POLYMORPHISM – 1 PART

- Run-time vs. compile-time binding in C++
- Name Overloading, Name Overriding, and Name Hiding
#include <iostream>
using namespace std;

void sayHi( );
int main( ) {
    sayHi( );
    return 0;
}

void sayHi( ) {
    cout << "Hello, cruel world!" << endl;
}
The compiler binds any call to `sayHi`, such as the one in `main`, to the code that implements `sayHi`, in this case a single `cout` statement.

We say that the compile-time binding occurs for `sayHi` because the compiler determines what code to be executed whenever `sayHi` is invoked.

The compiler binds any invocation of `sayHi` to `sayHi`’s entry point.
Requirements for C++ Polymorphism

- There must be an *inheritance hierarchy*.
- The classes in the hierarchy must have a *virtual method* with the same signature (virtual is a keyword).
- There must be either a *pointer* or a *reference* to a base class. The pointer or reference is used to invoke a virtual method.
Fig. 5.1.1. Run-Time Binding

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

class TradesPerson { // base class
public:
    virtual void sayHi() { cout << "Just hi " << endl; }
};
class Tinker : public TradesPerson { // derived class
public:
    virtual void sayHi() { cout << "Hi, I tinker." << endl; }
};
class Tailor : public TradesPerson { // derived class
public:
    virtual void sayHi() { cout << "Hi, I tailor." << endl; }
};

int main() {
    TradesPerson* p; // pointer to base class
    int which;
    // prompt user for a number:
    // 1 = = TradesPerson
    // 2 = = Tinker
    // 3 = = Tailor
    do {
        cout << "1 = = TradesPerson, 2 = = Tinker, 3 = = Tailor";
        cin >> which;
    } while ( which < 1 || which > 3 );
    // set pointer p depending on user choice
    switch ( which ) {
    case 1:  p = new TradesPerson; break;
    case 2:  p = new Tinker; break;
    case 3:  p = new Tailor; break;
    }
    // invoke the sayHi method via the pointer
    p->sayHi(); // *** run-time binding in effect
    delete p; // *** free the dynamically allocated storage
    return 0;
}
```
Discussions – Fig. 5.1.1.

- If the user enters 1, the address of dynamically allocated TradePerson will be assigned to p.
- If the user enters 2, the address of dynamically allocated Tinker will be assigned to p.
- If the user enters 3, the address of dynamically allocated Tailor will be assigned to p.

- Because sayHi is virtual, the system binds its invocation at run time:
  - If p points to a TradePerson, the system binds the call to TradesPerson : : sayHi.
  - If p points to a Tinker, the system binds the call to Tinker : : sayHi.
  - If p points to a Tailor, the system binds the call to Tailor : : sayHi.
TradesPerson* ptrs[10];

// randomly create TradesPersons, Tinkers, and Tailors
for (i = 0; i < 10; i++) {
    which = 1 + rand() % 3;
    switch (which) {
    case 1: ptrs[i] = new TradesPerson; break;
    case 2: ptrs[i] = new Tinker; break;
    case 3: ptrs[i] = new Tailor; break;
    }
}

A randomly generated integer determines whether we dynamically create a TradesPerson, a Tinker, or a Tailor. Next we iterate through the array, invoking each object's polymorphic sayHi method:

// polymorphically invoke the sayHi methods
for (i = 0; i < 10; i++)
    ptrs[i]->sayHi();
    delete ptrs[i];  // release the storage
Discussions

• If in the base class, the method is declared virtual, then any derived class method with the same signature is automatically virtual.

• If a virtual method is defined outside the class declaration, then the keyword virtual occurs only in the method’s declaration, not in its definition.
Example

class TradesPerson {
public:
virtual void sayHi( ) { cout << "Just hi."
<< endl; }
};
class Tinker : public TradesPerson {
public:
void sayHi( ) { cout << "Hi, I tinker."
<< endl; }
};
// ...
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

class TradesPerson { // base class 
public:
    virtual void sayHi() { cout << "Just hi. " << endl; }
};
class Tinker : public TradesPerson { // derived class 1 
public:
    virtual void sayHi() { cout << "Hi, I tinker." << endl; }
};
class Tailor : public TradesPerson { // derived class 2 
public:
    virtual void sayHi() { cout << "Hi, I tailor." << endl; }
};

int main() {
    srand ( time( 0 ) );
    TradesPerson* ptrs[ 10 ]; // pointers to base class
    unsigned which, i;

    // randomly create TradesPersons, Tinkers, and Tailors
    for( i = 0; i < 10; i++ ) {
        which = 1 + rand() % 3;
        switch ( which ) {
            case 1: ptrs[ i ] = new TradesPerson; break;
            case 2: ptrs[ i ] = new Tinker; break;
            case 3: ptrs[ i ] = new Tailor; break;
        }
    }

    // polymorphically invoke the sayHi methods
    for( i = 0; i < 10; i++ ) {
        ptrs[ i ] -> sayHi();
        delete ptrs[ i ]; // release the storage
    }
    return 0;
}
The main function contains a call to
\texttt{srand( time ( 0 ) );}
in order to avoid the generation of the same random numbers each time it runs.

The keyword \texttt{virtual} must be explicitly declared in the base class. Any derived class method with the \textit{same signature} will be automatically \texttt{virtual}. 
Example 5.1.4.

class C {
  public:
  virtual void m( ); // declaration--"virtual” occurs
   // . . .
};

void C::m( ) { // definition--"virtual" does not occur
  // . . .
}

**Discussions:** virtual method m is defined outside the class declaration. Therefore, the keyword virtual occurs only in its declaration.
EXAMPLE 5.1.5.

```c++
virtual void f(); //*** ERROR: not a method
int main() {
    // ...
}
```

**Discussions:** contains an error because it declares `f`, a top-level function rather than a method, to be virtual.
A virtual method, like a regular method, can be inherited by a derived class from a base class.

class TradesPerson {
  public:
    virtual void sayHi() { cout << "Just hi." << endl; }
};

class Tinker : public TradesPerson {
  // *** remove Tinker::sayHi
};

int main( 0 ) {
  Tinker t1;
  t1.sayHi(); // *** inherited sayHi
  // . . .
}
Run-Time Binding and the Vtable – Example 5.1.7.

class B {  // base class
    public:
        virtual void m1( ) { /* . . . */ }  // 1-st virtual method
        virtual void m2( ) { /* . . . */ }  // 2-nd virtual method
    };

class D : public B {  // derived class
    public:
        virtual void m1( ) { /* . . . */ }  // override m1
    };

int main ( ) {
    B  b1;  // base class object
    D  d1;  // derived class object
    B* p;  // pointer to base class
    // ...
    // p is set to b1's or d1's address
    p->m1();  //*** vtable lookup for run-time binding
    // ..
}
## Vtable

<table>
<thead>
<tr>
<th>Virtual Method</th>
<th>Sample Entry Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>B :: m1</td>
<td>0x7723</td>
</tr>
<tr>
<td>B :: m2</td>
<td>0x23b4</td>
</tr>
<tr>
<td>D :: m1</td>
<td>0x99a7</td>
</tr>
<tr>
<td>D :: m2</td>
<td>0x23b4</td>
</tr>
</tbody>
</table>
Discussions

There is a separate vtable entry for each virtual method in the application.

However, B : : m2 and D : : m2 have the same entry point (0 x 23b4) because derived class D does not override virtual method m2 but rather uses the version inherited from its base class B.
Constructors and Destructors

EXAMPLE 5.1.8.

class C {
public:
    virtual C( ); //***** ERROR: constructor
    virtual C( int ); //***** ERROR: constructor
    virtual ~C( ); // ok, destructor
    virtual void m( ); // ok, regular method
};

- A constructor cannot be virtual.
- A destructor can be virtual.
Virtual Destructors

Example 5.1.9.

```c++
void f() {
    A* ptr;       // pointer to base class
    ptr = new Z;  // points to derived class object
    delete ptr;  // ~A() fires but not ~Z()
}  // *** Caution: 5,000 bytes of inaccessible storage
```
Fig. 5.1.3. The Need of virtual destructors

#include <iostream>
using namespace std;

class A {  // base case
public:
    A() { cout << endl << "A() firing" << endl; }
p = new char[5];  // allocate 5 bytes
    ~A() { cout << "~A() firing" << endl; delete[] p; }  // free 5 bytes
private:
    char* p;
};

class Z : public A {
public:
    Z() { cout << "Z() firing" << endl; q = new char[5000]; }  // allocate 5,000 bytes
    ~Z() { cout << "~Z() firing" << endl; delete[] q; }  // free 5,000 bytes
private:
    char* q;
};

void f();
int main() {
    for (unsigned i = 0; i < 3; i++)
        f();
    return 0;
}

void f() {
    A* ptr;  // pointer to base class
    ptr = new Z;  // points to derived class object
    delete ptr;  // ~A() fires but not ~Z()
}  // *** Caution: 5,000 bytes of inaccessible storage
The Output of Fig. 5.1.3.

\[ A() \text{ firing} \]
\[ Z() \text{ firing} \]
\[ \sim A \text{ firing} \]
\[ A() \text{ firing} \]
\[ Z() \text{ firing} \]
\[ \sim A \text{ firing} \]
\[ A() \text{ firing} \]
\[ Z() \text{ firing} \]
\[ \sim A \text{ firing} \]
Note that only A’s destructor fires when f is invoked. Pointer ptr is of type A*, although we have it point to a derived class object:

```cpp
ptr = new Z; // points to derived class object
```

The call to new causes the constructors A() and Z() to fire.

When we invoke delete on ptr, however, only ~A() fires – despite the fact that ptr points to a Z object.

Because the destructors are not virtual, compile-time binding is in effect.
class A  {  // base case
public:
   // ...
   virtual ~A( ) {  // virtual destructor
      cout << "~A( ) firing" << endl;
      delete[ ] p;  // free 5 bytes
   }
   // ...
};
Fixing the Example 5.1.10.

By making the base class destructor \(~A()\) \textbf{virtual}, we thereby make derived class destructor \(~Z()\) \textbf{virtual} as well. For code clarity, we should put the keyword \textbf{virtual} in \(~Z()\) ‘s declaration, but \(~Z()\) is still \textbf{virtual} even if we fail to do so. With this change, the output becomes:

\begin{itemize}
\item \textbf{A( ) firing}
\item \textbf{Z( ) firing}
\item \textbf{~Z( ) firing}
\item \textbf{~A( ) firing}
\end{itemize}
Object Methods and Class Methods

EXAMPLE 5. 1. 11.

class C {
    public:
        static virtual void f();  // *** ERROR: static and virtual static
        void g();  // ok, not virtual
        virtual void h();  // ok, not static
};

⚠️ The code segment contains an error because it tries to make a method both static and virtual.
Name Overloading

Definition: Methods in the same class share the same name, but they have different formal parameter list.

Name overloading always involves compile-time binding.
EXAMPLE 5.3.1.

class C  {
public:
    C( )    /*   default constructor
    C( int x ) { /*   ….  */   }   //  constructor initializer
};

void f( double d  ) { /* …   */  }
void f( char c )   {   /*  ….  */   }

int main( )   {
    C c1;   //   default constructor called
    C c2( 26 )   //   constructor initializer called
    f( 3.14 );   //   f( double ) called
    f( 'z ' );   //   f( char ) called
    //   ...
}

In the code segment invocation of the two top-level functions f and the two constructors C involve compile-time binding.
Discussions

There are two top-level functions named f and two constructors named C.

Invocations of all four functions involve *compile-time* binding.

The compiler uses signatures to do the bindings.
Name Overriding

Suppose that the base class B has a method m and its derived class D also has a method m with the same signature.

If the methods are virtual, run-time binding is at work in any invocation of m through pointers or references.

If the methods are virtual, the derived class method D :: m overrides the base class method B :: m.

If the methods are not virtual, compile time binding is at work in any invocation of m.
EXAMPLE 5.3.2.

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

class B {  // base class
public:
    void m() { cout << "B::m" << endl; }
};

class D : public B {  // derived class
public:
    void m() { cout << "D::m" << endl; }
};

int main() {
    B* p;  // pointer to base class
    p = new D;  // create a D object
    p->m();  // invokes m
    return 0;
}
```

B :: m
Name Overriding cont’d

- Note that `p` points to a `D` object, not to a `B` object.
- Further, `p` is used to invoke `m`. The compiler uses `p`’s data type `B*` to bind the call to `B :: m`.
- Recall that `D` inherits `m` from `B`.

So `p-> m( ) ;` is equivalent to `p -> B :: m ( ) ;`
Fig. 5.3.1. Compile-Time Binding in the TradesPerson Hierarchy

```cpp
#include <iostream>
using namespace std;
class TradesPerson {   // base class
    public:
        void sayHi() {  cout << "Just hi." << endl; }
    };
class Tinker : public TradesPerson {   // derived class 1
    public:
        void sayHi() {  cout << "Hi, I tinker." << endl; }
    };
class Tailor : public TradesPerson {   // derived class 2
    public:
        void sayHi() {  cout << "Hi, I tailor." << endl; }
    };

int main( ) {
    TradesPerson* p;   // pointer to base class
    int which;
    // prompt user for a number:
    // 1 = = TradesPerson
    // 2 = = Tinker
    // 3 = = Tailor
    do {
        cout << "  1 = TradesPerson, 2 = = Tinker, 3 = = Tailor ";
        cin >> which;
        while ( which < 1 ||  which > 3 ) ;
    } // set pointer p depending on user choice
    switch ( which ) {
        case 1: p = new TradesPerson; break;
        case 2: p = new Tinker; break;
        case 3: p= new Tailor; break;
    } // invoke the sayHi method via the pointer
    p->sayHi();   // *** sayHi( ) from class TradesPerson is always invoked
    delete p;    // *** free the dynamically allocated storage
    return 0;
}
```
The user again enters an integer used in a `switch` statement to create a single object:

- a `TradesPerson` object if the user enters 1,
- a `Tinker` object if the user enters 2,
- and a `Tailor` object if the user enters 3.

Pointer `p` holds the address of the dynamically created object, and `p` is used to invoke the object's `sayHi` method.

Regardless of whether `p` points to a `TradesPerson` object, a `Tinker` object, or a `Tailor` object, `TradesPerson::sayHi` is always invoked.
Name Hiding

Suppose that base class \( B \) has a nonvirtual method \( m \) and its derived class \( D \) also has a method \( m \).

\( D \)’s local method \( D::m \) is said to hide the inherited method \( B::m \).

Name hiding is particularly tricky if the derived class’s method has a different signature than the base class’s method of the same name.
EXAMPLE 5.3.4.

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

class B {
public:
    void m( int x ) { cout << x << endl; }
};

class D : public B {
public:
    void m() { cout << "Hill << endl; }
};

int main() {
    D d1;
    d1.m(); // OK: D::m expects no arguments
    d1.m(26); // ERROR: D::m expects no arguments
    return 0;
}
```
• Example 5.3.4. generates a fatal compile-time error because it tries to invoke D's method m with a single argument. D's base class B does have a method m that expects a single argument, and D inherits this method. The problem is that D has a method with the same name. Therefore, the local D: : m hides the inherited B: : m. To invoke the inherited m with an argument, we must amend the code to

```
d1.B: :m( 26 ); // OK: explicitly call B: :m
```

• Name hiding can occur with virtual as well as nonvirtual methods. In effect, name hiding occurs with virtual methods whenever a derived class virtual method fails to override the base class virtual method.
EXAMPLE 5.3.5.

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

class B {
    public:
        virtual void m( int x ) { cout << x << endl; }
};
class D : public B {
    public:
        virtual void m( ) { cout << Hi, << endl; }
};

int main( ) {
    D d1;
    d1.m( );
    d1.m( 26 ); // ERROR: D's m takes no arguments
    return 0;
}
```
Discussions

• Example 5.3.5. generates a fatal compile-time error. D’s local method D : : m, which is virtual hides the inherited method B : : m.

• One fix is to invoke B : : m explicitly
  
  d1.B : : m ;
The examples in this section illustrate some problems that can arise if functions share a name. Nonetheless, it is sometimes desirable for functions to share a name. The obvious cases are:

- Top-level functions with overloaded names. This is a convenience to programmers, who then can use one name such as `print` to invoke many different print functions, that is, many functions whose signatures differ but whose name happens to be `print`. It is quite common to overload operators such as `< <` as top-level functions (see Chapter 6).
- Constructors with overloaded names. A class often has more than one constructor, which requires name overloading for the constructors.
- Nonconstructor methods of the same class with the same name. A class `c`, for example, might have three `print` methods for convenience. The motivation here is the same as in overloading top-level functions.
- Methods, especially `virtual` ones, in a hierarchy. For polymorphism to occur, the `virtual` methods must have the `same signature` and, therefore, the same name. In typical polymorphism, a derived class's local `virtual` method overrides a `virtual` method inherited from the base class. For overriding to occur, the methods must be `virtual` and have the `same signature`. 