Tutorial of C++

Input from keyboard and output to screen can be performed through cin >> and cout <<:

```
#include <iostream>
using namespace std;
void main() {
   int a;                  // a is an integer variable
   char s [100];           // s points to a string of max 100 characters
   cout << "This is a sample program." << endl;
   cout << endl;           // Just a line feed (end of line)
   cout << "Type your age : ";
   cin >> a;
   cout << "Type your name: ";
   cin >> s;
   cout << endl;
   cout << "Hello " << s << " you're " << a << " old." << endl;
   cout << "Bye!" << endl;
   return 0;
}
```

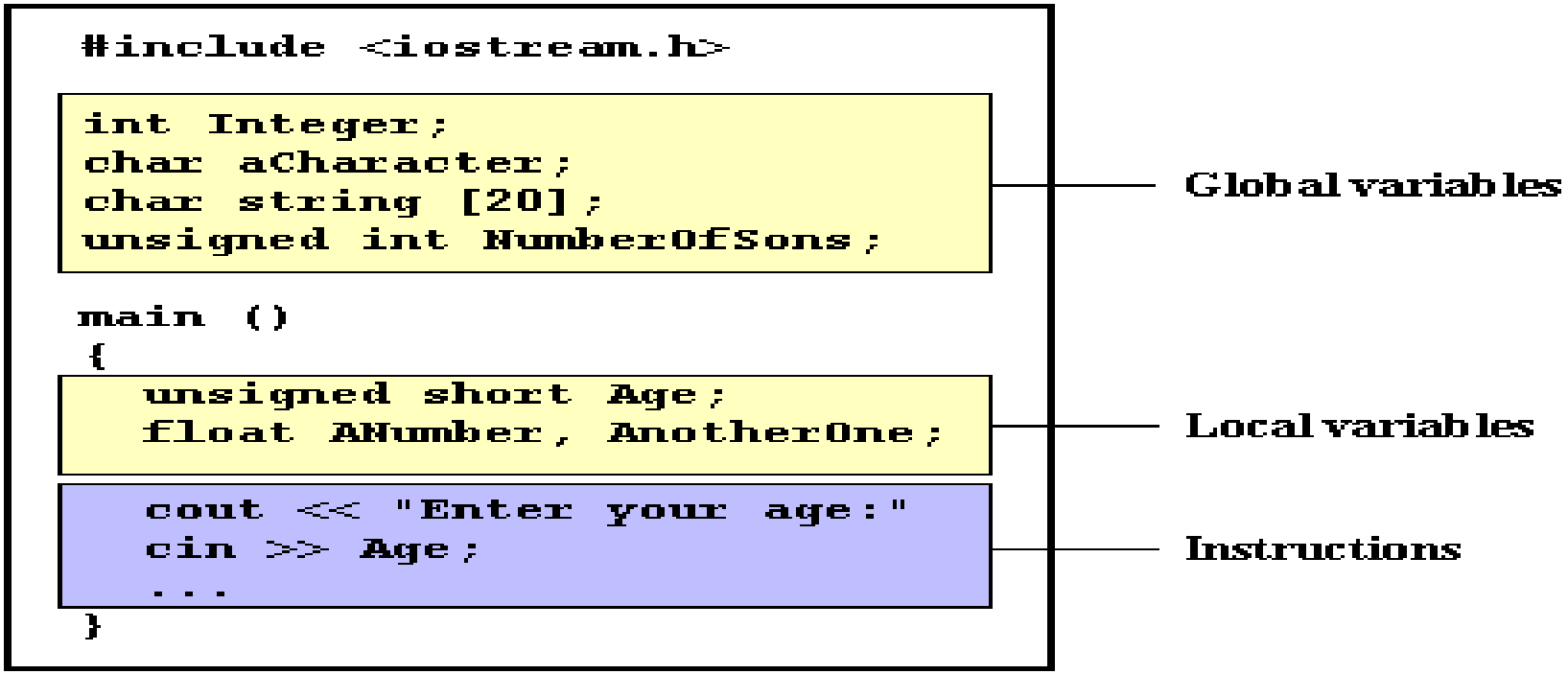
Variables can be declared everywhere inside the code:

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

int main (){
  double a;
  cout << "Hello, this is a test program." << endl;
  cout << "Type parameter a: ";
  cin >> a;

  a = (a + 1) / 2;
  double c;
  c = a * 5 + 1;
  cout << "c contains      : " << c << endl;
  int i, j;
  i = 0;
  j = i + 1;
  cout << "j contains      : " << j << endl;
  return 0;
}
```

Scope of variables

A variable can be either of global or local scope.

- A global variable is a variable declared in the main body of the source code, outside all functions.
- A local variable is one declared within the body of a function or a block.



A global variable can be accessed even if another variable with the same name has been declared inside the function:

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

double a = 128;
int main ()
{
  double a = 256;
  cout << "Local a:  " << a   << endl;
  cout << "Global a: " << ::a << endl;
  return 0;
}
```

References can be used to allow a function to modify a calling variable:

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

void change (double &r, double s){
   r = 100;
   s = 200;
}

int main (){
   double k, m;
   k = 3;
   m = 4;
   change (k, m);
   cout << k << ", " << m << endl;        // Displays 100, 4.
   return 0;
}
```

If you are using pointers:

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

void change (double *r, double s){
   *r = 100;
   s = 200;
}

int main () {
   double k, m;
   k = 3;
   m = 4;
   change (&k, m);
   cout << k << ", " << m << endl;        // Displays 100, 4.
   return 0;
}
```

Namespaces

Allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

```
namespace identifier
{
entities
}
```

where identifier is any valid identifier and entities such as the set of classes, objects and functions that are included within the namespace.

```
namespace myNamespace
{
  int a, b;
}
```

In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::.

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors.

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

namespace first
{
  int var = 5;}

namespace second
{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

```
5
3.1416
```

The keyword using is used to introduce a name from a namespace into the current declarative region.

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

namespace first
{
  int x = 5;
  int y = 10;}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

```
5
2.7183
10
3.1416
```

The keyword using can also be used as a directive to introduce an entire namespace:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;}

int main () {
  using namespace first;
  cout << x << endl;
  cout << y << endl;
  cout << second::x << endl;
  cout << second::y << endl;
  return 0;}
```

```
5
10
3.1416
2.7183
```

Using and using namespace have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope.

If we had the intention to first use the objects of one namespace and then those of another one, we could

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

namespace first{
  int x = 5;}

namespace second{
  double x = 3.1416;}
```

```
int main () {
  {    using namespace first;
    cout << x << endl;  }
  {    using namespace second;
    cout << x << endl;  }
  return 0;
}

5
3.1416
```

Namespace std

All the files in the C++ standard library declare all of its entities within the std namespace. That is why we have generally included the using namespace std; statement in all programs that used any entity defined in iostream.

Method Overload

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

double test (double a, double b){
  return a + b;}

int test (int a, int b){
  return a - b;}

int main (){
  double   m = 7,  n = 4;
  int      k = 5,  p = 3;

  cout << test(m, n) << " , " << test(k, p) << endl;

  return 0;
}
```

The OPERATORS OVERLOAD can be used to define the basic symbolic operators:

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

struct vector{
   double x;
   double y;};

vector operator * (double a, vector b){
   vector r;
   r.x = a * b.x;
   r.y = a * b.y;
   return r;
}

int main (){
   vector k, m;
   k.x =  2;
   k.y = -1;                  // k = vector (2, -1)
   m = 3.1415927 * k;
   cout << "(" << m.x << ", " << m.y << ")" << endl;
   return 0;
}
```

Record

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

struct movies_t {
  string title;
  int year;
} yours;

void printmovie (movies_t movie);

int main () {
    cout << "Enter title: ";
  getline (cin, yours.title);
  cout << "Enter year: ";
  getline (cin, yours.year);
  cout << "My favorite movie is:\n ";
  printmovie (yours);
  return 0;}

void printmovie (movies_t movie){
  cout << movie.title;
  cout << " (" << movie.year << ")\n";}
```

Pointers to Records

Records can be pointed by its own type of pointers:

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

struct movies_t {
  string title;
  int year;
};

int main ()
{
  string mystr;
  movies_t amovie;
  movies_t * pmovie;
  pmovie = &amovie;
  cout << "Enter title: ";
  getline (cin, pmovie->title);
  cout << "Enter year: ";
  getline (cin, pmovie->year);

  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";
  return 0;
}
```

```
pmovie->title
```

is equivalent to:

```
(*pmovie).title
```

Both mean that we are evaluating the member title of the data structure pointed by a pointer called pmovie.

It must be clearly differentiated from:

```
*pmovie.title
```

```
which is equivalent to:  *(pmovie.title)
```

And that would access the value pointed by a pointer member called title.

Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed by a	(*a).b
*a.b	Value pointed by member b of object a	*(a.b)

Nesting structures

```
struct movies_t {
  string title;
  int year;
};

struct friends_t {
  string name;
  string email;
  movies_t favorite_movie;
  } charlie, maria;

friends_t * pfriends = &charlie;
```

After the previous declaration we could use any of the following expressions:

```
charlie.name
maria.favorite_movie.title
charlie.favorite_movie.year
pfriends->favorite_movie.year
```

In standard C a struct just contains data. In C++ a struct definition can also include functions.

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

struct vector{
   double x;
   double y;
   double surface ()   {
      double s;
      s = x * y;
      if (s < 0) s = -s;
      return s;
   }
};

int main (){
   vector a;
   a.x = 3;
   a.y = 4;
   cout << "The surface of a: " << a.surface() << endl;
   return 0;}
```

Class:

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

class vector{
public:
   double x;
   double y;
   double surface ()   {
      double s;
      s = x * y;
      if (s < 0) s = -s;
      return s;   }
};

int main (){
   vector a;
   a.x = 3;
   a.y = 4;
   cout << "The surface of a: " << a.surface() << endl;
   return 0;}
```

Constructor and Destructor

- They are automatically called whenever an instance of a class is created or destroyed (new, delete...).
- If an object is destroyed, for example by leaving its definition scope, the destructor of the corresponding class is invoked.
- If this class is derived from other classes their destructors are also called, leading to a recursive call chain.

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

class person{
public:
  char *name;
  int age;
  person (char *n = "no name", int a = 0)   {
    name = new char [100];
    strcpy (name, n);
    age = a;
    cout << "Instance initialized, 100 bytes allocated" << endl;   }
```

```
~person ()                          // The destructor
  {
    delete name;                    // instead of free!

    cout << "Instance going to be deleted, 100 bytes freed" << endl;
  }
};

int main (){
   cout << "Hello!" << endl << endl;
   person a;
   cout << a.name << ", age " << a.age << endl << endl;
   person b ("John", 21);
   cout << b.name << ", age " << b.age << endl << endl;
   cout << b.name << ", age " << b.age << endl << endl;
  return 0;
}
```

Private variables and Methods:

```
class Point
{
public:
   Point();
   Point(double xval, double yval);
   void move(double dx, double dy);
   double getX() const;
   double getY() const;
private:
   double x;
   double y;
};
```

Class Inheritance:

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

class vector{
public:
  protected double x;
  protected double y;
  vector (double a, double b)   {
    x = a;
    y = b;
  }
  double module()   {
    return sqrt (x*x + y*y);
  }

  double surface()   {
     return x * y;   }
};
```

```
class trivector: public vector   // trivector is derived from vector
{
public:
  double z;
  trivector (double m=0, double n=0, double p=0): vector (m, n)
  {
    z = p;                    // Vector constructor will
  }                           // be called before trivector
                              // constructor, with parameters
                              // m and n.

  double module ()            // define module() for trivector
  {
    return sqrt (x*x + y*y + z*z);
  }

  double volume ()   {
     return this->surface() * z;        // or x * y * z
  }
};
```

Initializing Members

Constructors can initialize their members in two different ways.

A constructor can use the arguments passed to it to initialize member variables in the constructor definition:

```
complx(double r, double i = 0.0) { re = r; im = i; }
```

Or a constructor can have an *initializer list* within the definition but prior to the constructor body:

```
complx(double r, double i = 0) : re(r), im(i) { /* ... */ }
```

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

class B1 {
  int b;
public:
  B1() { cout << "B1::B1()" << endl; };

  B1(int i) : b(i) { cout << "B1::B1(int)" << endl; }
};

class D : public B1 {
  int d1, d2;
public:
  D(int i, int j) : B1(i+1), d1(i)
  { d2 = j;}
};

int main() {
  D obj(1, 2);
}
```

Use of header and implementation files

There are two ways to put your code in header (**.h**) and implementation (**.cpp**) files:

1) Use only **.h** files

2) Add **.cpp** files to project

1) Use only .h files

This option is used by many libraries. Put all definitions in the **.h** files. This has the advantage that they are easy to call: you only need to #include the necessary **.h** files.

```
//Example.h

include <iostream>

struct Example
{
  void sayHello() const
  {
    cout << "Hello" << endl; //Definition in .h file
  }
};
```

```
//Main.cpp

include "Example.h"

int main()
{
  Example myExample;
  myExample.sayHello();
}
```

2) Add .cpp files to project

This option is used in larger non-library code. The header files contain the declarations, the implementation files the definitions. The advantage is that compiling is quickest.

```
//Example.h

struct Example
{
  void sayHello() const;
};
```

```
//Example.cpp

include "Example.h"

include <iostream>

void Example::sayHello() const
{
  std::cout << "Hello" << std::endl;
}
```

```
//Main.cpp

include "Example.h"

int main()
{
  Example myExample;
  myExample.sayHello();
}
```

If you intent to develop a serious C or C++ software, you need to separate the source code in .h header files and .cpp source files.

A header file vector.h:

```
class vector{
public:
   double x;
   double y;
   double surface();
};
```

A source code file vector.cpp:

```
#include "vector.h"
using namespace std;

double vector::surface(){
   double s = 0;
   for (double i = 0; i < x; i++)   {
      s = s + y; }
   return s;
}
```

And a source code file main.cpp:

```
using namespace std;
#include <iostream>
#include "vector.h"

int main (){
   vector k;
   k.x = 4;
   k.y = 5;
   cout << "Surface: " << k.surface() << endl;
   return 0;
}
```

It is possible to declare arrays of objects:

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

class vector{
public:
  double x;
  double y;
  vector (double a = 0, double b = 0)   {
    x = a;
    y = b;   }
  double module ()   {
    return sqrt (x * x + y * y);   }
};

int main (){
  vector s [1000];
  vector t[3] = {vector(4, 5), vector(5, 5), vector(2, 4)};
  s[23] = t[2];
  cout << t[0].module() << endl;
  return 0;}
```

A class' variable can be declared static. Then only one instance of that variable exists, shared by all instances of the class.

It must be initialised outside the class declaration :

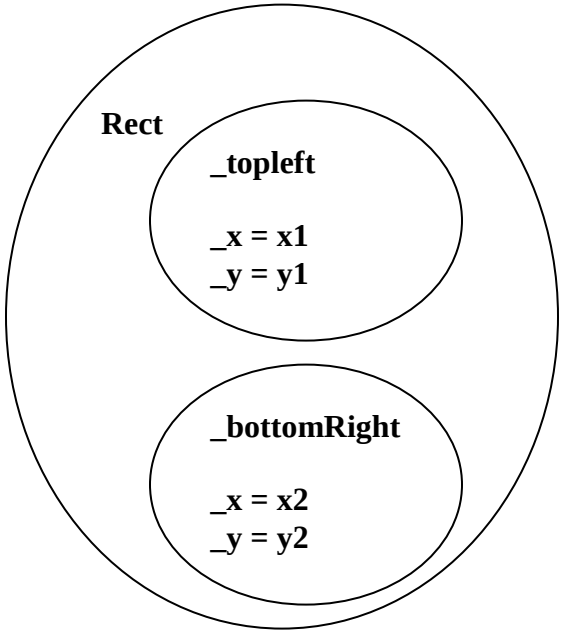
```
using namespace std;
#include <iostream>

class vector{
public:
  double x;
  double y;
  static int count;
  vector (double a = 0, double b = 0)   {
    x = a;
    y = b;
    count++;   }

  ~vector()   {
    count--;
  }
};
```

A class variable can also be constant.

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

class vector{
public:
   double x;
   double y;
   const static double pi = 3.1415927;
   vector (double a = 0, double b = 0)   {
      x = a;
      y = b;   }
   double cilinder_volume ()   {
      return x * x / 4 * pi * y;   }
};

  int main(){
       cout << "The value of pi: " << vector::pi << endl << endl;
       vector k(3, 4);
       cout << "Result: " << k.cilinder_volume() << endl;
      return 0;
}
```

Initializing Member Objects

Classes can contain member objects of class type




```
class Point {
        public:
                Point( int x, int y ) { _x = x; _y = y; }
        private:
                int _x, _y; };

// Declare a rectangle class that contains objects of type Point.
class Rect {
        public: Rect( int x1, int y1, int x2, int y2 );
        private: Point _topleft, _bottomright; };
  // Define the constructor for class Rect. This constructor initializes the objects of type Point.
        Rect::Rect( int x1, int y1, int x2, int y2 ) : _topleft( x1, y1 ), _bottomright( x2, y2 ) { }
    int main() { }
```

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

class IntPair {
public:
  int a;
  int b;
  IntPair(int i, int j) : a(i), b(j) { }
};

class MyClass {
  IntPair nums;
public:
  // Initialize nums object using
// initialization syntax.
  MyClass(int x, int y) : nums(x,y) { }
  int getNumA() {
    return nums.a;
  }
  int getNumB() {
    return nums.b;
  }};
```

```
int main()
{
  MyClass object1(7, 9), object2(5, 2);
  cout << "Values in object1 are " << object1.getNumB() <<
        " and " << object1.getNumA() << endl;
  cout << "Values in object2 are " << object2.getNumB() <<
        " and " << object2.getNumA() << endl;
  return 0;
}
```

Answer the following questions given this declaration of a **rectangle** class. Except in the cases where you are asked to define them, assume member functions have been defined properly.

```
class rectangle
{
  public:
    rectangle();
    void setUpperLeft(double x, double y);  // sets upper left corner coords
    void setLength(double r);               // sets length
    void setWidth(double r);                // sets width
    double getX();                      // returns x coordinate of center
    double getY();                      // returns y coordinate of center
    double getLength();                 // return length
    double getWidth();                  // return width
    double perimeter();                 // return perimeter of the rectangle
    double area();                      // returns area of the rectangle
  private:
    double upperX, upperY, length, width;
};
```

Write the function definitions for **area()** and **perimeter()**.

```
double rectangle::area() {
  return length*width;
}

double rectangle::perimeter() {
  return 2*length + 2*width;
}
```

Write a code snippet to declare a **rectangle** object and set its upper left corner to (10, 12), its length to 14 and its width to 7.

```
rectangle r;
r.setUpperLeft(10, 12);
r.setLength(14);
r.setWidth(7);
```

Write a function declaration of a member function which overloads the '*****' operator to handle the following from main, assuming that t is an object of type rectangle:

rectangle scaledT = t * 5.5;

rectangle operator*(double);

Now write the function definition of this operator overload, such that the result of '*****' is a rectangle with the length and width of the first operand scaled by the second operand (so if the statement above was executed, **scaledT** would be a rectangle with a length and width 5.5x that of **t**).

```
rectangle rectangle::operator*(double d) {
  rectangle r;
  r.length = length * d;
  r.width = width * d;
  return r;
}
```