

C++ Revision

—

Contents

- Pointer
- Inheritance
- Virtual function
- Abstract class
- Interface class



Pointer


Pointer

- A variable that stores the memory address as its value.
- Is created with the * operator

```
int a = 5;
```

```
int* ptr = &a; // Assign the address of a to the pointer
```

Address-of operator



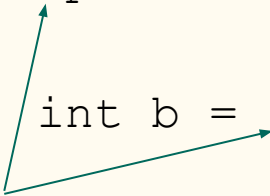
a			
	5		
3244	3245	3246	

ptr			
	3245		

Pointer

- Pointers are said to "point to" the variable whose address they store.
- Pointers can be used to directly access the variable they point to using the dereference operator (*).

```
int a = 5;  
int* ptr = &a; // ptr points to a  
*ptr = 10;      // a's value will now be 10  
int b = *ptr;    // A new variable b will be created.  
                // a's value will be copied to b
```



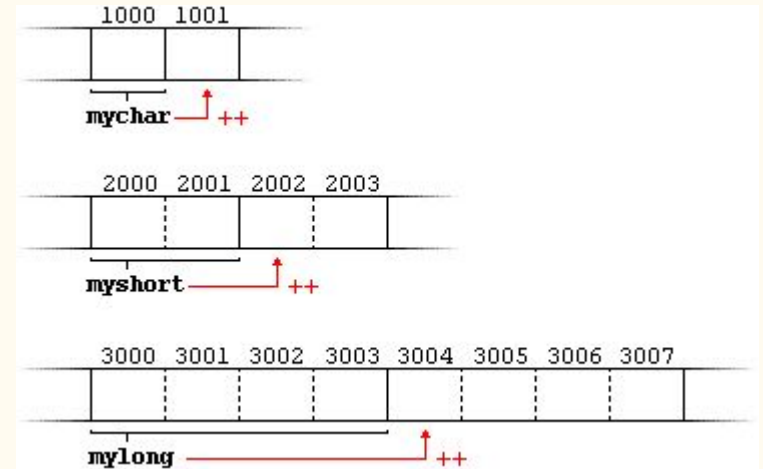
Dereference operator

Pointer

Pointers arithmetics

Addition and subtraction operations are allowed on pointers.

```
char *mychar;  
short *myshort;  
long *mylong;  
  
// ++ will move the pointer to the next  
// x byte(s), x being the size of the type  
++mychar; // Moves to the next byte  
++myshort;  
++mylong;
```



Pointer

Pointers and arrays

- Arrays work very much like pointers to their first elements.
- An array can always be implicitly converted to the pointer of the proper type.

```
int arr[10];  
int* ptr = arr;  
*ptr++ = 5;  
*(ptr+3) = 6;  
arr[5] = 9;  
  
std::cout << "arr contains "  
for (int ele : arr) {  
    std::cout << ele << ", " ;  
}  
std::cout << std::endl;
```

Pointer

Pointers and dynamic memory allocation

Dynamic memory is allocated using operator `new` followed by a data type specifier.

```
int* ptr = new int(2);
```

It creates and initializes objects with dynamic storage duration, i.e., objects whose lifetime is not limited by the scope in which they were created.

It returns a pointer to the beginning of the new block of memory allocated.

Pointer

Pointers and dynamic memory allocation

- Once the memory allocated using operator `new` is no longer needed, it can be freed using operator `delete` so that the memory becomes available.

```
delete ptr;
```

Memory leaks

- If the memory allocated using operator `new` is not deleted, and the original value of pointer is lost, then the memory cannot be deallocated: a memory leak occurs.

Pointer

Memory leak example

```
double square(double num) {  
    double* ptr = new double(num);  
  
    return (*ptr) * (*ptr); // Return without deallocating ptr  
}  
  
int main() {  
    double s = square(100.);  
}
```

Pointer initialization

```
int var = 5;

int* ptr = &var;

int* ptr1;    // It is an uninitialized pointer
ptr1 = &var;  // Now it is initialized to point to var

int* ptr2 = nullptr;    // A null pointer points to nowhere

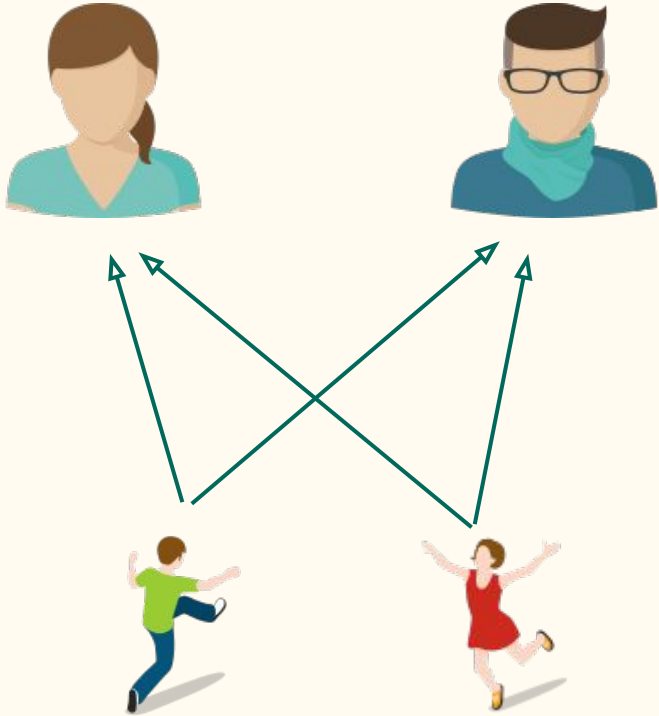
int* ptr3 = 0;          // Null pointer.

int* ptr4 = new int(var); // Dynamically allocated pointer
```

Inheritance

—

Inheritance



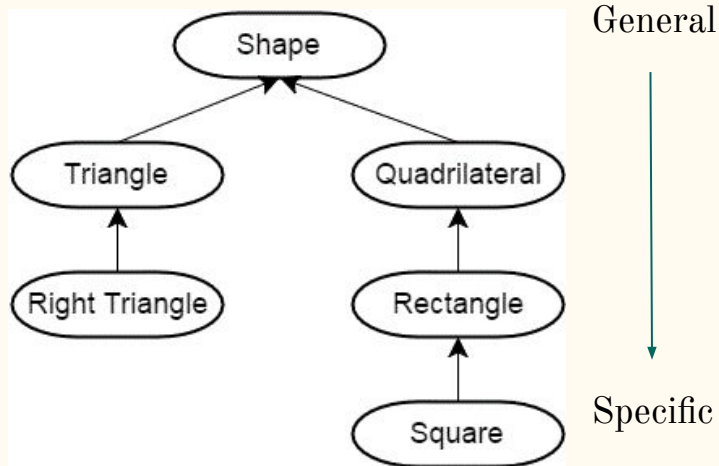
In biology, inheritance is the passing on of traits from parents to their offspring.

Similar notion in programming.

In programming, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.

Inheritance

- Models an “is-a” relationship between objects.
- Involves creating new objects by directly acquiring the attributes and behaviors of other objects and then extending or specializing them.



This diagram goes from general (top) to specific (bottom), with each item in the hierarchy inheriting the properties and behaviors of the item above it.

Inheritance

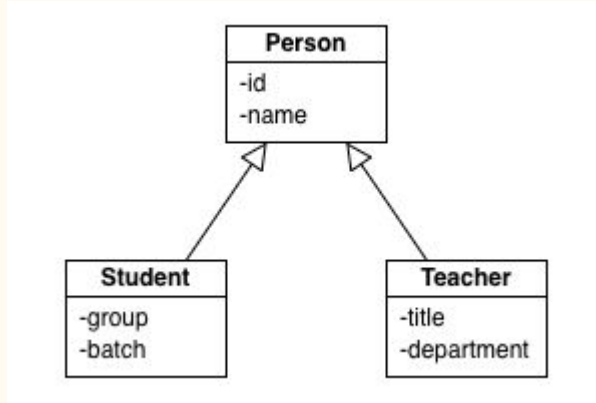
- Inheritance in C++ takes place between classes.
- In an inheritance (is-a) relationship, the class being inherited from is called the **parent class**, **base class**, or **superclass**, and the class doing the inheriting is called the **child class**, **derived class**, or **subclass**.
- A child class inherits both behaviors (member functions) and properties (member variables) from the parent (with different access permission).
- These variables and functions become members of the derived class.
- Child classes can have their own members that are specific to that class (specialization).

Inheritance

When you derive a class from another class, the new class gets all the functionality of the base class plus whatever new features you add.

You can add data members and functions to the new class but you cannot remove anything from what the base class offers.

Inheritance



Person is a base class.

Person is inherited (or extended) by child classes, Student and Teacher.

In UML class diagrams, inheritance (also called generalization) is indicated by a triangular arrowhead on the line connecting the parent and child classes.

Inheritance

Its big payoff is that it permits code reusability.

A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

Example

Access specifier

```
// Base class
class Person {
    public:
        long id;
        std::string name;

        std::string getName() const
        { return name; }
};
```

Access level

```
// Derived class
class Student : public Person {
    public:
        std::string group;
        std::string batch;

        std::string getGroup()
const
        { return group; }
};
```

Inheritance in C++

Syntax

```
class DerivedClassName : access-level BaseClassName
```

`access-level` (aka visibility mode, derivation type) specifies the type of derivation/inheritance, and it can be

- `private` (by default)
- `protected` or
- `public`

Polymorphism:
Virtual function, abstract
class and interface class

—

Virtual function

Virtual means existing in appearance but not in reality.

When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class.

A virtual function is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class.

To make a function virtual, we simply place the “`virtual`” keyword before the function declaration.

Virtual Function: Example

```
class Polygon {  
public:  
    virtual double area() {  
        return 0;  
    }  
};
```

```
class Triangle : public Polygon {  
protected:  
    double base, height;  
public:  
    Triangle(double base = 0, double height =  
0)  
        : base(base), height(height) {}  
  
    virtual double area() {  
        return height * base / 2;  
    }  
};
```

```
class Rectangle : public Polygon {  
protected:  
    double length, width;  
public:  
    Rectangle(double length = 0, double width =  
0)  
        : length(length), width(width) {}  
  
    virtual double area() {  
        return length * width;  
    }  
};
```

Why virtual function?

Suppose you have **a number of objects of different classes** but you want to put them all in an array and **perform a particular operation on them using the same function call**.

For example, suppose you have Rectangle objects and Triangle objects in an array and you want to get their area.

With the help of virtual functions, you will be able to do the following

```
Polygon* polygons[n];  
// Initialize the objects  
// ...  
for (Polygon *p : polygons) {  
    std::cout << "The area is " << p->area() << std::endl;  
}
```


Pure virtual functions

- A pure virtual function (or abstract function) is a special kind of virtual function that has no body at all!
- A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.
- To create a pure virtual function, rather than define a body for the function, we simply assign the function the value 0.

```
class Polygon
{
public:
    Polygon() {}
    virtual double area() = 0;
};
```

Pure virtual functions

A pure virtual function indicates that “it is up to the derived classes to implement this function”.

A pure virtual function is useful when

- we have a function that we want to put in the base class, but only the derived classes know what it should return.
- we want our base class to provide a default implementation for a function, but still force any derived classes to provide their own implementation.

Pure virtual functions, Abstract base classes

Using a pure virtual function has two main consequences:

1. Any class with one or more pure virtual functions becomes an **abstract base class**, which means that it **cannot be instantiated!**
2. Any derived class must define a body for this function, or that derived class will be considered an abstract base class as well.

Interface class

An interface class is a class that has no member variables, and where all of the functions are pure virtual.

Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

Interface class: Example

Suppose we want to implement a Last-In-First-Out (LIFO) list, called Stack, with three functions: push (adding an element), pop (removing the last inserted element, and top (peeking the last inserted element without removing it).

It can be implemented by storing the data contiguously (in an array) or non-contiguously (in a linked list).

We want the client code to be least affected when we change the details of the implementation (i.e., changing the contiguous storage to non-contiguous one).

Interface class: Example

In this case, we can define an interface class that defines the behaviours / functionalities of the LIFO list.

```
class Stack {  
public:  
    virtual ~Stack() {}  
  
    virtual bool push(const int element) = 0;  
    virtual bool pop(int &element) = 0;  
    virtual bool top(int &element) const = 0;  
};
```

Interface class: Example

The details are then implemented in a derived class.

```
class ArrayStack : public Stack
{
private:
    int *data;
    int topIndex;
    int size;
public:
    ArrayStack(int size);

    virtual bool push(const int element);
    virtual bool pop(int &element);
    virtual bool top(int &element) const;
};
```

```
ArrayStack::ArrayStack(int size)
    : size(size), topIndex(-1),
      data(new int[size]) {}

bool ArrayStack::push(const int element){
    if (topIndex < size - 1){
        topIndex++;
        data[topIndex] = element;
        return true;
    } else {
        return false;
    }
}
```

Interface class: Example

```
bool ArrayStack::top(int &element) const{
    if (topIndex < 0) {
        return false;
    } else {
        element = data[topIndex];
        return true;
    }
}

bool ArrayStack::pop(int &element) {
    if (top(element)) {
        topIndex--;
        return true;
    } else {
        return false;
    }
}
```

```
int main()
{
    Stack *s = new ArrayStack(10);
    s->push(10);
    s->push(9);

    int element;

    s->top(element);
    std::cout << "Top element is " << element;

    s->pop(element);
    std::cout << "Popped element is "
               << element;

}
```