Advanced C++ programming

Inheritance

Giuseppe Lipari - Luigi Palopoli

CRIStAL - Université de Lille
Embedded Intelligence and Robotic Systems - Università di Trento

Outline

Outline

Standard Template Library

Inheritance

Virtual functions

Abstract classes

STL

- The Standard Template Library is normally distributed with the compiler
- It contains generic code (templates), including:
  - containers (vector, list, deque, map, set)
  - algorithms (sort, foreach, etc.)
  - I/O streams (cout, cin, fstreams, etc.)
  - strings
  - . . . and much more
- All classes defined in the STL are in the namespace `std`

Vector

- Vector is the generalisation of the C array

an array of integers

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

Vector

- Vector is the generalisation of the C array

an array of integers

a vector of strings

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

prints 1

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

prints 1

prints the first element

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

prints 1

prints the first element

out of range: undefined behaviour

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

prints 1

prints the first element

out of range: undefined behaviour

out of range: raises exception

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

prints 1

prints the first element

out of range: undefined behaviour

out of range: raises exception

Vector of objects

- Vector requires the following basic properties of the template class
  - Copy constructor; (otherwise you cannot insert elements)
  - Assignment operator; (otherwise you cannot return an object)
- It is possible to pre-allocate space for the vector:
  - This is used to avoid excessive allocation overhead when we have an idea of the size we need

```
vector<MyClass> v(10); // reserves 10 elements
```

Iterators

- Iterators are a generic way to access elements in a container, according to a predefined order
- The iterator is usually a class provided by the container itself
- It can be seen as a *pointer* to the elements of the container
  - `begin()` returns an iterator to the first element
  - `end()` returns the iterator pointing *beyond* the last element of the array
  - it is possible to use `++` and `--` to increment/decrement the iterator (i.e. move to the next/previous element)
  - it is possible to access the *pointed element* by using the dereferencing `operator*()`

```
vector<int> v;
vector<int>::iterator i;

for (i = v.begin(); i != v.end(); i++) cout << *i;
```

Iterators

```
int a[4] = {2, 4, 6, 8};
vector<int> v = {2, 4, 6, 8};

// visit the container with indexes
for (int i=0; i<4; i++) cout << a[i];
cout << endl;
for (int i=0; i<4; i++) cout << v[i];
cout << endl;

// visit the container with pointers/iterators
for (int *p=a; p!=&a[4]; p++) cout << *p;
cout << endl;
for (vector<int>::iterator q=v.begin();
  q != v.end(); q++) cout << *q;
cout << endl;

 // visit with iterators
vector<int>::iterator q = v.end();
do {
  q--; cout << *q;
} while (q != v.begin());
cout << endl;
```

Why iterators ?

- Iterators are available for all containers in the standard library, with the same uniform interface
- They represent a simple and uniform way to visit a container
- Many template functions and member functions accept iterators parameters
- see `examples/iterator-example2.cpp`

Vector internal implementation

- The vector is internally implemented as a variable size array
  - Therefore, internally it allocates and deallocates memory depending on the current number of elements
  - However, all elements are stored sequentially in memory
  - Therefore, when you perform an insertion in the middle using function member `insert()`, it simply moves all elements one step ahead to make space for the additional element to be inserted in the right place
  - Similarly, a `push_back()` may imply a copy of all elements!
  - Therefore, insertion in a vector is a costly operation which potentially takes $O(n)$.

Auto

- the keyword `auto` tells the compiler to automatically deduce the type:

```
char *array[10] = "My Array";

auto p = begin(array);
```

in this case, the type of `p` is `char *`, because an iterator to a standard array is a pointer to the elements of the array

```
vector<int> v = {1, 3, 5, 7, 11, 13, 17, 19};

auto p = begin(v);
```

in this case, the type of `p` is `vector<int>::iterator`.

- We will analyse the rules for deducing the type later on
- For the moment, let's observe that the use of `auto` and `begin()` reduces the number of characters to type, and makes the code more generic

Range-based for loops

- Sometimes a for loop over a container can be boring to write. C++11 provides a convenient syntax:

```
vector<int> vec;
...
for (int i : vec)
        cout << i << ", ";
```

- Notice that `i` is not an iterator here, but the actual variable that holds the value of the elements inside the vector
- Of course, if the content has a strange type, you can also use auto:

```
vector<MyClass *> v;
...
for (auto p : v)
        cout << p->getX() << ", " << p->getY() << endl;
```

Range-based for loops and references

- If you want to modify the content of the container, use references

```
vector<MyClass> v;
...
for (auto &obj : v) obj.set(value);
```

- You can use range-based loops over any container that provides:
  - `begin()` and `end()` functions that return an iterator
  - The iterator must support pre-increment (`operator++()`), dereferencing (`operator*()`) and inequality (`operator!=()`)
- Therefore, you can write your own container that provides such functions, and use the new range-based loop with your container

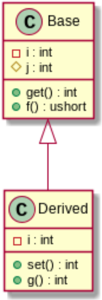
Outline

Code reuse

- In C++ (like in all OO programming), one of the goals is to re-use existing code
- There are two ways of accomplishing this goal: composition and inheritance
- Composition
  - it consists of using an object with its interface inside another object
- Inheritance
  - Inheritance consists in enhancing an existing class with new, more specific code
- Most of the times, the two mechanism must work together

Inheritance

- Class Diagram




```
class Base {
  int i;
protected:
  int j;
public:
  Base() : i(0),j(0) {};
  ~Base() {};
  int get() const {return i;}
  int f() const {return j;}
};

class Derived : public Base {
  int i;
public:
  Derived() : Base(),  i(0) {};
  ~Derived() {};
  void set(int a) {j  = a; i+= j}
  int g() const {return i;}
};
```

Syntax

```
class Derived : public Base {
  int i;
public:
  Derived() : Base(),
              i(0)
  {}

  ~Derived() {}
  void set(int a) {
    j = a;
    i+= j;
  }
  int g() const {
    return i;
  }
};
```

class Derived derives publicly from Base

Syntax

```
class Derived : public Base {
  int i;
public:
  Derived() : Base(),
              i(0)
  {}

  ~Derived() {}
  void set(int a) {
    j = a;
    i+= j;
  }
  int g() const {
    return i;
  }
};
```

class Derived derives publicly from Base

Therefore, to construct Derived, we must first construct Base

Syntax

```
class Derived : public Base {
  int i;
public:
  Derived() : Base(),
              i(0)
  {}

  ~Derived() {}
  void set(int a) {
    j = a;
    i+= j;
  }
  int g() const {
    return i;
  }
};
```

class Derived derives publicly from Base

Therefore, to construct Derived, we must first construct Base

j is a member of Base declared as protected; therefore, Derived can access it

Syntax

```
class Derived : public Base {
  int i;
public:
  Derived() : Base(),
              i(0)
  {}

  ~Derived() {}
  void set(int a) {
    j = a;
    i+= j;
  }
  int g() const {
    return i;
  }
};
```

class Derived derives publicly from Base

Therefore, to construct Derived, we must first construct Base

j is a member of Base declared as protected; therefore, Derived can access it

i is a member of Derived. There is another i that is a private member of Base, so it cannot be accessed from Derived

Use of Inheritance

- Now we can use `Derived` as a special version of `Base`

```
int main()
{
  Derived b;
  cout << b.get() << endl; // calls A::get();
  b.set(10);
  cout << b.g() << endl;
  b.g();
  Base *a = &b;  // Automatic type conversion
  a->f();
  Derived *p = new Base; // error!
}
```

- See `examples/example1.cpp`

Public Inheritance

- Public inheritance means that the derived class *inherits* the same interface of the base class
  - All members in the `public` part of `Base` are also part of the `public` part of `Derived`
- All members in the `protected` part of `Base` are part of the `protected` part of `Derived`
- All members in the private part of `Base` are not accessible from `Derived`.
- This means that if we have an object of type `Derived`, we can use all functions defined in the `public` part of `Derived` and all functions defined in the `public` part of `Base`.

Overloading and hiding

- There is no overloading across classes

```
class A {
  ...
public:
  int f(int, double);
};

class B : public A {
  ...
public:
  void f(double);
}
```

```
int main()
{
  B b;
  b.f(2,3.0);      // ERROR!
}
```

- `A::f()` has been hidden by `B::f()`
- to get `A::f()` into scope, you can use the `using` directive:

```
using A::f(int, double);
```

Upcasting

- It is possible to use an object of the derived class through a pointer to the base class.

```
class A {
public:
  void f() { ... }
};
class B : public A {
public:
  void g() { ... }
};

A* p;
p = new B();
p->f();
p->g();
```

A pointer to the base class

Upcasting

- It is possible to use an object of the derived class through a pointer to the base class.

```
class A {
public:
  void f() { ... }
};
class B : public A {
public:
  void g() { ... }
};

A* p;
p = new B();
p->f();
p->g();
```

A pointer to the base class

The pointer now points to an object of a derived class

Upcasting

- It is possible to use an object of the derived class through a pointer to the base class.

```
class A {
public:
  void f() { ... }
};
class B : public A {
public:
  void g() { ... }
};

A* p;
p = new B();
p->f();
p->g();
```

A pointer to the base class

The pointer now points to an object of a derived class

Call a function of the interface of the base class: correct

Upcasting

- It is possible to use an object of the derived class through a pointer to the base class.

```
class A {
public:
  void f() { ... }
};
class B : public A {
public:
  void g() { ... }
};

A* p;
p = new B();
p->f();
p->g();
```

A pointer to the base class

The pointer now points to an object of a derived class

Call a function of the interface of the base class: correct

Error! g() is not in the interface of the base class, so it cannot be called through a pointer to the base class!

References

- Same thing is possible with references

```
class A {
public:
  void f() { ... }
};
class B : public A {
public:
  void g() { ... }
};

void h(A &x)
{
  x.f();
  x.g();
}

B obj;
h(obj);
```

Function h takes a reference to the base class

References

- Same thing is possible with references

```
class A {
public:
  void f() { ... }
};
class B : public A {
public:
  void g() { ... }
};

void h(A &x)
{
  x.f();
  x.g();
}

B obj;
h(obj);
```

Function h takes a reference to the base class

Of course, it is possible to call functions in the interface of the base class

References

- Same thing is possible with references

```
class A {
public:
  void f() { ... }
};
class B : public A {
public:
  void g() { ... }
};

void h(A &x)
{
  x.f();
  x.g();
}

B obj;
h(obj);
```

Function h takes a reference to the base class

Of course, it is possible to call functions in the interface of the base class

This is an error! `g()` is not in the interface of `A`

References

- Same thing is possible with references

```
class A {
public:
  void f() { ... }
};
class B : public A {
public:
  void g() { ... }
};

void h(A &x)
{
  x.f();
  x.g();
}

B obj;
h(obj);
```

Function h takes a reference to the base class

Of course, it is possible to call functions in the interface of the base class

This is an error! `g()` is not in the interface of `A`

Calling the function by passing a reference to an object of a derived class: correct.

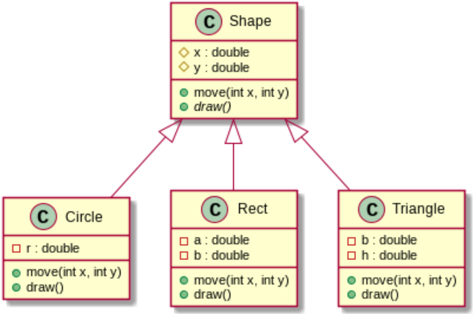
Extension through inheritance

- Why this is useful?
  - All functions that take a reference (or a pointer) to `A` as a parameter, continue to be valid and work correctly when we pass a reference (or a pointer) to `B`
  - This means that we can reuse all code that has been written for `A`, also for `B`
  - In addition, we can write additional code specifically for `B`
  - However, notice that, until now, to access a function of class `B`, we need a pointer or a reference to class `B`.
- We need also to modify (customize, extend, etc.) the behaviour of existing code
  - we need to call a function of class `B` through a pointer to the base class `A`.

Outline

Virtual functions

- Let's introduce virtual functions with an example:



Implementation

```
class Shape {
protected:
  double x,y;
public:
  Shape(double x1, double y2);
  void move(int x, int y);
  virtual void draw() = 0;
};

class Circle : public Shape {
  double r;
public:
  Circle(double x1, double y1,
         double r);
  virtual void draw();
};
```

```
class Rect : public Shape {
  double a, b;
public:
  Rect(double x1, double y1,
       double a1, double b1);
  virtual void draw();
};

class Triangle : public Shape {
  double a, b;
public:
  Triangle(double x1, double y1,
           double a1, double b1);
  virtual void draw();
};
```

Collecting Shapes

Let's make an array of Shapes:

```
vector<Shapes *> shapes;

shapes.push_back(new Circle(2,3,10));
shapes.push_back(new Rect(10,10,5,4));
shapes.push_back(new Triangle(0,0,3,2));

for (int i=0; i<3; i++) {
  shapes[i]->move(i, i);
  shapes[i]->draw();
}
```

- Which method `draw` is called ?
  - how does the compiler knows which function should be called?

Virtual vs. regular methods

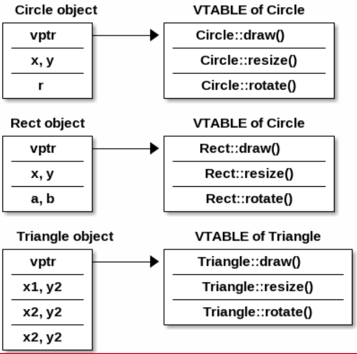
```
class Shape {
protected:
  double x,y;
public:
  Shape(double xx, double yy);
  void move(double x, double y);
  virtual void draw();
  virtual void resize(double scale);
  virtual void rotate(double degree);
};

class Circle : public Shape {
  double r;
public:
  Circle(double x, double y,
                  double r);
  void draw();
  void resize(double scale);
  void rotate(double degree);
```

- `move()` is a regular method
- `draw()`, `resize()` and `rotate()` are virtual.
- see `examples/shape.hpp`

Virtual table

- When you put the `virtual` keyword in front of at least one method of the class:
  1. The compiler builds a vtable for the class, that contains pointers to the virtual methods of the class ;
  2. Each object of the class contains a hidden field, called vptr that points to the corresponding vtable



Calling a virtual function

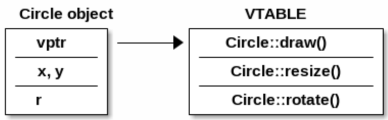
- When compiling the code
  - For each class $\rightarrow$one VTABLE
  - For each object $\rightarrow$one VPTR (first element of the object in memory)
- When the compiler sees a call to a virtual function, it performs a late binding, or dynamic binding
  - gets the vtable from the vptr ;
  - move to the right position into the vtable (depending on which virtual function we are calling)
  - call the function

Example of dynamic binding

vptr = first element at address shapes[4]

```
Shape *shapes[10];
// init

shapes[4]->resize();
```



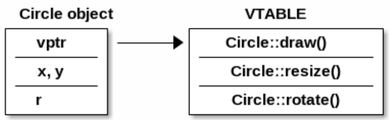
Example of dynamic binding

vptr = first element at address shapes[4]

vptr[0] points at draw, vptr[1] points at resize, ...

```
Shape *shapes[10];
// init

shapes[4]->resize();
```



Example of dynamic binding

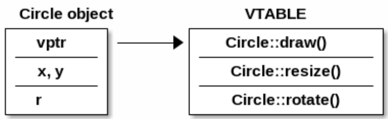
```
Shape *shapes[10];
// init

shapes[4]->resize();
```

vptr = first element at address shapes[4]

vptr[0] points at draw, vptr[1] points at resize, ...

call function at vptr[1], by passing shapes[4] as **this**



Difference with Java

Java

All methods are implicitely virtual. We look at the type of the object:

```
Base obj = new Derived();

obj.method(); // the Derived method is called, always
```

C++

Depends on the method signature

- for virtual methods, dynamic binding is performed
- for non-virtual methods, static binding is performed

```
Base *p = new Derived();

p->v_method();  // the Derived method is called
```

Dynamic vs static binding

- Which functions are called in the following code?

```
class Base {
public:
  void f() { cout << "Base::f()" << endl; g(); }
  virtual void g() { cout << "Base::g()" << endl; }
};
class Der : public Base {
public:
  void f() { cout << "Der::f()" << endl; g(); }
  virtual void g() { cout << "Der::g()" << endl; }
};
...

Base *p = new Der{};
p->g();
p->f();

Der  b{};
Base &r = b;
```

Overloading and Overriding

- The virtual function in all derived class must have exactly the same prototype as the virtual function in the base class
  - otherwise it is a different function
- In particular, the return type must be the same

- There is only one exception to this rule:
  - if the base class virtual method returns a pointer or a reference to an object of the base class . . .
  - . . . the derived class can change the return value to a pointer or reference of the derived class

Overload and Override

Correct

```
class A {
public:
  virtual A& f();
  int g();
};

class B: public A {
public:
  virtual B& f();
  double g();
};
```

Wrong

```
class A {
public:
  virtual A& f();
};

class C: public A {
public:
  virtual int f();
};
```

Virtual destructors

- What happens if we try to destruct an object through a pointer to the base class?

```
class A {
public:
  A();
  ~A();
};

class B : public A {
public:
  B();
  ~B();
};

int main() {
  A *p;
  p = new B;
  // ...
```

Big mistake!

- The destructor of the base class is called, which *destroys* only part of the object
- Soon a segmentation fault...
- Solution: declare the destructor as virtual

Restrictions

- Calling a virtual function from constructor/destructor is not a good idea
  - In Java, this is an ERROR, and should never be done
- In C++:
  - if you call a virtual function in the constructor of the base class, the base class method is called, even if you are constructing an object of a derived class
  - if you call a virtual function in the destructor of the base class, the base class method is called, even if you are destroying an object of a derived class
- So, unlike Java, C++ is safe, however this causes confusion in programmers, and should be avoided
- See C++FaqLite and Stackoverflow

Example

examples/virt_in_constr.cpp

Outline

Pure virtual functions

- A virtual function is pure if no implementation is provided

```
class Abs {
public:
  virtual int fun() = 0;
  virtual ~Abs();
};
class Derived public Abs {
public:
  Derived();
  virtual int fun();
  virtual ~Derived();
};
```

This is a pure virtual function. No object of Abs can be instantiated.

One of the derived classes must *finalize* the function to be able to instantiate the object.

Interface classes

- An abstract class is a class than contains a pure virtual method
  - cannot be instantiated
- An interface class is an abstract class that contains only pure virtual methods
  - Unlike Java, there is no special keyword to denote an interface