VIRTUAL FUNCTIONS AND POLYMORPHISM

VIRTUAL FUNCTIONS

A *virtual function* is a member function that is declared within a base class and redefined by a derived class.

To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**.

LATE BINDING VS EARLY BINDING

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.)

Ex. Operator overloading, function overlaoding

Late binding refers to function calls that are not resolved until run time.

Ex. Virtual functions are used to achieve late binding.

```cpp
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
public:
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
```

```cpp
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

OUTPUT

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

```
d2.vfunc(); // calls derived2's vfunc()
```

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of **vfunc()** .

it is only when access is through a base-class pointer (or reference) that run-time polymorphism is achieved

CALLING A VIRTUAL FUNCTION THROUGH A BASE CLASS REFERENCE

```
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
public:
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
// Use a base class reference parameter.
void f(base &r) {
r.vfunc();
}
```

```
int main()
{
base b;
derived1 d1;
derived2 d2;
f(b);//passbase
   object to f()
f(d1); // pass a derived1
   object to f()
f(d2); // pass a derived2
   object to f()
 }
```

THE VIRTUAL ATTRIBUTE IS INHERITED

```
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
/* derived2 inherits virtual function
vfunc()
from derived1. */
class derived2 : public derived1 {
public:
// vfunc() is still virtual
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
```

```
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc();
p = &d1;
p->vfunc();
p = &d2;
p->vfunc();
return 0;
}
```

VIRTUAL FUNCTIONS ARE HIERARCHICAL

```
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
```

```
class derived2 : public base {
public:
// vfunc() not overridden by derived2, base's is used
};
```

```
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // use base's vfunc()
return 0;
}
```

PURE VIRTUAL FUNCTIONS

A *pure virtual function* is a virtual function that has no definition within the base class. To declare a pure virtual function, use this general form:

virtual *type func-name(parameter-list*) = 0;

```cpp
#include <iostream>
using namespace std;
class number {
protected:
int val;
public:
void setval(int i) { val = i; }
// show() is a pure virtual function
virtual void show() = 0;
};
class hextype : public number {
public:
void show() {
cout << hex << val << "\n";
}
};
class dectype : public number {
public:
void show() {
cout << val << "\n";
}
};
```

```cpp
class octtype : public number {
public:
void show() {
cout << oct << val << "\n";
}
};

int main()
{
dectype d;
hextype h;
octtype o;
d.setval(20);
d.show(); // displays 20 - decimal
h.setval(20);
h.show(); // displays 14 - hexadecimal
o.setval(20);
o.show(); // displays 24 - octal
return 0;
}
```

ABSTRACT CLASSES

A class that **contains at least one pure virtual function** is said to be *abstract*.

Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), **no objects of an abstract class may be created.**

you can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism

GENERIC CLASSES

template <class *Ttype*> class *class-name* {
.
..
};

class-name <*type*> *ob*;

```
// This function demonstrates a generic stack.
#include <iostream>
using namespace std;
const int SIZE = 10;
// Create a generic stack class
template <class StackType> class stack {
StackType stck[SIZE]; // holds the stack
int tos; // index of top-of-stack
public:
stack() { tos = 0; } // initialize stack
void push(StackType ob); // push object on stack
StackType pop(); // pop object from stack
};
// Push an object.
template <class StackType> void stack<StackType>::push(StackType ob)
{
if(tos==SIZE) {
cout << "Stack is full.\n";
return;
}
stck[tos] = ob;
tos++;
}
// Pop an object.
template <class StackType> StackType stack<StackType>::pop()
{
if(tos==0) {
cout << "Stack is empty.\n";
return 0; // return null on empty stack
}
tos--;
return stck[tos];
}

int main()
{
// Demonstrate character stacks.
stack<char> s1, s2; // create two charact
int i;
s1.push('a');
s2.push('x');
s1.push('b');
s2.push('y');
s1.push('c');
s2.push('z');
for(i=0; i<3; i++) cout << "Pop s1: " << s
"\n";
for(i=0; i<3; i++) cout << "Pop s2: " << s
"\n";
// demonstrate double stacks
stack<double> ds1, ds2; // create two doub
ds1.push(1.1);
ds2.push(2.2);
ds1.push(3.3);
ds2.push(4.4);
ds1.push(5.5);
ds2.push(6.6);
for(i=0; i<3; i++) cout << "Pop ds1: " <<
"\n";
for(i=0; i<3; i++) cout << "Pop ds2: " <<
```

AN EXAMPLE WITH TWO GENERIC DATA TYPES

```
template <class Type1, class Type2> class
myclass
{
Type1 i;
Type2 j;
public:
myclass(Type1 a, Type2 b) { i = a; j = b; }
void show() { cout << i << ' ' << j <<
'\n'; }
};
int main()
{
myclass<int, double> ob1(10, 0.23);
myclass<char, char *> ob2('X', "Templates
add power.");
ob1.show(); // show int, double
ob2.show(); // show char, char *
return 0;
}
```

USING NON-TYPE ARGUMENTS WITH GENERIC CLASSES

```
// Demonstrate non-type template arguments.
#include <iostream>
#include <cstdlib>
using namespace std;
// Here, int size is a non-type argument.
template <class AType, int size> class atype {
AType a[size]; // length of array is passed in size
public:
atype() {
register int i;
for(i=0; i<size; i++) a[i] = i;
}
AType &operator[](int i);
};
// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
if(i<0 || i> size-1) {
cout << "\nIndex value of ";
cout << i << " is out-of-bounds.\n";
exit(1);
}
return a[i];
}
```

```
int main()
{
atype<int, 10> intob;
atype<double, 15> doubleob; int i;
cout << "Integer array: ";
for(i=0; i<10; i++) intob[i] = i;
for(i=0; i<10; i++) cout << intob[i] << " ";
cout << '\n';
cout << "Double array: ";
for(i=0; i<15; i++) doubleob[i] = (double) i/3;
for(i=0; i<15; i++) cout << doubleob[i] << " ";
cout << '\n';
intob[12] = 100; // generates runtime error
return 0;
}
```

USING DEFAULT ARGUMENTS WITH TEMPLATE CLASSES

```
// Here, AType defaults to int and size defaults
to 10.
template <class AType=int, int size=10> class
atype {
AType a[size]; // size of array is passed in size
public:
atype() {
register int i;
for(i=0; i<size; i++) a[i] = i;
}
AType &operator[](int i);
};
```

```
atype<int, 100> intarray; // integer array, size
100
atype<double> doublearray; // double array, default
size
atype<> defarray; // default to int array of size
10
```

Templates

```
template <class Ttype> ret-type func-name(parameter list)
{
// body of function
}
```

- *Ttype* is a placeholder name for a data type used by the function.

```
// Function template example.
#include <iostream>
using namespace std;
// This is a function template.
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

```
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j <<
'\n';
cout << "Original x, y: " << x << ' ' << y <<
'\n';
cout << "Original a, b: " << a << ' ' << b <<
'\n';
swapargs(i, j); // swap integers
swapargs(x, y); // swap floats
swapargs(a, b); // swap chars
cout << "Swapped i, j: " << i << ' ' << j <<
'\n';
cout << "Swapped x, y: " << x << ' ' << y <<
'\n';
cout << "Swapped a, b: " << a << ' ' << b <<
'\n';
return 0;
}
```

Another way of writing templates

```
template <class X>
void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

A Function with Two Generic Types

```
#include <iostream>
using namespace std;
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
cout << x << ' ' << y << '\n';
}
int main()
{
myfunc(10, "I like C++");
myfunc(98.6, 19L);
return 0;
}
```

Explicitly Overloading a Generic Function

```
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
cout << "Inside template swapargs.\n";
}
// This overrides the generic version of
swapargs() for ints.
void swapargs(int &a, int &b)
{
int temp;
temp = a;
a = b;
b = temp;
cout << "Inside swapargs int”;
}
```

```
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j
<< '\n';
cout << "Original x, y: " << x << ' ' << y
<< '\n';
cout << "Original a, b: " << a << ' ' << b
<< '\n';
swapargs(i, j); // calls explicitly
overloaded swapargs()
swapargs(x, y); // calls generic swapargs()
swapargs(a, b); // calls generic swapargs()
cout << "Swapped i, j: " << i << ' ' << j
<< '\n';
cout << "Swapped x, y: " << x << ' ' << y
<< '\n';
cout << "Swapped a, b: " << a << ' ' << b
<< '\n';
return 0;
}
```

Overloading a Function Template

```
// Overload a function template declaration.
#include <iostream>
using namespace std;
// First version of f() template.
template <class X> void f(X a)
{
cout << "Inside f(X a)\n";
}
// Second version of f() template.
template <class X, class Y> void f(X a, Y b)
{
cout << "Inside f(X a, Y b)\n";
}
int main()
{
f(10); // calls f(X)
f(10, 20); // calls f(X, Y)
return 0;
}
```

Using Standard Parameters with Template Functions

```
const int TABWIDTH = 8;
// Display data at specified tab position.
template<class X> void tabOut(X data, int
tab)
{
for(; tab; tab--)
for(int i=0; i<TABWIDTH; i++) cout << ' ';
cout << data << "\n";
}
int main()
{
tabOut("This is a test", 0);
tabOut(100, 1);
tabOut('X', 2);
tabOut(10/3, 3);
return 0;
}
```

Generic Function Restrictions

- When functions are overloaded, you may have different actions performed within the body of each function. But a generic function must perform the same general action for all versions—only the type of data can differ

- These functions could *not* be replaced by a generic function because they do not do the same thing

```
#include <iostream>
#include <cmath>
using namespace std;
void myfunc(int i)
{
cout << "value is: " << i << "\n";
}
void myfunc(double d)
{
double intpart;
double fracpart;
fracpart = modf(d, &intpart);
cout << "Fractional part: " << fracpart;
cout << "\n";
cout << "Integer part: " << intpart;
}
int main()
{
myfunc(1);
myfunc(12.2);
return 0;
}
```

Applying Generic Functions

- A Generic Sort
- Compacting an Array