Polymorphism

Polymorphism

- The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

- Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee.

- So a same person posses have different behavior in different situations. This is called polymorphism.

Types of Poymorphism

- Compile Time Polymorphism
- Run Time Polymorphism

Compile Time Polymorphism

- Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation).
- Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators.
- The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

Run Time Polymorphism

- The opposite of early binding is late binding.
- As it relates to C++, late binding refers to function calls that are not resolved until run time.
- As function calls are not determined at compile time, the object and the function are not linked until run time.

Run Time Polymorphism

- The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code."
- Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times.
- Virtual functions are used to achieve late binding.

Function Overloading

- Function overloading is the process of using the same name for two or more functions.
- The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters.
- It is only through these differences that the compiler knows which function to call in any given situation.

Example

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in types of parameters
double myfunc(double i);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(5.4); // calls myfunc(double i)
return 0;
}
```

```
 double myfunc(double i)
 {
 return i;
}
int myfunc(int i)
{
 return i;
 }
```

Output:
10 5.4

Example 2

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(4, 5); // calls myfunc(int i, int j)
return 0;
}
```

```
int myfunc(int i)
{
return i;
}
int myfunc(int i, int j)
{
return i*j;
}
```

Output:
10 20

Key Points

- The key point about function overloading is that the functions must differ in regard to the types and/or number of parameters. Two functions differing only in their return types cannot be overloaded.
- For example, this is an invalid attempt to overload **myfunc( ):**

```
int myfunc(int i);
float myfunc(int i);
```

// Error: differing return types are insufficient when overloading.

- Sometimes, two function declarations will appear to differ, when in fact they do not. For example, consider the following declarations.

```
void f(int *p);
void f(int p[]); // error, *p is same as p[]
```

- Remember, to the compiler ***p is the same as p[ ]. Therefore, although the two** prototypes appear to differ in the types of their parameter, in actuality they do not.

Overloading Constructors

- Many times you will create a class for which there are two or more possible ways to construct an object.
- In these cases, you will want to provide an overloaded constructor for each way.
- This is a self-enforcing rule because if you attempt to create an object for which there is no matching constructor, a compile-time error results.

Example

```
#include <iostream>
#include <cstdio>
using namespace std;
class date {
int day, month, year;
public:
date(char *d);
date(int m, int d, int y);
void show_date();
};
// Initialize using string.
date::date(char *d)
```

```
 void date::show_date(){
cout << month << "/" << day;
cout << "/" << year << "\n";
}
int main()
{
date ob1(12, 4, 2003), ob2("10/22/2003");
ob1.show_date();
ob2.show_date();
return 0;
}
```

Output:
12/4/2003
10/22/2003

Allowing Both Initialized and Uninitialized Objects

- Another common reason constructors are overloaded is to allow both initialized and uninitialized objects (or, more precisely, default initialized objects) to be created.
- This is especially important if you want to be able to create dynamic arrays of objects of some class, since it is not possible to initialize a dynamically allocated array.
- To allow uninitialized arrays of objects along with initialized objects, you must include a constructor that supports initialization and one that does not.

Example

```
#include <iostream>
#include <new>
using namespace std;
class powers {
int x;
public:
// overload constructor two ways
powers() { x = 0; } // no initializer
powers(int n) { x = n; } // initializer
int getx() { return x; }
void setx(int i) { x = i; }
};
int main()
{
powers ofTwo[] = {1, 2, 4, 8, 16}; // initialized
powers ofThree[5]; // uninitialized
```

```
// show powers of three
cout << "Powers of three: ";
for(i=0; i<5; i++) {
cout << ofThree[i].getx() << " ";
}
cout << "\n\n";
// dynamically allocate an array

p = new powers[5]; // no initialization
}
```

Output:
Powers of two: 1 2 4 8 16
Powers of three: 1 3 9 27 81
Powers of two: 1 2 4 8 16