Polymorphism and Virtual Functions

Jiun-Long Huang
National Chiao Tung University

Introduction

- Polymorphism is associating many meanings to one function
  - Compile time polymorphism is supported through function overloading and operator overloading.
  - Run time polymorphism is supported through virtual function which enables programmers to design a common interface that can be used on different but related objects

Dynamic versus Static Binding

- Binding is the association between one function name and its implementation
- Static binding is performed in compile time.
  - When compiling a program, the compiler reserves a space in memory for all user defined functions and keeps track of the addresses of memory locations allocated to store each of the functions.
  - A function's name is bound with the function's address, which is the starting address of the storage space in memory reserved for the function's code.

Static Binding (Early Binding)

- The compiler binds all function calls to the addresses of the code that implement each of the functions at compile time if the function is not an inline function.
  - In the case of inline functions, the function's name is substituted with the actual function's code (not its address).

Dynamic Binding (Late Binding)

- Dynamic binding
  - Function calls are resolved at run time.
- The order of the function calls in programs that use dynamic binding depends on an action taken by the user.

Function Pointer

- A function pointer is a pointer that stores the starting address of a function's code.
  - Function pointer is used to implement dynamic binding

```
void (*fpt)(int, int)=and_gate;
(*fpt)(1,0); //Using the fpt pointer to call and_gate()
fpt=or_gate;
(*fpt)(1,1); //Using the fpt pointer to call or_gate()
```

```
#include <iostream>
using namespace std;
void func1(int x) {
  cout<<"func1 "<<x<<endl;
}
void func2(int x) {
  cout<<"func2 "<<x<<endl;
}
void func3(int x) {
  cout<<"func3 "<<x<<endl;
}
void func4(int x) {
  cout<<"func4 "<<x<<endl;
}
int main()
{
  int a;
  cin>>a;
  if (a==0)
    func1(a);
  else if (a==1)
    func2(a);
  else if (a==3)
    func3(a);
  else if (a==4)
    func4(a);
}
```

Poor performance

```
#include <iostream>
using namespace std;
void func1(int x) {
  cout<<"func1 "<<x<<endl;
}
void func2(int x) {
  cout<<"func2 "<<x<<endl;
}
void func3(int x) {
  cout<<"func3 "<<x<<endl;
}
void func4(int x) {
  cout<<"func4 "<<x<<endl;
}
int main() {
  int a;
  void (*fptr[4])(int)={func1, func2, func3, func4};
  cin>>a;
  if (a>=0 && a<=3)
    (*fptr[a])(a);
}
```

Discussions

- Dynamic binding involves more function overhead than static binding, and therefore may reduce the speed of a program.
- Dynamic binding is much more flexible than static binding and can respond to the user's events at run time.
- In most of the practical examples in which run time flexibility is a priority, the programmer would not consider the tradeoff of speed over flexibility.

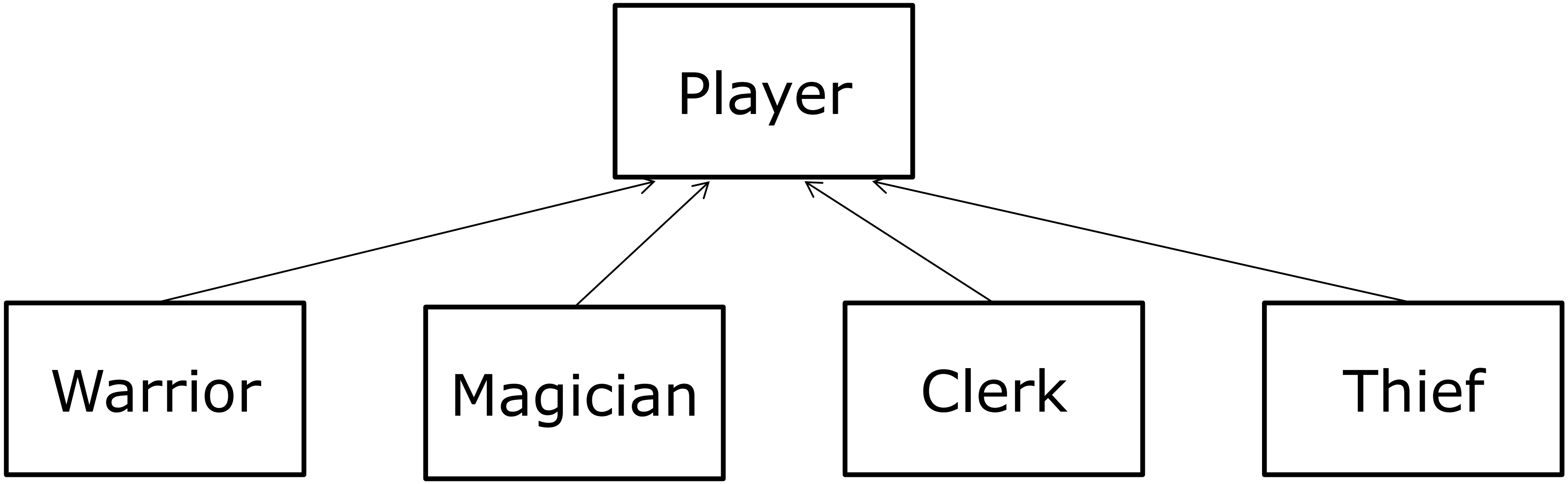
Virtual Functions

- C++ provides a tool called virtual functions to support dynamic binding and run time polymorphism.

```
virtual return_type function_name(list of paramaters);
```

Virtual Functions

- If a pointer of the base class type points to a derived class object, the virtual function declared within the derived class will be invoked by using the pointer.
- The function declared within the derived class, which has the same signature as the virtual function in the base class, is also virtual whether or not it is explicitly declared as virtual.


Player
Warrior
Magician
Clerk
Thief

```
#include <iostream>
using namespace std;
class Player {
public:
  void attack(void) { cout<<"The player punches."<<endl;}
  //virtual void attack(void) { cout<<"The player
punches."<<endl;}
};
class Warrior : public Player {
public:
  void attack(void) { cout<<"The warrior slashes with a
sword."<<endl;}
};
class Magician : public Player {
public:
  void attack(void) { cout<<"The magician attacks with a
staff."<<endl;}
};
class Clerk : public Player {
public:
  void attack(void) { cout<<"The clerk attacks with a
staff."<<endl;}
};
```

```
class Thief : public Player {
public:
  void attack(void) { cout<<"The thief stabs with a
dagger."<<endl;}
};

int main()
{
  Player p;
  Warrior w;
  Magician m;
  Clerk c;
  Thief t;
  Player * players[5];
  players[0]=&p;
  players[1]=&w;
  players[2]=&m;
  players[3]=&c;
  players[4]=&t;
  for(int i=0;i<5;i++)
    players[i]->attack();
};
```

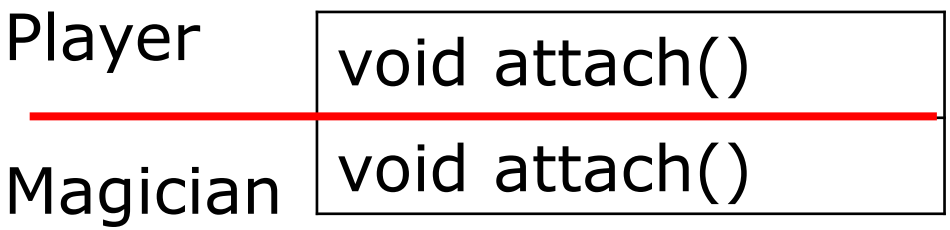
Using virtual function

```
The player punches.
The warrior slashes with a sword.
The magician attacks with a staff.
The clerk attacks with a staff.
The thief stabs with a dagger.
```

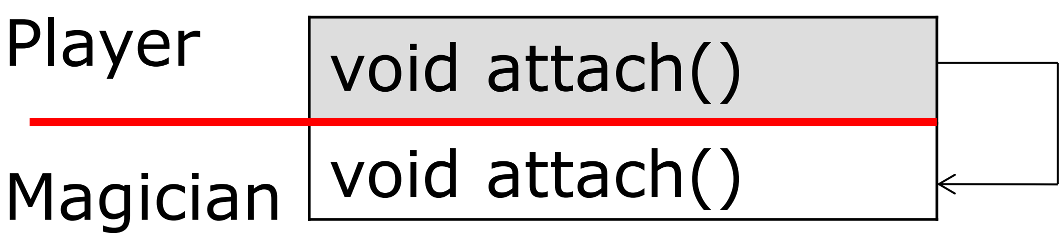
Not using virtual function

```
The player punches.
The player punches.
The player punches.
The player punches.
The player punches.
```

- Without virtual function



- With virtual function



Abstract Base Classes

- When designing an inheritance hierarchy, virtual functions should be used to define common behavior(s) (actions) of the classes that form the hierarchy.
- The definition of a common behavior, in the form of a virtual function, begins with a base class at the top of the hierarchy.

- The virtual function is then redefined (overridden) at every level of derived classes to describe a specific behavior of each class.
- If a derived class does not redefine the virtual function, then the behavior (function) of its base class is inherited.

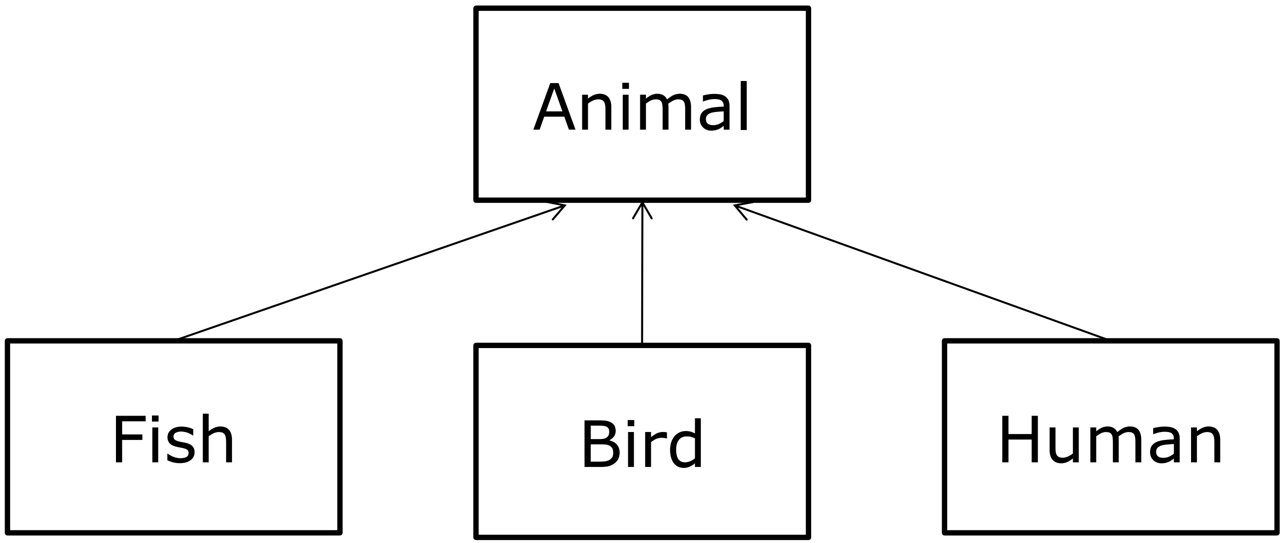
- An abstract class is a class that will never instantiated.
- When designing abstract base classes, pure virtual functions should be used whenever there is no need to define these functions within the base class.
  - A pure virtual function is a virtual function that does not have a definition (code) in its class.

Pure Virtual Function

- To declare a pure virtual function, the =0; initializer must substitute the body of the function.
  - This initializer specifies that the function has no body (no definition).
  - The class becomes an abstract base class.
  - The derived classes are forced to override pure virtual functions

```
virtual return_type function_name(list of paramaters)=0;
```

- An object cannot be instantiated from an abstract class due to the incomplete class definition resulting from the missing code in the pure virtual function(s).
- The opposite of an abstract class is a concrete class, from which objects can be instantiated.
  - A concrete class does not contain any member function declared as a pure virtual function.


Animal
Fish
Bird
Human

```
#include <iostream>
using namespace std;
class Animal { // abstract class
  public:
    virtual void move(void)=0; // pure virtual function
};
class Fish : public Animal { // concrete class
  public:
    void move(void) { cout<<"The fish swims."<<endl;}
};
class Bird : public Animal {
  public:
    void move(void) { cout<<"The bird flies."<<endl;}
};
```

```
class Human : public Animal {
  public:
    void move(void) { cout<<"Human walks."<<endl;}
};
int main()
{
  Fish f;
  Bird b;
  Human h;
  Animal * animals[3];
  animals[0]=&f;
  animals[1]=&b;
  animals[2]=&h;
  for(int i=0;i<3;i++)
    animals[i]->move();
};
```

Virtual Destructors

- Constructors could not be declared as virtual functions because of the following reasons:
  - Constructors cannot be inherited.
  - Constructors' names have to match the names of their corresponding classes.

Virtual Destructors

- Destructors can be declared virtual.
  - Constructors cannot be virtual
- It is sometimes necessary to create virtual destructors in order to prevent some problems that occur especially when attributes of derived classes are dynamically allocated.

```
virtual ~class_name()
{
   //body of destructor
}
```

Virtual Destructor

- C++ decides which class destructor to invoke by checking a pointer type, not the type of an object to which the pointer points.
  - This can cause a variety of problems such as memory leak.
- To prevent these problems, a polymorphic class should have a virtual destructor, even if the class does not require an explicit destructor.

```
#include <iostream>
using namespace std;
class Animal {
  public:
    virtual ~Animal()
        {cout << "Destroying the animal."<<endl; };
    //~Animal() { cout << "Destroying the animal."<<endl;}
};
class Fish : public Animal {
  private:
    char * name;
  public:
    Fish() {name=new char[10];}
    ~Fish() {
        delete[] name;
        cout <<"Destroying the fish."<<endl; }
};
```

```
class Bird : public Animal {
  private:
    char * name;
  public:
    Bird() { name=new char[10]; }
    ~Bird() {
      delete[] name;
      cout << "Destroying the bird."<<endl; }
};
int main()
{
  Fish * f=new Fish();
  Bird * b=new Bird();
  Animal * animals[2];
  animals[0]=f;
  animals[1]=b;
  for(int i=0;i<2;i++)
    delete animals[i];
};
```

Without virtual destructor

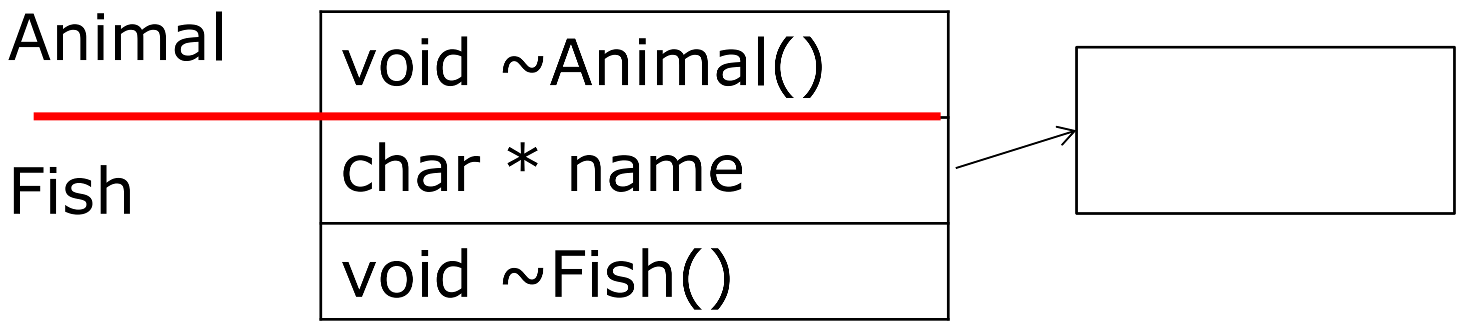
```
Destroying the animal.
Destroying the animal.
```

Memory leak!

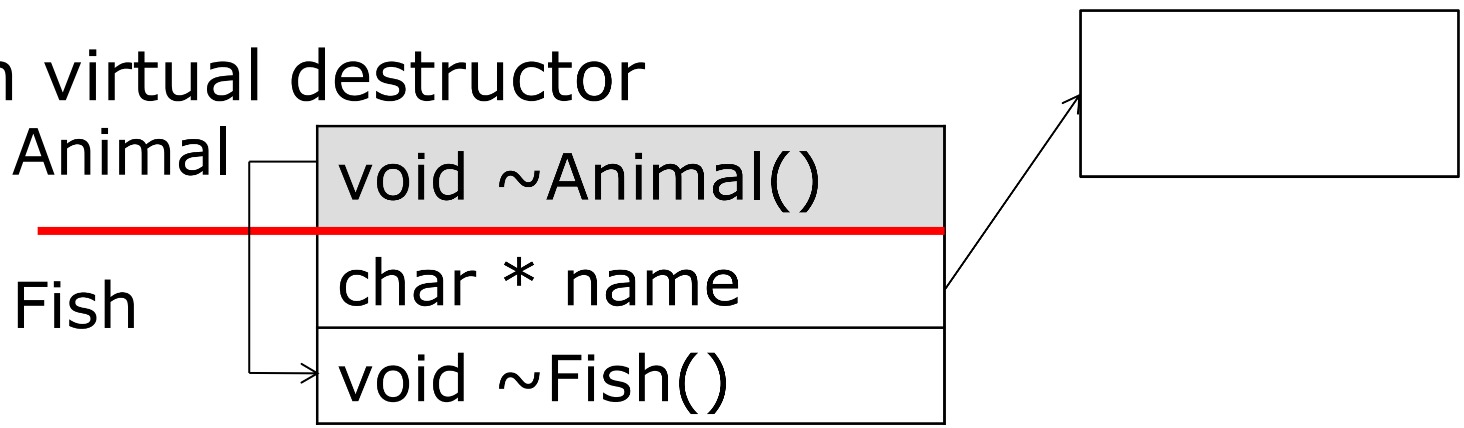
With virtual destructor

```
Destroying the fish.
Destroying the animal.
Destroying the bird.
Destroying the animal.
```

- Without virtual destructor



- With virtual destructor



Using Polymorphism

- Virtual functions (including virtual destructors) and abstract base classes are fundamental tools in the implementation of run time polymorphism (dynamic binding).
- Polymorphism enables programmers to use the same interface (functions) on different types of objects, thus reducing program development time.

```
class TV {
public:
  virtual TV()=0;
  virtual void powerOn()=0;
  virtual void powerOff()=0;
  virtual void changeVolume(int)=0;
  virtual void changeChannel(int)=0;
};
```

Interface

```
class BrandX : public TV {
public:
  ~BrandX();
  void powerOn();
  void powerOff();
  void changeVolume(int);
  void changeChannel(int);
};
```

```
class BrandY : public TV {
public:
  BrandY();
  void powerOn();
  void powerOff();
  void changeVolume(int);
  void changeChannel(int);
};
```