 7 8 9

  for (i = 7; i < 10; ){ // incremenet left blank
    std::cout << i++ << " "; // Note that original i is printed out, then i is incremented
  }
  // output: 7 8 9

  for ( ; ; ){ // all left blank, causing an eternal loop as the condition are always met
    std::cout << "Hello";
  }
  // output: "HelloHelloHello...." foreverly repeats itself

  return 0;
}
```

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.

```cpp
#include <iostream>
int main(){
  for (int i = 0; i < 10; i++){
    if (i%2) continue; // skips every odd number as i%2 is true when i is odd
    if (i>6) break; // stops the loop if i > 6
    std::cout << i << std::endl;
  }
  return 0;
}
// Output:
// 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.

```cpp
#include <iostream>
int main(){
  for (int i = 0; i < 10; i++){
    { // note this scope
      int count = 0;
      count++;
      std::cout << count << " ";
    }
  }
  // output: 1 1 1 1 1 1 1 1 1 1
  return 0;
}
```

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

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.

```cpp
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
  int i = 2;
  return i; // you should always return the same type of variable
}

float my_func2(){
  return 0.2;
}

int main(){
  int i = my_func(); // my_func() returns 2, and assign it to i; note that the scope of both i are different
  float j = my_func2(); // returns 0.2
  return 0;
}
```

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.

```
#include <iostream>
void my_func(){
  std::cout << "Hello" << std::endl;
  return; // this can be omitted
}

int main(){
  my_func(); // prints "Hello"
  return 0;
}
```

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

```
int my_add(int a, int b){ // a and b are the inputs of this function, with both type being int
  return a+b;
}

int main(){
  std::cout << my_add(2, 5) << std::endl; // prints 7
  std::cout << my_add( my_add(4, 5) , 7) << std::endl; // prints 16
  return 0;
}
```

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

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

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

```
int myFunc1(int a = 2, b){ // invalid
  //...
}
int myFunc2(int a, b = 2){ // valid
  //...
}
```

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

```
int my_func2(){
  return 2;
}
int my_func(){
  return my_func2();
}
int main(){
  int i = my_func(); // return 2
  return 0;
}
```

This demonstrates the effect on a function calling itself.

```
int f(int n){
  if (n == 1) return 1;
  else return n * f(n - 1);
}
```

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.

```
int main(){
  my_func(); // error, there is no my_func() upper than this line
  return 0;
}
void my_func(){
  return;
}
```

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*.

```
void my_func(); // prototype of my_func()

int main(){
  my_func(); // ok
  return 0;
}
void my_func(){ // implementation of my_func()
  return;
}
```

Logical Operators revisited

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

```cpp
#include <iostream>
bool _true(){
  std::cout << "_true() called." << std::endl;
  return true;
}

bool _false(){
  std::cout << "_false() called." << std::endl;
  return false;
}

int main(){
  _false() && _true(); // only _false() would be called since it must be false no matter the result of _true()
  std::cout << std::endl;
  _true() && _false(); // both called
  std::cout << std::endl;
  _true() || _false(); // only _true() would be called since it must be true no matter the result of _false()
  std::cout << std::endl;
  _false() || _true(); // both called
  return 0;
}

// Expected output:
// _false() called.
//
// _true() called.
// _false() called.
//
// _true() called.
//
// _false() called.
// _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.

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

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

```
// this is another file, file2.cpp
// if wish to use the functions in file1.cpp, you must
#include "file1.h" // includes the header file for file1.cpp

int main(){
  int i = glob_int; // ok
  int j = my_func(); // ok
  float k = my_func2(); // ok
  return 0;
}
```

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.