Welcome
Notes
Welcome to OO
Object Terminology
Modular Programming
Types Overloading
Dynamic Memory
Member Functions
Construction
Current Object
Member Operators
Class + Resources
Helper Functions
Input Output
Derived Classes
Derived Functions
Virtual Functions
Abstract Classes
Templates
Polymorphism
I/O Refinements
D C + Resources
Standards
Bibliography
Library Functions
ASCII Sequence
Operator Precedence
C++ and C
Workshops
Assignments
Handouts
Practice
Resources

Part E - Polymorphism

Virtual Functions

Design polymorphic objects to amplify the reusability of code
Distinguish monomorphic and polymorphic objects
Describe the difference between early binding and dynamic dispatch

"Respecting the inclusion relationship implies substitutability - operations that apply to entire sets should apply to any of their subsets as well." (Sutter, Alexandrescu, 2005)

Types | Function Bindings | Polymorphic Objects | Reusability Flexibility | Summary | Exercises

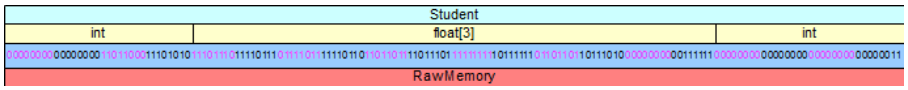
Object-oriented languages support selection of behavior across related types through polymorphism. *Polymorphism* is the third principal concept that these languages implement (alongside encapsulation and inheritance). Polymorphism refers to the multiplicity of meanings attached to a single identifier. Polymorphic stands for 'of many forms'. A polymorphic language selects an operation on an object based on the type associated with the object.

Virtual functions are an example of inclusion polymorphism. Object-oriented languages implement inclusion polymorphism through member functions in a hierarchy. The type of a polymorphic object can change throughout its lifetime to any type in the same inheritance hierarchy. We distinguish between the static and dynamic type associated with a polymorphic object. Its static type is the type of the object's hierarchy, its dynamic type is the object's actual type.

This chapter describes how C++ implements inclusion polymorphism. The chapter describes the concept of types, the options for binding a function call to its definition and how polymorphic objects are implemented in C++.

TYPES

Raw memory stores information in the form of bit strings. These bit strings represent variables, objects, addresses, instructions, constants, etc. Without knowing *what* a bit string represents, the compiler cannot interpret the bit string. By associating a type with a region of memory, we tell the compiler how to interpret the bit string in that region of memory.

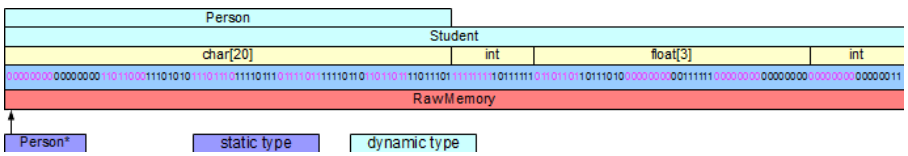


For example, if we associate a region of memory with a **`Student`** and define the structure of a **`Student`**, the compiler knows that the first 4 bytes holds an **`int`** stored in equivalent binary form, the next 12 bytes holds an array of 3 **`float`**s and the remaining 4 bytes hold an **`int`**.

C++ Pointers

C++ implements a polymorphic object using pointer syntax. The pointer type identifies the static type of the inheritance hierarchy to which the object belongs. This static type is known at compile time. The pointer holds the address of the polymorphic object.

To dereference the object's address, the compiler needs to know its dynamic type. The dynamic type is the referenced type of the object. Initially, we specify the dynamic type at object creation time through the constructor that we invoke.



In the following example, we instantiate a **`Person*`** object by dynamically allocating memory once for a **`Person`** type and one for a **`Student`** type.

```
void show(const Person*);

// Polymorphic Objects

Person  jane("Jane");
float g[] = {54.6f, 67.7f, 89.6f};
Student john("John", 1234, g, 3);

Person* pJane = &jane;
Person* pJohn = &john;

// possibly different behaviors
show(pJohn);
show(pJane);
...
```

By implementing different behaviors for different types in the same hierarchy, we enable different execution paths in **`show()`** for different dynamic types.

FUNCTION BINDINGS

The compiler binds a function call to a function definition using an object's type. The object's type determines the member function to call in the inheritance hierarchy.

The binding of a member function can take either of two forms:

- early binding - based on the object's static type
- dynamic dispatch - based on the object's dynamic type

Early Binding

Consider the following definition of our **`Student`** class from the chapter entitled Functions in a Hierarchy:

```
// Early Binding
// Student.h

#include <iostream>
const int NC = 30;
const int NG = 20;

class Person {
    char name[NC+1];
  public:
    Person();
    Person(const char*);
    void display(std::ostream&) const;
};

class Student : public Person {
    int no;
    float grade[NG];
    int ng;
  public:
    Student();
    Student(int);
    Student(const char*, int, const float*, int);
    void display(std::ostream&) const;
};
```

The implementation file is also the same as in the chapter entitled Functions in a Hierarchy:

```
// Student.cpp

#include <cstring>
#include "Student.h"
using namespace std;
```

```
Person::Person() {
    name[0] = '\0';
}

Person::Person(const char* nm) {
    strncpy(name, nm, NC);
    name[NC] = '\0';
}

void Person::display(ostream& os) const {
    os << name << ' ';
}

Student::Student() {
    no = 0;
    ng = 0;
}

Student::Student(int n) {
    float g[] = {0.0f};
    *this = Student("", n, g, 0);
}

Student::Student(const char* nm, int sn, const float* g, int ng_) : Person(nm) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        *this = Student();
    }
}

void Student::display(ostream& os) const {
    if (no > 0) {
        Person::display(os);
        os << no << ":\n";
        os.setf(ios::fixed);
        os.precision(2);
        for (int i = 0; i < ng; i++) {
            os.width(6);
            os << grade[i] << endl;
        }
        os.unsetf(ios::fixed);
        os.precision(6);
    } else {
        os << "no data available" << endl;
    }
}
```

Note that this hierarchy has two distinct definitions of the member function named **`display()`**.

The **`main()`** function listed below defines a global function named **`show()`**. This client code calls that global function twice, first for a **`Student`** object and second for a **`Person`** object. The global function **`show()`** in turn calls the **`display`** member function on **`p`**. The compiler binds the call to this member function to its **`Person`** version. C++ applies this convention irrespective of the argument type in the call to **`show()`**. That is, the compiler uses the *parameter type* in definition of **`show()`** to determine the kind of binding to implement. We call this binding an *early binding*.

The client program produces the output shown on the right:

```
// Function Bindings
// functionBindings.cpp

#include <iostream>
#include "Student.h"
```

```
void show(const Person& p) {
    p.display(std::cout);
    std::cout << std::endl;
}

int main() {
    Person jane("Jane Doe");
    float gh[] = {89.4f, 67.8f, 45.5f};
    Student harry("Harry", 1234, gh, 3);
    harry.display(std::cout);
    jane.display(std::cout);
    std::cout << std::endl;
    show(harry);
    show(jane);
}
```

```
Harry 1234:
 89.40
 67.80
 45.50
Jane Doe
Harry
Jane Doe
```

Early binding occurs at compile time and is the most efficient binding of a member function call to that function's definition. Early binding is the default in C++.

Note that shadowing does not occur inside the global function **`show()`**. **`show()`** has no way of knowing which version of **`display()`** to select aside from the type of its parameter **`p`**. The statements **`harry.display()`** and **`jane.display()`** in the **`main()`** function demonstrate shadowing. The call to **`display()`** on **`harry`** shadows the base version of **`display()`**.

Dynamic Dispatch

The output in the above example omits the details for the **`Student`** part of **`harry`**. To output these details, we need to postpone the binding of the call to **`display()`** until run-time when the executable code is aware of the dynamic type of object **`p`**. We refer to this postponement as *dynamic dispatch*.

C++ provides the keyword **`virtual`** for dynamic dispatching. If this keyword is present, the compiler inserts code that binds the call to most derived version of the member function based on the dynamic type.

For example, the keyword **`virtual`** in the following class definition instructs the compiler to postpone calling the **`display()`** member function definitions until run-time:

```
// Dynamic Dispatch
// Student.h

#include <iostream>
const int NC = 30;
const int NG = 20;

class Person {
    char name[NC+1];
  public:
    Person();
    Person(const char*);
    virtual void display(std::ostream&) const;
};

class Student : public Person {
    int no;
    float grade[NG];
    int ng;
  public:
    Student();
    Student(int);
    Student(const char*, int, const float*, int);
    void display(std::ostream&) const;
};
```

Note that the implementation file and the client program have not changed. Because the keyword is present, the compiler overrides the early binding of **`display()`** so that the **`show()`** function will call the most derived version of **`display()`** for the type of the argument passed to it. The following client code (identical to that above) then produces the output shown on the right:

```
// Function Bindings
// functionBindings.cpp
```

```
#include <iostream>
#include "Student.h"

void show(const Person& p) {
    p.display(std::cout);
    std::cout << std::endl;
}

int main() {
    Person jane("Jane Doe");
    float gh[] = {89.4f, 67.8f, 45.5f};
    Student harry("Harry", 1234, gh, 3);

    harry.display(std::cout);
    jane.display(std::cout);
    std::cout << std::endl;
    show(harry);
    show(jane);
}
```

```
Harry 1234:
 89.40
 67.80
 45.50
Jane Doe
Harry 1234:
 89.40
 67.80
 45.50

Jane Doe
```

Each call to **show()** passes a reference to an object of different dynamic type:

- **show(harry)** passes an unmodifiable reference to a **Student**
- **show(jane)** passes an unmodifiable reference to a **Person**

In each case, the executable code binds at run time the version of **display()** that is the most derived version for the dynamic type referenced by the parameter in **show()**.

Note that if we pass the argument to the **show()** function by value instead of by reference, the **show()** function would still call the most derived version of **display()**, but that most derived version would be for the **Person** version, since the copied object would be a **Person** in all cases.

Overriding Dynamic Dispatch

To override dynamic dispatch with early binding, we resolve the scope explicitly:

```
void show(const Person& p) {
    p.Person::display(std::cout);
}
```

Documentation

Some programmers include the qualifier **virtual** in derived class declarations as a form of documentation. This improves readability but has no syntactic effect.

We can identify a member function as **virtual** even if no derived class exists. This clarifies the intent of the original developer for subsequent developers of the hierarchy

POLYMORPHIC OBJECTS

A polymorphic object is an object that can change its dynamic type throughout its lifetime. Its static type identifies the hierarchy of types to which the object belongs. Its dynamic type identifies the rule for interpreting the bit string in the region of memory currently allocated for the object.

We specify the static type of a polymorphic object through

- a pointer declaration
- a receive-by-address parameter
- a receive-by-reference parameter

For example, the highlighted code specifies the static type pointed to by **person**:

```
// Polymorphic Objects - Static Type

#include <iostream>
#include "Student.h"
```

```
void show(const Person* p) {
    // ...
}

void show(const Person& p) {
    // ...
}

int main() {
    Person* p = nullptr;

    // ...

}
```

We specify the dynamic type of a polymorphic object by allocating memory dynamically using the appropriate constructor from the inheritance hierarchy.

The highlighted code in the example below identifies the dynamic type. The results produced by this code are listed on the right:

```
// Polymorphic Objects - Dynamic Type
// dyanmicType.cpp

#include <iostream>
#include "Student.h"

void show(const Person& p) {
    p.display(std::cout);
    std::cout << std::endl;
}

int main() {
    Person* p = nullptr;

    p = new Person("Jane Doe");
    show(*p);
    delete p;

    float g[] = {89.4f, 67.8f, 45.5f};
    p = new Student("Harry", 1234, g, 3);
    show(*p);
    delete p;
}
```

```
Jane Doe
Harry 1234:
 89.40
 67.80
 45.50
```

In the `main()` function:

- `p` initially points to nothing (holds the null address). The object's dynamic type is undefined.
- after the first allocation, `p` points to a `Student` type (dynamic type).
- after the second allocation, `p` points to a `Person` type (the new dynamic type).

The static and dynamic types are related to one another through the hierarchy.

Note that we only need one `show()` function to display both dynamic types.

`p` holds the address a polymorphic object throughout its lifetime. That address may change with deallocations and fresh allocations of memory. The dynamic type may be of any type in the `Person` hierarchy.

`show()` is a polymorphic function. Its parameter receives an unmodifiable reference to any type in the `Person` hierarchy.

Good Design

It is good programming practice to dynamically dispatch the destruction of any object in an inheritance hierarchy as `virtual`. If an object of a derived class acquires a resource, typically the destructor of that class releases the resource. To ensure that any object in the hierarchy calls the destructor of its most derived class at destruction time, we declare the base class destructor `virtual`. Since the destructor of any derived class automatically calls the destructor of its immediate base class, all destructors in the object's hierarchy will be called in turn.

Good design codes the destructor in a base class as `virtual`, even if no class is currently derived from that base class. The presence of a `virtual` base class destructor ensures that the most derived destructor will be called if and when a class is derived from the base class without requiring an upgrade to the definition of the base class.

REUSABILITY AND FLEXIBILITY

Implementing inclusion polymorphism improves reusability and flexibility of code.

Virtual functions reduce code size considerably. Our **`show()`** function works on references of any type within the **`Person`** hierarchy. We only define member functions (**`display()`**) for those classes that require specialized processing.

Consider a client application that uses our hierarchy. During the life cycle of the hierarchy, we may add several classes. Our original client code, without any alteration, will selects the most derived version of the member function for each upgrade of the hierarchy. We will only need to add client code to create objects of new derived classes.

SUMMARY

- polymorphism refers to the multiplicity of logic associated with the same name.
- static type is the type of the object's hierarchy and is available at compile-time
- dynamic type is the type of the object referenced and may change with different calls to the same function
- early binding of a call to a member function's definition occurs at compile-time
- the keyword virtual on a member function's declaration specifies dynamic dispatch
- a polymorphic object's pointer type identifies the object's static type
- a polymorphic object's constructor identifies the object's dynamic type
- declare a base class destructor virtual even if there are no derived classes

EXERCISES

- Complete the Handout on Virtual Functions.
- Read the notes on static and dynamic typing in Wikipedia,
- Read the notes on polymorphism in Wikipedia,

print this page

Top

Previous: Functions in a Hierarchy

Next: Abstract Base Classes

ICT | Home | Outline | Timeline

Notes | IPC Notes | MySeneca | Workshops | Assignments | Instructor