Logging in From a Linux System or Localhost
4 days 45 min ago

Auto response
4 days 6 hours ago

Re: final issue.
4 days 22 hours ago

Re: Rocky, thank you for this
4 days 23 hours ago

Learning C/C++ Step-By-Step - Page 16

Submitted by ganesh35 (Contact Author) (Forums) on Wed, 2009-01-07 18:52. ::

16. Step-by-Step C/C++ --- C++ Programming - Polymorphism

Polymorphism

1. Function Overloading

If a function with its name differed by arguments behavior is called functions polymorphism or function overloading.

```
// An example program to demonstrate the use of function overloading
#include <iostream>
using namespace std;
void printline()
{
     for(int i=0;i<=80; i++) cout << "-";
}

void printline(int n)
{
     for(int i =0 ;i<=n;i++) cout << "-";
}

void printline(int n,char ch)
{
     for(int i=0;i<=n; i++) cout << ch;
}

int main()
{
     printline();
     printline(5);
     printline(10, '*');
     return 0;
}
```

Polymorphism

Polymorphism is one of the crucial features of OOP. It simply means one name, multiple forms. We have already seen how the concept of polymorphism is

implemented using overloaded functions and operators. The overloaded member functions are selected for invoking by matching arguments, both type and number. The compiler knows this information at the compile time and therefore compiler is able to select the appropriate function for a particular call at the compile time itself. This is called ***early binding*** or ***static binding*** or ***static linking***. Also known as ***compile time polymorphism***, ***early binding*** simply means that an object is bound to its functions call at compile time.

Now let us consider a situation where the function name and prototype is the same in both the base and derived classes. For example, considers the following class definitions.

```
#include <iostream>
using namespace std;
class A
{
     int x;
     public : void show();
};

class B : public A
{
     int y;
     public : void show();
};

int main()
{
     B b;
     b.show();
     return 0;
}
```

How do we use the member function ***show( )*** to print the values objects of both the classes A and B ? Since the prototype of ***show( )*** is the same in the both places, the function is not overloaded and therefore static binding does not apply. In fact, the compiler does not know what to do and defers the decision.

It would be nice if the appropriate member function could be selected while the program is running. This is known as ***runtime polymorphism***. How could it happen? C++ supports a mechanism known as **virtual** function to achieve runtime polymorphism. At runtime, when it is known what class objects are under consideration, the appropriate version of the function is called.

Since the function is linked with a particular class much later after the compilation, this process is termed as ***late binding***. It is also known as ***dynamic binding*** or ***dynamic linking*** because the selection of the appropriate function is done dynamically at runtime.

3. Types of Polymorphism

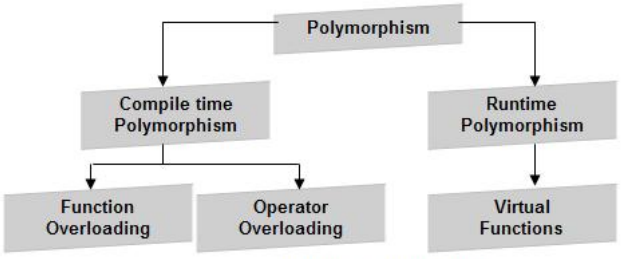
Polymorphism is of two types namely.

1 Compile time polymorphism
Or Early binding
Or Static binding
Or Static linking polymorphism.

An object is bound to its function call at compile time.

2 Runtime polymorphism
Or late binding
Or Dynamic binding
Or Dynamic linking polymorphism.

The selection and appropriate function is done dynamically at run time.



Achieving polymorphism

Dynamic binding is one of the powerful features of C++. This requires the use of pointers to objects. We shall discuss in detail how the object pointers and virtual functions are used to implement dynamic binding.

4. Normal Member Functions Accessed with Pointers

The below program consist of a base class

```
/*   Normal functions accessed from pointer */
/* Polymorphism with classes (without using VIRTUAL polymorphism */
#include <iostream>
using namespace std;
```

```
class BASE
{
     public :
          void disp() { cout << "\nYou are in BASE class "; }
};

class DERIVED1 : public BASE
{
     public :
          void disp() { cout << "\nYou are in DERIVED1 class"; }
};

class DERIVED2 : public BASE
{
     public :
          void disp() { cout << "\nYou are in DERIVED2 class"; }
};

int main()
{
      DERIVED1 d1;               // Object of derived class 1
      DERIVED2 d2;               // Object of derived class 2
      BASE *b;                      // pointer to base class

      b=&d1;                          // Assign address of d1 in pointer b
      b->disp();                      // call to disp()
      b=&d2;                          // Assign address of d2 pointer b
      b->disp();                      // call to disp()
      return 0;
}
```

The above program demonstrates:

- A BASE class
- DERIVED1, DERIVED2 classes derived from BASE
- Derived classes objects (d1,d2)
- BASE class pointer *b

Output

You are in BASE class
You are in BASE class

5. Virtual Function

Virtual means ***existing in effect but not in reality.***
A member function can be made as virtual function by preceding the member function with the keyword ***virtual.***

```
/* Polymorphism with Classes (Virtual polymorphism) */

#include <iostream>
using namespace std;
class B
{
     public :
          void show(){ cout << "\nclass B method Show() "; }
          virtual void disp() { cout << "\nclass B method disp()"; }
};

class D : public B
{
     public :
          void show(){cout << "\nclass D method Show() "; }
          void disp(){ cout << "\nclass D method disp()"; }
};

int main()
{
     D d1;
     d1.show();
     d1.disp(); // Base class member

     B b;
     D d;
     B *Bptr;
     Bptr = &b;
     Bptr->show();
     Bptr->disp(); // Base class member

     Bptr=&d;
     Bptr->show(); // derived class members
     Bptr->disp(); // Base class member
     return 0;
}
```

Output

class D method Show()
class D method disp()

6. Pure Function

A function defined in a base class and has no definition relative to derived class is called pure function. In simple words ***a pure function is a virtual function with no body.***

```
#include <iostream>
using namespace std;
class B
{
     public :
          void show(){ cout << "\nclass B method Show() "; }
          virtual void disp() = 0; // pure virtual function
};

class D : public B
{
     public :
          void show(){cout << "\nclass D method Show() "; }
          void disp(){ cout << "\nclass D method disp()"; }
};

int main()
{
     D d1;
     d1.show(); // O/P : Class D method show()
     d1.disp(); // O/P : Class D method disp()

     D d;
     B *Bptr;

     Bptr=&d;
     Bptr->show(); // O/P : Class B method show()
     Bptr->disp(); // O/P : Class D method disp()
     return 0;
}
```

Bptr -> show() is the default executable function from Base
Bptr -> disp() is the default executable function from Base but it is declared as a virtual pure function so at runtime Derived class's ***disp()*** will be called.

```
/* Program to demonstrate the advantage of pure virtual functions */

#include <iostream>
using namespace std;
enum boolean { false, true };
class NAME
{
     protected : char name[20];
     public :
          void getname()
          { cout << "Enter name :"; cin >> name; }

          void showname()
          { cout << "\nName is "<< name; }

          boolean virtual isGradeA() = 0; // pure virtual function
};

class student : public NAME
{
     private : float avg;
     public :
          void getavg()
          {
               cout << "\nEnter Student Average :";
               cin >> avg;
          }

          boolean isGradeA()
          { return (avg>=80) ? true : false ; }
};

class employee : public NAME
{
     private : int sal;
     public :
          void getsal()
          { cout << "\nEnter salary "; cin >> sal; }

          boolean isGradeA()
          { return (sal>=10000) ? true : false ; }
};

int main()
{
     NAME *names[20]; // no of pointer to name
     student *s; // pointer to student
     employee *e; // pointer to employee
     int n = 0; // no of Names on list
     char choice;
     do{
          cout << "Enter Student or Employee (s/e) ";
          cin >> choice;
          if(choice=='s')
          {
               s = new student; // make a new student
               s->getname();
               s->getavg();
               names[n++]=s;
          }
          else
          {
               e = new employee; // make a new employee
               e->getname();
               e->getsal();
               names[n++]=e;
          }
          cout << "Enter another (y/n) ?"; // do another
          cin >> choice;
     } while(choice=='y');
     for(int j=0; j<n; j++)
     {
          names[j]->showname( );
```

```
          if(names[j]->isGradeA( )==true)
               cout << "He is Grade 1 person";
     }
     return 0;
}
```


7. Assignment and Copy-Initialization

The C++ compiler is always busy on your behalf, doing things you can't be bothered to do. If you take charge, it will defer to your judgement; otherwise it will do things its own way. Two important examples of this process are the **assignment operator and the copy constructor.**

You've used the assignment operator many times, probably without thinking too much about it. Suppose **a1** and **a2** are objects. Unless you tell the compiler otherwise, the statement.

a2 = a1; // set a2 to the value of a1

Will cause the compiler to copy the data from **a1,** member-by-member, into **a2.** This is the default action of the assignment operator, =.

You're also familiar with initializing variables, initializing an object with another object, as in

alpha a2(a1); // initialize a2 to the value of a1

Causes a similar action. The compiler creates a new object, **a2,** and copies the data from **a1,** member-by-member, into **a2**. This is the default action of the copy constructor.

Both these default activities are provided, free of charge, by the compiler. If member-by-member copying is what you want, you need take no further action. However, if you want assignment of initialization to do something more complex, then you can override the default functions. We'll discuss the techniques for overloading the assignment operator and the copy constructor separately.

Overloading the Assignment Operator

```
// Overloading the Assignment ( = ) Operator
#include <iostream>
using namespace std;
class alpha
{
     private:
          int data;
     public:
          alpha() { } // no-arg constructor
          alpha( int d )
          { data = d; } // one-arg constructor
          void display()
          { cout << data; } // Display data
          alpha operator =(alpha & a) // overloaded = operator
          {
               data = a.data; // not done automatically
               cout << "\n Assignment operator invoked ";
               return alpha(data);
          }
};

int main()
{
     alpha a1(37);
     alpha a2;
     a2 = a1; // Invoke overloaded =
     cout << "\n a2 = "; a2.display(); // display a2

     alpha a3 = a2; // does NOT invoke =
     cout << "\n a3 = "; a3.display(); // display a3
     return 0;
}
```

Output:

a2 = 37
a3 = 37

8. The COPY Constructor

As we discussed, we can define and at the same time initialize an object to the value of another object with two kinds of statement:

alpha a3(a2); // Copy initializing
alpha a3 = a2; // copy initialization, alternate syntax

Both styles of definition invoke a **copy constructor:** that is, a constructor that copies its argument into a new object. The default copy constructor, Which is provided automatically by the compiler for every object, performs a member-by-member copy. This is similar to what the assignment operator does; the difference is that the copy constructor also creates also creates a new object.

The following example demonstrates the copy constructor.

```
#include <iostream>
using namespace std;
class alpha
{
     private :
          int data;
     public:
          alpha( ) { } // no-args constructor
          alpha(int d) { data = d; } // one-arg constructor
          alpha(alpha& a) // copy constructor
          {
               data = a.data;
               cout << "\nCopy constructor invoked";
          }
          void display( )
          { cout << data; }
          void operator = (alpha& a) // overloaded = operator
          {
               data = a.data;
               cout << "\nAssignment operator invoked";
          }
};
```

```
int main()
{
     alpha a1( 37 );
     alpha a2;

     a2 = a1; // invoke overloaded =
     cout << "\na2 = "; a2.display(); // display a2

     alpha a3( a1 ); // invoke copy constructor

     // alpha a3 = a1; // equivalent definition of a3

     cout << "\na3 = "; a3.display(); // display a3
     return 0;
}
```

The above program overloads both the assignment operator and the copy constructor.
The overloaded assignment operator is similar to that in the past example.

9. 'this' Pointer

C++ uses a unique keyword called **this** to represent an object that invokes a member functions. **This** is a pointer that points to the object for which *this* function was called.

This pointers simply performs make task **to return object it self.**

The following program defines **i, j** objects and **i** is assigned with the value of 5 and the entire object of **i** is assigned by its member function to **j**

```
#include <iostream>
using namespace std;
class A
{
     int a;
     public:
          A() { }
          A(int x)
          { a = x; }
          void display()
          {
               cout << a;
          }
          A get()
          {

               return *this; // Return it self

          }
};
int main()
{
     A i(5);
     A j;

     j = i.get();
     j.display();
     return 0;
}
```

Ref: Object-oriented Programming in Turbo C++: Robert Lafore

previous up
Learning C/C++ Step-By-Step - Page 15

add comment | view as pdf | print: this | all page(s)

Related Tutorials

- Beginner's Guide To c++
- An Explanation of Pointers (C++)

Worth Learning in scarcity of time :D
Submitted by Iftekhar Ahmed (not registered) on Fri, 2010-05-14 20:31.

Dude.....

Its relly helpful, I have 48 Hrs for my Exams.....

Thanks for posting .....

Regards,

Iftekhar Ahmed

reply | view as pdf

Nice tutorial
Submitted by Paul (not registered) on Wed, 2009-01-07 23:40.

Thanks, this is a nice tutorial, I'm actually a php programer and I saw a lot similar with php's language, but I have a question:

Can I compile a c++ program under linux to make a .exe to run on windows and viceversa?

And how to do that if it is possible?

reply | view as pdf

Re: Nice tutorial
Submitted by dbrion (not registered) on Tue, 2009-06-30 12:58.

"Can I compile a c++ program under linux to make a .exe to run on windows "

Yes :

I confirm the last post and cross compilers can natively be found with recent Fedora (and any linux distribution, messeems!) The way all the stuff necessary could be downloaded and made working can be found with adaptations of :

cran.**r**-project.org/doc/contrib/**cross**-build.pdf

(one just has to stop at "make xtools": they are downloaded and put into a proper place).

Else, if you like linux and have "only" Windows, you can use cygwin (avoids dualbooting or switching to another computer) with the sources (it is very rare that sources need to be different between both systems, they use the same compiler -gcc/g++- and almost the same include file names....) ...

"and viceversa"

No (not because linux executables do not have a .exe suffix!):

but you can edit/compile/test your program(s) under cygwin (google search can find it) until it works and port it (i.E the source, the compilation and test scripts) to Linux (there is almost no effort, apart the efforts linked with different disk nomenclatures, of course).

Else, unless emulating linux with vmplayer/qemu/vbox -all these emulators are Windows ported-? but it is more complicated than C/C++....

reply | view as pdf

Re: Nice tutorial
Submitted by Abhijeet (not registered) on Sun, 2009-01-11 02:25.

Though i haven't done it myself, mingw32 can create win32 executables from linux. As its description says "Minimalist GNU win32 (cross) compiler, A Linux hosted, win32 target, cross compiler for C/C++, Freedom through obsolescence. Those who still really need to can now build windows executables from the comfort of Debian." You could check it out.

reply | view as pdf