Basics of C++:

Program structure

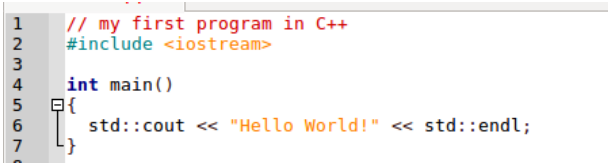
Variables and types

Functions

Statements and flow control

Object Oriented Programming

Program structure



```
1  // my first program in C++
2  #include <iostream>
3
4  int main()
5  {
6     std::cout << "Hello World!" << std::endl;
7  }
```

- Line 1: Comments
- Line 2: Preprocessor directives
- Line 3: Blank lines
- Line 4: main() function declaration
- Lines 5 and 7: braces are used to indicate the beginning and end of function definitions
- Line 6: this line is a C++ statement

Variables and types 1

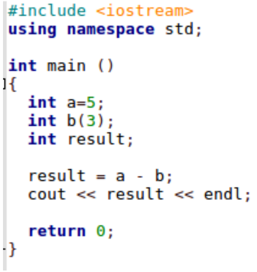
- Variables
  - variables are used when we need storage for any value which will be used by our program
  - we can define variable as a portion of memory to store a value
- Identifiers
  - each variable needs a unique name called identifier
  - a valid identifier is a sequence of one or more letters, digits, or underscores
  - important: C++ language is "case sensitive" (variable x is not the same as X)
- Variable Types
  - **bool**: stores either value true or false
  - **char**: a one-byte character ('A' or '@')
  - **int**: integer number value (7 or 1024)
  - **float** and **double**: represent real values, such as 3.14 or 0.01 (with different levels of precision)

Variables and types 2

- Declaration of variables
  - C++ is a strongly-typed language
  - every variable to be declared with its type before being used
  - example

```
int a;
float number;
int a, b, c;
```

- Initialization of variables



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

int main ()
{
   int a=5;
   int b(3);
   int result;

   result = a - b;
   cout << result << endl;

   return 0;
}
```

Functions

A function is a set of specific statements and which can be called by name from some point of the program. The syntax to define a function is:

type name (parameter1, parameter2, ...)   statements

- **type**: type of the value returned by the function
- **name**: is the function's identifier
- **parameters** (as many as needed): input to the function
- **statements**: the function's body (block of statements surrounded by braces   )
- example function:

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

int add(int a, int b)
{
  int result;
  result=a+b;
  return result;
}

int main()
{
  int x;
  x = add(1,2);
  cout << "The output is: "<< x << endl;
}
```

Functions 2

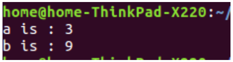
Arguments can be passed by value and by reference:

- **by value**: C++ copies the actual value of an argument into the formal parameter of the function; in this case, changes made to the parameter inside the function have no effect on the argument
- **by reference**: C++ is passing the reference of an argument in the calling function to the corresponding formal parameter; the function can modify the value of the argument
- example

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

void square_val(int a)
{
        a *= a;
}
void square_ref(int& a)
{
        a *= a;
}

int main()
{
  int a(3), b(3);
  square_val(a);
  square_ref(b);
  cout << "a is : "<< a << endl;
  cout << "b is : "<< b << endl;
}
```



```
home@home-ThinkPad-X220:~/
a is : 3
b is : 9
```

Statements and flow control 1

Selection statements: if and else

- the *if* keyword is used to execute a statement or block, if, and only if, the condition is true

```
if (x > 0)
  cout << "x is positive" << endl;
```

- by using *else* keyword we can specify what happens when the condition is not fulfilled

```
if (x > 0)
  cout << "x is positive" << endl;
else
  cout << "x is not positive" << endl;
```

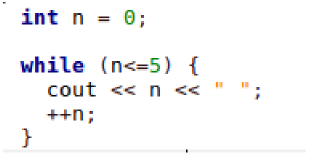
- if + else structures can be nested with the intention of checking a range of values

```
if (x > 0)
  cout << "x is positive" << endl;
else if (x < 0)
  cout << "x is negative" << endl;
else
  cout << "x is 0" << endl;
```

Statements and flow control 2

Iteration statements (loops)

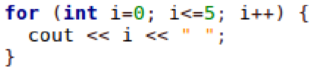
- the *while* loop repeats the statement while expression is true



```
int n = 0;

while (n<=5) {
   cout << n << " ";
   ++n;
}
```

- the *for* loop is designed to iterate a number of times



```
for (int i=0; i<=5; i++) {
   cout << i << " ";
}
```

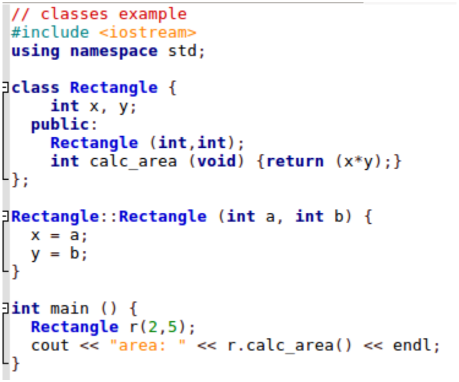
- $i{+}{+}$ increments i after the expression has been evaluated
- $++i$ increments i before the expression is evaluated

Object Oriented Programming Principles

- Abstraction - hides complexity by providing a more abstract picture (hides details at the design level)
- Encapsulation - hiding the implementation details and only exposing necessary methods (hides details at the implementation level)
- Inheritance - allows programmers create new classes that share some of the attributes of existing classes
- Polymorphism - the ability of different objects to respond to the same message in different ways

Object Oriented Programming

- A Class is a user defined data-type which has data members and member functions
- Data members and member functions can be accessed and used by creating an instance of that class
- An Object is an instance of a Class



```
// classes example
#include <iostream>
using namespace std;

class Rectangle {
    int x, y;
  public:
    Rectangle (int,int);
    int calc_area (void) {return (x*y);}
};

Rectangle::Rectangle (int a, int b) {
  x = a;
  y = b;
}

int main () {
  Rectangle r(2,5);
  cout << "area: " << r.calc_area() << endl;
}
```