More Tutorial on C++:

OBJECT POINTERS

- Accessing members of an object by using the **dot** operator.

```
class D {
    int j;
public:
    void set_j(int n);
    int mul();
};

int main( ) {
    D ob;
    ob.set_j(4);
    cout << ob.mul();
    return 0;

}
```

- Access a member of an object via a pointer to that object.

To obtain the address of an object, precede the object with the **&** operator.

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

class myclass {
  int a;
  public:
      myclass(int x); //constructor
      int get( );
};

myclass::myclass(int x) {
a=x;
}

int myclass::get( ) {
return a;
}
```

```
int main( ) {
  myclass ob(120); //create object
  myclass *p; //create pointer to object
  p=&ob; //put address of ob into p
cout << "value using object: " << ob.get( );
cout << "\n";
cout << "value using pointer: " << p->get( );
return 0;
}
```

Assigning object

One object can be assigned to another provided that both are of the same type. By default, when one object is assigned to another, a copy of all the data members is made.

```
class myclass {
   int a, b;
public:
   void set(int i, int j) { a = i; b = j; };
   void show( ) { cout << a << " " << b << "\n"; }
};

int main( ) {
   myclass o1, o2;
   o1.set(10, 4);
   //assign o1 to o2
   o2 = o1;
   o1.show( );
   o2.show( );
   return 0;
}
```

Output:

```
10  4
10  4
```

Passing object to functions

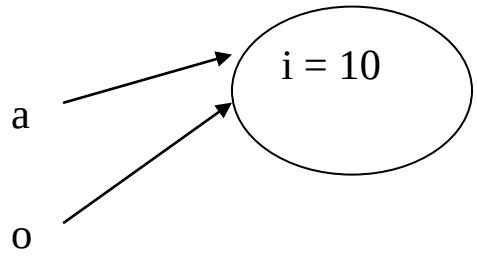
- Pass by value:

The default method of parameter passing in C++, including objects, is by value. Changes to the object inside the function do not affect the object in the call.

```
class samp {
  int i;
public:
  samp(int n) { i = n; }
  int get_i( ) { return i; }
};

// Return square of o.i
int sqr_it(samp o) {
  return o.get_i( )* o.get_i( );
}

int main( ) {
  samp a(10), b(2);
  cout << sqr_it(a) << "\n";
  cout << sqr_it(b) << "\n";
  return 0;
}
```

- Pass by pointer: the argument used in the call can be modify by the function.

```
class samp {
 int i;
public:
  samp(int n) { i = n; }
  int get_i( ) { return i; }
  void set(int n) {i = n;}
};

// Set o.i to its square. This affect the calling argument

void sqr_it(samp *o) {
    o->set(o->get_i( )*o->get_i( ));
}

int main( ) {
samp a(10);
sqr_it(&a); // pass a's address to sqr_it
// ...
}
```



Returning object from functions

Functions can return objects. First, declare the function as returning a class type. Second, return an object of that type using the normal **return** statement.

```
class samp {
  char s[80];
public:
  void show( ) { cout << s << "\n"; }
  void set(char *str) { strcpy(s, str); }
};
```

```
//Return an object of type samp
samp input( ) {
  char t[80];
  samp str;
  cout << "Enter a string: ";
  cin >> t;
  str.set(t);
  return str;
}
```

```
int main( ) {
samp ob;
//assign returned object to ob
ob = input( );
ob.show( );
return 0;
}
```

When an object is returned by a function, a temporary object is automatically created which holds the return value. It is this object that is actually returned by the function. After the value is returned, this object str is destroyed.

ARRAYS, POINTERS, AND REFERENCES

Arrays of objects

The syntax for declaring an array of objects is exactly as that used to declare an array of any other type of variable. Further, arrays of objects are accessed just like arrays of other types of variables.

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

class samp {
  int a;

public:
  samp(int n) {a = n; }
  void set_a(int n) {a = n;}
  int get_a( ) { return a; }
};

int main( ) {
  samp ob[4]; //array of 4 objects
  int i;
  for (i=0; i<4; i++) ob[i].set_a(i);
  for (i=0; i<4; i++) cout << ob[i].get_a( ) << " ";
  cout << "\n";
  return 0;}
```

Multidimensional arrays of objects

```
class samp {
  int a;
public:
  samp(int n) {a = n; }
  int get_a( ) { return a; }
};

int main( ) {
    samp ob[4][2] = {
          1, 2,
          3, 4,
          5, 6,
           7, 8 };
    int i;
    for (i=0; i<4; i++) {
      cout << ob[i][0].get_a( ) << " ";
      cout << ob[i][1].get_a( ) << "\n";
    }
return 0;
}
```

Using pointers to objects

- The object's members are referenced using the arrow (- >) operator instead of the dot (.).
- Pointer arithmetic using an object pointer is the same as it is for any other data type.

```
class samp {
  int a, b;
public:
  samp(int n, int m) {a = n; b = m; }
  int get_a( ) { return a; }
  int get_b( ) { return b; }
};
int main( ) {
  samp ob[4] = {
              samp(1, 2),
              samp(3, 4),
              samp(5, 6),
              samp(7, 8)
};
int i;
samp *p;
p = ob; // get starting address of array
for (i=0; i<4; i++) {
  cout << p->get_a( ) << " ";
  cout << p->get_b( ) << "\n";
  p++; // advance to next object
}
```

The THIS pointer

When a member function is called, how does C++ know which object it was called on?” The answer is that C++ utilizes a hidden pointer named “this”!

```
#include < iostream >
#include < cstring >

class inventory {
  char item[20];
  double cost;
  int on_hand;

public:
  inventory(char *item, double cost, int on_hand) {
      strcpy(this->item, item);
      this->cost = cost;
      this->on_hand = on_hand;
}
void show( );
};
void inventory::show( ) {
  cout << this->item; //use this to access members
  cout << ": £" << this->cost;
  cout << "On hand: " << this->on_hand <<"\n";
}
```

Using NEW and DELETE

- The **new** operator provides dynamic storage allocation

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

int main( ) {
  int *p;
  p = new int; //allocate room for an integer
  if (!p) {
  cout << "Allocation error\n";
  return 1;
}

  *p = 1000;
  cout << "Here is integer at p: " << *p << "\n";
  delete p; // release memory
  return 0;
}
```

Create objects in a C++ program.

- Define a variable as being of a particular class. Space is allocated for the object and the constructor for the object is called. When an object variable goes out of scope, its destructor is called automatically.
- Declare a variable that is a pointer to the object class and call the C++ **new** operator, which will allocate space for the object and call the constructor. In this case, the pointer variable must be explicitly deallocated with the delete operator. The constructor for the object is executed when new is called, and the destructor is executed when delete is called.

```
class samp {
  int i, j;
public:
  void set_ij(int a, int b) { i=a; j=b; }
  int get_product( ) { return i*j; }
};
int main( ) {
  samp *p;
  p = new samp; //allocate object
  if (!p) {
  cout << "Allocation error\n";
  return 1; }
  p- >set_ij(4, 5);
  cout<< "product is: "<< p- >get_product( ) << "\n";
  delete p; // release memory
  return 0; }
```

Example of array allocation

```
// Allocating dynamic objects
class samp {
  int i, j;
public:
  void set_ij(int a, int b) { i=a; j=b; }
  ~samp( ) { cout << "Destroying...\n"; }
  int get_product( ) { return i*j; }
};

int main( ) {
  samp *p;
  int i;
  p = new samp [10]; //allocate object array
  if (!p) {
  cout << "Allocation error\n";
  return 1;
}
for (i=0; i<10; i++) p[i].set_ij(i, i);
for (i=0; i<10; i++) {
  cout << "product [" << i << "] is: ";
  cout << p[i].get_product( ) << "\n";
}
  delete [ ] p; // release memory the destructor should be called 10 times
  return 0;
}
```

References

- **Passing references to objects**

When you pass an object by reference, no copy is made. However, that changes made to the object inside the function affect the object used as argument.

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

void f(int &n); //declare a reference parameter

int main( ) {
  int i=0;
  f(i);
  cout << "Here is i's new value: " << i << "\n";
  return 0;
}

// f( ) now use a reference parameter
void f(int &n) {
  n = 100; // put 100 into the argument
}
```

Returning references

A function can return a reference. It can be employed to allow a function to be used on the left hand side of an assignment statement.

```
int x; // x is a global variable

int main( ) {
  f( ) = 100; // assign 100 to the reference
   // returned by f( ).
  cout << x << "\n";
  return 0;
}

// return an int reference
int &f( ) {
  return x; // return a reference to x
}
```

You must be careful when returning a reference that the object you refer to does not go out of scope.

```
// return an int reference
int &f( ) {
   int x; // x is now a local variable
   return x; // returns a reference to x
}
```

In this case, **x** is now local to **f( )** and it will go out of scope when **f( )** returns. This means that the reference returned by **f( )** is useless.

References restrictions

• You cannot reference another reference.
• You cannot obtain the address of a reference.
• You cannot create arrays of reference.
• References must be initialised unless they are members of a class, or are function parameters.

FUNCTION OVERLOADING

Overloading constructor functions

```
class myclass {
int x;
public:
  myclass( ) { x = 0; } // no initialiser
  myclass(int n ) { x = n; } // initialiser
  int getx( ) { return x; }
};
int main( ) {
  myclass o1(10); // declare with initial value
  myclass o2; // declare without initialiser
  cout << "o1: " << o1.getx( ) << "\n";
  cout << "o2: " << o2.getx( ) << "\n";
  return 0;
}
```

When a dynamic array is allocated as the example blow, it cannot be initialised. Thus, if the class contains a constructor that takes an initialiser, you must include an overloaded version that takes no initialiser.

```
class myclass {
  int x;
public:
  myclass( ) { x = 0; } // no initialiser
  myclass(int n ) { x = n; } // initialiser
  int getx( ) { return x; }
  void setx(int x) { x = n; }
};

int main( ) {
  myclass *p;
  myclass ob(10); // initialise single variable
  p = new myclass[10]; // can't use initialiser here
  if (!p) {
  cout << "Allocation error\n";
  return 1;
  }
  int i;
  for (i=0; i<10; i++) p[i]= ob;
  for (i=0; i<10; i++)
  cout<< "p["<< i << "]: "<< p[i].getx( ) << "\n";
  return 0;}
```

Creating and using a copy constructor

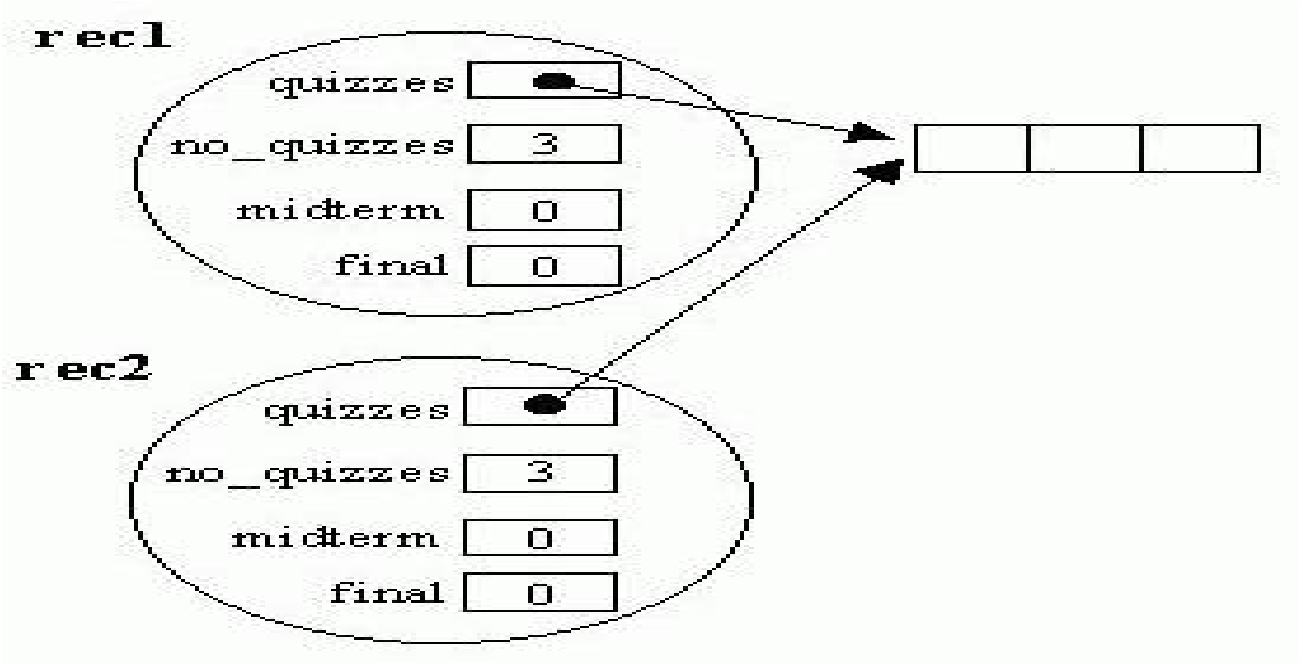
A **copy constructor** is a special constructor in the C++ programming language used to create a new object as a copy of an existing object. This constructor takes a single argument: a reference to the object to be copied.

```
class TestRec
{
public:
   TestRec();
   TestRec(int size);
   TestRec(TestRec&);
   ~TestRec();          // DESTRUCTOR
   void SetScore(int i, int score);
   char Grade();              // returns a letter grade

private:
   int* quizzes;     // points to a dynamic array
   int no_quizzes;   // keep track of the length of quizzes array
   int midterm;
   int final;
};
```

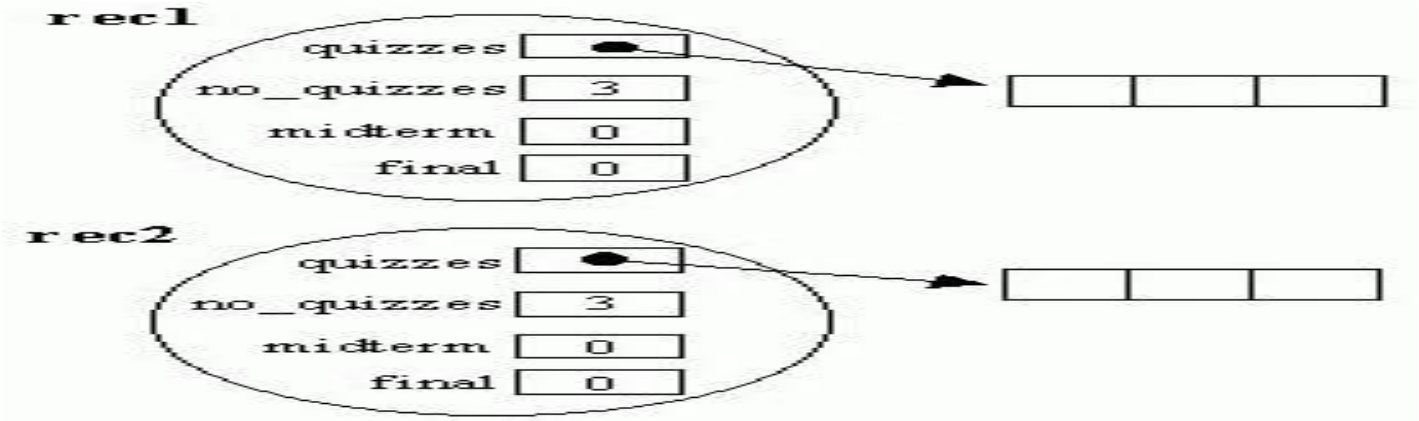
```
TestRec::TestRec(){
   no_quizzes = 1;        //
   quizzes = new int[1];  // dynamic array of size 1; non-NULL
   midterm = 0;
   final = 0;
}

TestRec::TestRec(int size){
   no_quizzes = size;
   quizzes = new int[no_quizzes];  // dynamic array of the given size
   midterm = 0;
   final = 0;
}

// Destructor definition
TestRec::~TestRec()
{
   delete [] quizzes;  // destructor deallocates dynamic array
}
```

A big problem with using a pointer inside class -- **when an object is created as a copy of another object, dynamic array will end up being shared** because only the value in the pointer cell is copied. This scheme is called **"shallow copying"**.

```
int main() {
  TestRec rec1(3);
    ...
  TestRec rec2 = rec1;  // DANGER!! rec2's quizzes points to rec1's quizzes
```



- What we want to do is to do **"deep copying"**, to make a separate array for the object.



To do so, we must write **copy constructor.**

```
// Copy all information in rec (parameter) to the class object
TestRec::TestRec(const TestRec& rec) {
   no_quizzes = rec.no_quizzes;
   quizzes = new int[no_quizzes];

   for (int i = 0; i < no_quizzes; i++) {
      quizzes[i] = rec.quizzes[i];
   }
   midterm = rec.midterm;
   final = rec.final;}
```

The most common form of copy constructor is shown here:

```
class-name(const class-name &obj) {
// body of constructor
}
```

Copy constructor is automatically called when objects are copied:

1. Object is **passed by value**

```
bool isPass(const TestRec rec) // rec passed by value -- copy constructor fired
{ char g = rec.Grade();
  if (g == 'A' || g == 'B' || g == 'C')
       return true;
  else return false;}
```

2. Object is **returned by value**

```
TestRec getInput()       // return by value
{ int size;
  cout << "Enter the number of quizzes: ";
  cin >> size;
  TestRec rec(size);  // local object by constructor
  for (int i = 0; i < size; i++) {
        int score;
        cout << "Enter a score: ";
        cin >> score;
        rec.SetScore(i, score);
   }
   return rec; } // object return by value -- copy constructor fired
```

3. Object is **declared and initialized** to another object

```
int main() {
  TestRec rec1(10);
  ...
  TestRec rec2(rec1),  // declared and initialized by copying from rec1
  TestRec rec4 = getInput(); // declared and initialized to the result of
            //  the function call (getInput)
```

Constructors, destructors, and inheritance

It is possible for the base class, the derived class, or both to have constructor and destructor functions.

When a base class and a derived class both have constructor and destructor functions, the constructor functions are executed in order of derivation. The destructor functions are executed in reverse order.

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

class base {
public:
  base( ) { cout << "Constructing base\n"; }
  ~base( ) { cout << "Destructing base\n"; }
};

class derived : public base {
public:
  derived( ) { cout << "Constructing derived\n"; }
  ~derived( ) { cout << "Destructing derived\n; }
};
int main( ) {
  derived obj;
  return 0;
}
```

```
This program displays:

Constructing base
Constructing derived
Destructing derived
Destructing base
```

The following program shows how to pass an argument to a derived class' constructor.

```
#include <iostream>
using namespace std;
class base {
public:
  base( ) { cout << "Constructing base\n"; }
  ~base( ) { cout << "Destructing base\n"; }
};

class derived : public base {
  int j;
public:
  derived(int n) {
      cout << "Constructing derived\n";
      j = n;
}
~derived( ) { cout << "Destructing derived\n; }
void showj( ) { cout << j << "\n"; }
};
int main( ) {
  derived o(10);// 10 is passed in the normal fashion
  o.showj( );
  return 0;
}
```

In the following example both the derived class and the base class take arguments:

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

class base {
  int i;
public:
  base(int n) {
  cout << "Constructing base\n";
  i = n;
}
~base( ) { cout << "Destructing base\n"; }
void showi( ) { cout << i << "\n"; }
};

class derived : public base {
  int j;

public:
  derived(int n) : base(n) {// pass argument to the base class
  cout << "Constructing derived\n";
  j = n;
}
~derived( ) { cout << "Destructing derived\n; }
  void showj( ) { cout << j << "\n"; }
};
```

```
int main( ) {
  derived o(10);
  o.showi( );
  o.showj( );
  return 0;
}
```

In most cases, the constructor functions for the derived class and the base class *will not* used the same argument. When this is the case, you need to pass one or more arguments to each, you must pass to the derived class constructor ***all*** arguments needed by ***both*** the derived class and the base class. Then, the derived class simply passes along to the base class those arguments required by it.

```
class base {
  int i;
public:
  base(int n) {
  cout << "Constructing base\n";
  i = n;
}
~base( ) { cout << "Destructing base\n"; }
void showi( ) { cout << i << "\n"; }
};

class derived : public base {
  int j;
public:
  derived(int n, int m) : base(m) {//pass argument to the base class
  cout << "Constructing derived\n";
  j = n;
}
~derived( ) { cout << "Destructing derived\n; }
void showj( ) { cout << j << "\n"; }
};
```

```
int main( ) {
derived o(10, 20); // 20
     //is pass to base()
o.showi( );
o.showj( );
return 0;
}
```

Template

```
template <class ttype>
ttype minimum (ttype a, ttype b){
   ttype r;
   r = a;
   if (b < a) r = b;
   return r;}

int main (){
   int i1, i2, i3;
   i1 = 34;
   i2 = 6;
   i3 = minimum (i1, i2);
   cout << "Most little: " << i3 << endl;
   double d1, d2, d3;
   d1 = 7.9;
   d2 = 32.1;
   d3 = minimum (d1, d2);
   cout << "Most little: " << d3 << endl;
   cout << "Most little: " << minimum (d3, 3.5) << endl;
   return 0;
}
```

The C++ compiler generates two versions of minimum:
int minimum (int a, int b) and
double minimum (double a, double b).

Overloaded Operators:

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

class vector{
public:
   double x;
   double y;


   vector (double = 0, double = 0);
   vector operator + (vector);
   double module();
   void set_length (double = 1);
};

vector::vector (double a, double b){
   x = a;
   y = b;

}
```

```
vector vector::operator + (vector a){
   return vector (x + a.x, y + a.y);
}

double vector::module(){
   return sqrt (x * x + y * y);}

void vector::set_length (double a){
   double length = this->module();

   x = x / length * a;
   y = y / length * a;}

int main (){
   vector a;
   vector b;
   vector c (3, 5);

   a = b + c;
   cout << "The module of vector c: " << c.module() <<
endl;

}
```

Static data

A data member of a class can be declared static in the public or private part of the class definition. Such a data member is created and initialized only once, in contrast to non-static data members, which are created again and again, for each separate object of the class.

Static data members are created when the program starts executing. Note, however, that they are always creates as members of their classes.

Static data members which are declared public are like `normal' global variables: they can be reached by *all code of the program* by simply using their names, together with their class names and the scope resolution operator.

```
class Test    {
    public:
        static int public_int;
    private:
        static int private_int;
};
int main()    {
    Test::public_int = 145;     // ok
    Test::private_int = 12;     // wrong, don't touch the private parts
    return (0);    }
```

Private static data

```
class Directory{
    public:
        // constructors, destructors, etc. (not shown)
    private:
        static char path[];
};
```

The data member path[] is a *private static data member*. During the execution of the program, only *one* Directory::path[] exists, even though more than one object of the class Directory may exist.

This private static data member could be inspected or altered by the constructor, destructor or by any other member function of the class Directory.

Since constructors are called for each new object of a class, static data members are never *initialized* by constructors.

The reason for this is that the static data members exist *before* any constructor of the class has been called. The static data members can be initialized during their definition, outside of all member functions, in the same way as global variables are initialized.

Static member functions

The differences between a static member function and non-static member functions are as follows.

- A static member function can access only static member data, static member functions and data and functions outside the class. A non-static member function can access all of the above including the static data member.
- A static member function can be called, even when a class is not instantiated. A non-static member function can be called only after instantiating the class as an object.
- A static member function cannot be declared virtual, whereas a non-static member functions can be declared as virtual
- A static member function cannot have access to the 'this' pointer of the class.

The static member functions are not used very frequently in programs. But nevertheless, they become useful whenever we need to have functions which are accessible even when the class is not instantiated.

```
class stat{
        int num;
        public:
        stat(int n = 0) {num=n;}
        static void print() {cout <<"static member function" <<endl;
        };

void main() {
      stat::print(); //no object instance required
      stat s(1);
      s.print(); //but still can be called from an object
}//end main
```