سلسلة تعلم البرمجة بلغة ++C الحديثة

Learn Modern C++ Programming Course

إعداد المهندس أحمد الديب

#32: Class Basics 2

[static] Members

```
class Date {
 public:
  Date(int dd = 0, int mm = 0, int yy = 0);
  static void set_default(int dd, int mm, int yy);
  void print() const;


 private:
  int d, m, y;
  static Date default_date;
};


Date Date::default_date{1, 1, 1970};

int main() {
  //
  Date d1;
  d1.print();                    // Date: 1.1.1970
  d1.set_default(2, 2, 1980);   // or Date::set_default
  Date d2;
  d2.print();  // Date: 2.2.1980
}
```

There is exactly one copy of a static member instead of one copy per object

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Self-Reference

```
class Person {
 public:
  Person() {}
  Person& name(std::string name) {
    _name = name;
    return *this;
  }
  // . . .

 private:
  std::string _name, _city, _company, _position;
};


int main() {
  //
  Person p1;
  p1.name("Rady").lives_in("Cairo").works_at("SoftTec").as("SW Dev.").print();
}
```

In a non-static member function, the keyword this is a pointer to the object for which the function was invoked.

In a non-const member function of class X, the type of this is X*. However, this is considered an rvalue, so it is not possible to take the address of this or to assign to this.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Member Types

```
template <typename T>
class Tree {
 private:
  using value_type = T;                   // member alias
  enum Policy { rb, splay, treeps };      // member enum
  class Node {                            // member class
   public:
    void f(Tree*);

   private:
    Node* right;
    Node* left;
    value_type value;
  };
  Node* top;

 public:
  void g(Node*);
};
```

Nested class can refer to types and static members of its enclosing class. It can also refer to non-static members (even private) when it is given an object of the enclosing class to refer to.

A class does not have any special access rights to the members of its nested class.

```
template <typename T>
void Tree<T>::Node::f(Tree* p) {
  top = right;                  // Error
  p->top = right;               // OK
  value_type v = left->value;   // OK
}
```

```
template <typename T>
void Tree<T>::g(Tree::Node* p) {
  value_type val = right->value;     // Error
  value_type v = p->right->value;    // Error
  p->f(this);                        // OK
}
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Class Types

- Concrete classes
  - Representation is part of its definition, preferred for small, frequently used, and performance-critical types, such as complex numbers and containers.
  - We can not modify the behavior, make you own class instead.
- Abstract classes
  - Provides interface to insulates a user from implementation details.
  - Behavior can be modified for improvements.
- Classes in class hierarchies

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Constructors and Destructors

```
class X {
  X(Sometype); // constructor: create an object
  X(); // default constructor
  X(const X&); // copy constructor
  X(X&&); // move constructor
  X& operator=(const X&); // copy assignment: clean up target and copy
  X& operator=(X&&); // move assignment: clean up target and move
  ~X(); // destructor: clean up
  // ...
};
```

A destructor does not take an argument,
and a class can have only one destructor.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Constructors and Destructors

```
class Member {
 public:
  Member() { std::cout << "Call member class constructor" << std::endl; }
  ~Member() { std::cout << "Call member class destructor" << std::endl; }
};

class Base {
 public:
  Base() { std::cout << "Call base class constructor" << std::endl; }
  ~Base() { std::cout << "Call base class destructor" << std::endl; }
};

class Derived : Base {
 public:
  Derived() { std::cout << "Call derived class constructor" << std::endl; }
  ~Derived() { std::cout << "Call derived class destructor" << std::endl; }
  Member member;
};

int main() {
  { Derived x; }
}
```

```
Call base class constructor
Call member class constructor
Call derived class constructor
Call derived class destructor
Call member class destructor
Call base class destructor
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Constructors and Destructors

A constructor builds a class object "from the bottom up":

- first, the constructor invokes its base class constructors,
- then, it invokes the member constructors, and
- finally, it executes its own body.

A destructor "tears down" an object in the reverse order:

- first, the destructor executes its own body,
- then, it invokes its member destructors, and
- finally, it invokes its base class destructors.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Calling Destructors

```
class Nonlocal {
 public:
  // ...
  void destroy() { this->~Nonlocal(); }  // explicit destruction
 private:
  // ...
  ~Nonlocal() {}  // don't destroy implicitly
};


void user() {
  Nonlocal x;                     // Error
  Nonlocal* p = new Nonlocal;   // OK
  // ...
  delete p;       // Error
  p->destroy();   // OK
}
```

If declared for a class X, a destructor will be implicitly invoked whenever an X goes out of scope or is deleted. This implies that we can prevent destruction of an X by declaring its destructor =delete or private.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Virtual Destructors

```
class Shape {
 public:
  // ...
  virtual void draw() = 0;
  virtual ~Shape(){};
};


class Circle : public Shape {
 public:
  // ...
  void draw() override { std::cout << "Draw circle" << std::endl; }
  ~Circle() override { std::cout << "Remove circle" << std::endl; }
  // ...
};


void test(Shape* p) {
  p->draw();  // invoke the appropriate draw()
  // ...
  delete p;  // invoke the appropriate destructor
};
```

A destructor can be declared to be virtual, and usually should be for a class with a virtual function.

The reason we need a virtual destructor is that an object usually manipulated through the interface provided by a base class is often also deleted through that interface

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Thank you