directory

- review
- 4.1 构造函数
- 4.2 析构函数
- 4.3 对象的构造与析构时机(局部对象和全局对象)
- 4.4 引用
- 4.5 运算符重载

review

- 函数重载
- 自定义类与对象
- 数据成员、成员函数
- 访问权限
- this指针
- 宏与内联函数

4.1 构造函数

- 对象初始化由编译器在创建对象处，**自动生成**调用构造函数的代码完成。
- 构造函数是类的特殊**成员函数**，确保类的每个对象都能正确地初始化。
- 构造函数**没有返回值类型**，函数名与类名相同
- 类的构造函数**可以重载**，即可以使用不同的函数参数进行对象初始化

初始化列表

- 构造函数可以使用**初始化列表**初始化**成员数据**
- 该列表在定义构造函数时使用，位置在函数体之前、**函数参数列表之后**，以冒号作开头。
- 形式：数据成员(初始值)

```
class Student {
    int ID;// 默认private类型，不能访问
public:
    Student(int id) { ID = id; }
    Student(int id): ID(id){} // 初始化函数列表
    Student(int year, int order) {
            ID = year * 10000 + order;
    }
    int getId(Student s) { return s.ID; }// 只能通过属性为public的函数访问
};
int main() {
    Student stu1(1);
    Student stu2(1, 2);
    std::cout << stu1.getId(stu1) << std::endl; // 1
    std::cout << stu2.getId(stu1) << std::endl; // 10002
}
```

- **按照声明的顺序初始化**，不是按照出现在初始化列表中的顺序

```
// 在下面的代码中，编译器先初始化ID1，再初始化ID2，因此ID1的值将不可预测
class Student {
 int ID1; //声明
 int ID2; //声明
public:
    Student(int id) : ID2(id), ID1(ID2) { }
};
```

- 在构造函数的初始化列表中，还可以**调用其他构造函数**，称为**委派构造函数**

```
class Info {
public:
    Info() { Init(); }
    Info(int i) : Info() { id = i; }
    Info(char c) : Info() { gender = c; }
private:
    void Init() {
        name = "Unknown";
        age = 0;
    }// 其他初始化
    int id;
    char gender;
    std::string name;
    int age;
};
```

就地初始化

- C++11之前，类中的一般成员变量不能在类定义时进行初始化，它们的初始化操作**只能通过构造函数进行。**
- C++11新增支持如下初始化操作，称为就地初始化:
- 注意：就地初始化只是一种简便的表达方式，**实际操作仍然在对象构造的时候执行**

```
class A {
private:
    int a = 1; //声明+初始化
    double b {2.0}; //声明+初始化
public:
    A() {} //a=1 b=2.0
    A(int i):a(i) {} //a=i b=2.0
    A(int i, double j):a(i), b(j) {} //a=i b=j
};
```

默认构造函数(缺省构造函数)

- **不带任何参数的**构造函数，或每个形参提供默认实参的构造函数

```
class A {
private:
    int a = 1;
    double b {2.0};
public:
    A() {} //定义默认构造函数
    A(int i):a(i) {}
    A(int i, double j):a(i), b(j) {}
};
```

- 使用默认构造函数（没有参数）来生成对象时，对象定义的格式为：

```
ClassName a;    //调用默认构造函数
ClassName b = ClassName();  //同样调用默认构造函数
```

注意区分：

```
ClassName c();  //这声明了一个返回值为ClassName的函数
```

- 在类的构造函数中，除了执行函数体内声明的语句，编译器还会做一些额外操作
- 例如会自动调用成员变量的默认构造函数
    - **先调用成员变量的构造**，再执行自己的构造函数

```
#include <iostream>
class A {
public:
    A() { std::cout << "A()" << std::endl; }
};
class B {
public:
    A a; // 先调用A类的默认构造函数
    B() { std::cout << "B()" << std::endl; }
};
B b;
int main() { return 0; }
```

隐式定义的默认构造函数

- 不用手动定义默认构造函数，编译器可以隐式地合成了一个默认构造函数

```
class A {
public:
    int data = 0;
};
A a;
```

等价于

```
class A {
public:
   int data = 0;
    A() {}
};
A a;
```

- 如果已经定义了其他构造函数，编译器将不会隐式合成默认构造函数

```
class A {
private:
    int a = 1;
    double b {2.0};
public:
   A(int i):a(i) {}
};

A a;  //编译错误
A a(2); //编译通过
```

- 可以手动指定生成默认版本的构造函数：即便其他构造函数存在，编译器也会定义隐式默认构造函数

```
// 按照下面方法生成类对象，编译和执行都不会报错。
// 此时'c'先被转换成int型值，然后调用构造函数A(int i)

class A {
private:
    int a = 1;
    double b {2.0};
    char c = 'c';
public:
    A() = default; // c++11起
    A(int i):a(i) {}
};
A a('c'); // 编译通过
```

- 使用delete显式地删除构造函数，避免产生未预期行为的可能性

```
class A {
private:
       int a = 1;
       double b {2.0};
       char c = 'c';
   public:
       A() = default;
       A(int i):a(i) {}
       A(char ch) = delete;
};
A a('c'); // 编译错误
```

其他

- 默认构造函数在什么情况下会被隐式定义
- 参考： https://zh.cppreference.com/w/cpp/language/default_constructor

```
struct A {
    int x;
    A(int x = 1): x(x) {} // 用户定义默认构造函数
};

struct B: A {
    // 隐式定义 B::B()，调用 A::A()
};

struct C {
    A a;
    // 隐式定义 C::C()，调用 A::A()
};

struct D: A {
    D(int y): A(y) {}
    // 不会声明 D::D()，因为已经有其他构造函数
};

struct E: A {
    E(int y): A(y) {}
    E() = default; // 显式预置，调用 A::A()
};

struct F {
    int& ref; // 引用成员
    const int c; // const 成员
    // F::F() 被隐式定义为弃置的
};

// 用户声明的复制构造函数（由用户提供，被弃置或被预置）
// 防止隐式生成默认构造函数
```

```
struct G {
    G(const G&) {}
    // G::G() 被隐式定义为弃置的
};

struct H {
    H(const H&) = delete;
    // H::H() 被隐式定义为弃置的
};

struct I {
    I(const I&) = default;
    // I::I() 被隐式定义为弃置的
};

int main() {
    A a;
    B b;
    C c;
//  D d; // 编译错误
    E e;
//  F f; // 编译错误
//  G g; // 编译错误
//  H h; // 编译错误
//  I i; // 编译错误
}
```

- 继承和虚函数以后讨论

对象数组的初始化

```
//无参定义对象数组，必须要有默认构造函数
A a[3] = {1, 3, 5}; // 三个实参分别传递给3个数组元素的构造函数

//如果构造函数只有一个参数
A a[3] = {1, 3, 5}; // 三个实参分别传递给3个数组元素的构造函数

//如果构造函数有多个参数
A a[3] = {A(1, 2), A(3, 5), A(0, 7)}; // 构造函数有两个整型参数
```

4.2 析构函数

- 类似指针，delete指针释放内存，析构清除和释放资源
- 一个类只有一个析构函数，名称是 **~类名**，**没有函数返回值**，**没有函数参数**
    - 清楚对象占用的资源是无条件的，不需要任何选项，即清楚方式唯一
- 编译器在对象生命期结束时自动调用类的析构函数，以便释放对象占用的资源，或其他后处理
- 和默认构造函数一样，析构函数除了执行函数体内声明的语句，编译器还会做一些**额外操作**
    - 例如会**自动调用成员变量的析构函数**
        - **先执行自己的析构函数，再调用成员变量的析构**

```
#include <iostream>
class A {
public:
    A() { std::cout << "A()" << std::endl; }
    ~A(){ std::cout << "A is distroyed" << std::endl;}
};
class B {
public:
    A a; // 先调用A类的默认构造函数
    B() { std::cout << "B()" << std::endl; }
    ~B(){ std::cout << "B is distroyed" << std::endl;}
};

int main() {
    B b;
    return 0;
}
//A()
//B()
//B is distroyed
//A is distroyed
```

- 隐式定义的析构函数不会delete指针成员
  - 因此可能造成内存泄露
- Q:
  - 析构函数在什么情况下会被隐式定义?
  - 析构函数的行为?
- 析构函数还会自动拓展一些行为，在继承、虚函数部分讨论

4.3 对象的构造与析构时机(局部对象和全局对象)

局部对象的构造与析构

- 在程序执行到该局部对象的代码时被初始化
- 在局部对象声明周期结束、即所在作用域结束后被析构

```
#include <iostream>
using namespace std;
class Example {
    int index;
public:
    Example(int i): index(i)
        {cout << index << " is created\n";  }
    ~Example() {  cout << index << " is destroyed\n"; }
};
void create_example(int i) {
    Example e(i); // 只在函数内存在
    cout << "Function is over\n";
}
```

```
int main() {
    for(int i = 1; i < 3; i++) {
        Example e(0); // 只在当前循环内存在
        create_example(i);
    }
    return 0;
}
// 输出
// 0 is created
// 1 is created
// Function is over
// 1 is destroyed
// 0 is destroyed

// 0 is created
// 2 is created
// Function is over
// 2 is destroyed
// 0 is destroyed
```

全局对象的构造与析构

- 在main函数调用之前进行初始化
- 在同一编译单元（源文件）按照定义顺序初始化
- 在不同编译单元中，对象初始化顺序不确定
- 在main函数执行完return之后被析构
- 全局对象存在问题：
    - 构造顺序不能确定，所以全局对象之间不能有依赖关系
    - 增大代码的耦合性
    - 使用参数替代全局对象

4.4 引用

- 类型名&引用名 变量名

```
int v0;
int& v1 = v0;// 在内存中是同一单元的两个不同名字
int& m = s.m;// 被引用的变量名可以是类的成员变量
```

- 修改被引用的变量和引用变量的效果是双向的

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    cout << "i=" << i << endl; // i=1
    int& j = i;  // j是初始化为的i的int引用
                 // i和j是同一个变量的两个别名
```

```
    cout << "j=" << j << endl; // j=1
    i = 2;
    cout << "j=" << j << endl; // j=2，修改i，等于修改j
    j = 3;
    cout << "i=" << i << endl; // i=3，修改j，等于修改i
    return 0;
}
```

- 引用必须在定义时进行初始化
- 引用不能修改引用指向
- 被引用的变量名可以是类的成员变量
- 函数参数是引用类型：表示函数的形式参数与实际参数是同一个变量，改变形参将改变实参
    - 如swap函数

```
void swap(int& a, int& b) { int tmp = b;
b = a;
a = tmp;
}
```

- 函数返回值可以是引用类型，但**不得指向函数临时变量**
- 把引用作为返回值

```
#include <iostream>
using namespace std;

int a[3] = {1,3,5}; // 全局数组
int& get(int i) { return a[i]; } // 返回a[i]的引用
int main() {
    for(int i = 0; i < 3; i++) {
        cout << "old a[" << i << "]=" << get(i) << endl;
        get(i) += 1;
        cout << "new a[" << i << "]=" << get(i) << endl;
    }
    return 0;
}
//old a[0]=1
// new a[0]=2
// old a[1]=3
// new a[1]=4
// old a[2]=5
// new a[2]=6
```

- 和指针的区别
    - 不存在空引用，引用必须连接到一块合法的内存
    - 一旦引用被初始化为一个对象，就不能被指向到另外一个对象， 指针可以在任何时候指向到另一个对象
    - 引用必须在创建时被初始化为一个对象；指针可以在初始化时置空，之后再指向对象

4.5 运算符重载

- 基本类型: int, long, char, double
- 自定义类型: class
- c++已经对基本类型内置了基本操作
  - 如何对自定义类型完成基本操作?
- 运算重载的两种方式
  - 全局函数: `A operator+(A a, A b){...}`
  - 成员函数:

```
class A {
    int data;
public:
    A operator+(A b){...};
}
```

```
#include <iostream>
using namespace std;
class A {
public:
    int data;
    A(int i) { data = i; }
    // 重载+=
    A& operator+=(A& a) { data += a.data; return *this;}
    // 重载+的第一种：成员函数
    A operator+(A& a) {
        A new_a(data + a.data);
        return new_a;
    }
};
// 重载+的第二种：全局
A operator+(A& a1, A& a2) {
    A new_a(a1.data + a2.data);
    return new_a;
}
int main() {
    A a1(2), a2(3);
    a1 += a2; // 调用operator+=()
    cout << a1.data << endl; // 5
    cout << (a1 + a2).data << endl; // 调用operator+()
    return 0;
}
```

- 可以重载的的运算符

前缀与后缀的++和--

- 前缀运算符重载声明

    - `ClassName operator++();`
    - `ClassName operator--();`
- 后缀运算符：加入没有使用的哑元参数区分前后缀
    - `ClassName operator++(int dummy)`
        - ++a <=> operator++(a)
        - a++ <=> operator++(a,int)
    - `ClassName operator--(int dummy)`
- 注：哑元可以没有变量名
    - `int fun(int,int a){ return a/10*10; }`

```
// 前缀++int重载示例
#include <iostream>
using namespace std;

class Test {
public:
  int data = 1;
  Test(int d) {data = d;}
  //成员函数方法:
  Test& operator++ () {
    ++data;
    return *this;
  }
};
//全局方法:
Test operator++(Test& t) {
    ++t.data;
    return t;
}

int main() {
  Test test(1);
  ++test;
  return 0;
}
```

```
//后缀int++重载示例
#include <iostream>
using namespace std;

class Test {
public:
  int data = 1;
  Test(int d) {data = d;}
  // 成员函数方法:
  Test operator++ (int) { //这里定义了哑元参数int，但没有变量名，所以函数实现中永远不会用到该变量
    Test test(data);
    ++data;
```

```
    return test;
  }
};
// 全局方法:
Test operator++(Test& t, int) {
    Test new_t(t.data);
    ++t.data;
    return new_t;
}
int main() {
  Test test(1);
  test++;
  return 0;
}
```

函数运算符()重载

- 在自定义类中也可以重载函数运算符()，它使对象看上去象是一个函数名

```
在自定义类中也可以重载函数运算符()，它使对象看上去象是一个函数名

ReturnType operator() (Parameters) {
    ...
}

ClassName Obj;
Obj(real_parameters); //注意不是调用构造函数!
//  Obj.operator() (real_parameters);
```

```
// 函数运算符()重载示例
#include <iostream>
using namespace std;

class Test {
public:
  int operator() (int a, int b) {
    cout << "operator() called. " << a << ' ' << b << endl;
    return a + b;
  }
};

int main() {
  Test sum;
  int s = sum(3, 4); /// sum对象看上去象是一个函数，故也称“函数对象”
  cout << "a + b = " << s << endl;

  int t = sum.operator()(5, 6);
  return 0;
}
```

重载

- 如果返回类型是引用，则数组运算符调用可以出现在等号左边，接受赋值，即
  - `Obj[index] = value;`
- 如果返回类型不是引用，则只能出现在等号右边 - `Var = Obj[index];`

```
#include <iostream>     // cout
#include <cstring>      // strcmp
using namespace std;

char week_name[7][4] = {    "mon", "tu", "wed",
                                "thu", "fri", "sat", "sun"};

class WeekTemperature {
    int temperature[7];
    int error_temperature;
public:
    int& operator[] (const char* name) // 字符串作下标 注意返回值
    {
        for (int i = 0; i < 7; i++) {
            if (strcmp(week_name[i], name) == 0)
                return temperature[i];
        }
        return error_temperature; //没有匹配到字符串
    }
};
// 关于数组下标运算符重载的测试
int main()
{
WeekTemperature beijing;
    beijing["mon"] = -3;
    beijing["tu"] = -1;
    cout    << "Monday Temperature: "
            << beijing["mon"] << endl;

    return 0;
}
//Monday Temperature:-3
```

只能成员函数重载的运算符

- =,[],(),->只能通过成员函数来重载
- 当没有自定义operator=时，编译器会自动合成一个默认版本的赋值操作
- 在类内定义operator=，编译器则不会自动合成
- 如果允许使用全局函数重载，可能会对是否自动合成产生干扰

对象输入输出 —— 流运算符重载

```
istream& operator>> (istream& in, Test& dst );

ostream& operator<< (ostream& out, const Test& src );
```

- 函数名为：operator>> 和 operator<<
- 不修改istream和ostream类的情况下，只能使用全局函数重载
- 返回值为：istream& 和 ostream&，均为引用
- 参数分别：流对象的引用、目标对象的引用。对于输出流，目标对象一般是常量引用。

```
#include <iostream>
using namespace std;

class Test {
    int id;
public:
    Test(int i) : id(i) { cout << "obj_" << id << " created\n"; }

    friend istream& operator>> (istream& in, Test& dst);
    friend ostream& operator<< (ostream& out, const Test& src);
};
istream& operator>> (istream& in, Test& dst) {
        in >> dst.id;
        return in;
}
ostream& operator<< (ostream& out, const Test& src) {
    out << src.id << endl;
    return out;
}

int main() {
    Test obj(1);
    cout << obj;  // operator<<(cout,obj)
    cin >> obj;  // operator>>(cin,obj)
    cout << obj;
    return 0;
}
```