创建与销毁·一

创建与销毁·一

By 曹菡雯（计 03）、赵晨阳（软 01）、罗华坤（化 93）、李晨宇（材 01）

readme

这一部分主要是 L4-创建与销毁的课堂笔记整理，由小组四名同学共同完成。在课件的基础上，我们尽力做到了对于每一页 PPT 都有完整的解读，同时联系了前后课程的内容与课程作业，对于课件中的操作也有一定的扩展。

5 月 2 日更新：将原文档中所有代码重新进行了编写，有助于阅读。
5 月 3 日更新：修改了 1.4 节的笔误

如果阅读时间不够充足，建议阅读课堂的扩展部分。
1.1.4 初始化列表和构造函数体的基本区别
1.2 委派构造函数实例
3.2 返回静态局部对象的引用

一、构造函数与析构函数

1.0 面向对象程序的可靠性

oop 三性：简单性、清晰性、普遍性。对用户定义类型进行严格的类型检查。隐藏实现，防止受到不必要的干扰。对象的初始化和清除，需要自动进行。忘记初始化或清除变量可能会导致程序崩溃。由类生成的对象是一种新型的变量，也要初始化。由于隐藏实现（访问权限控制），对象的有些私有数据成员只有类的设计者知道，而且只允许类的成员函数访问。尽管可以由通过显式调用对象成员函数来初始化对象，但这种做法缺少强制性，因而容易被程序员遗忘。

结论：如何进行初始化和清除(HOW)，应由类设计者决定。
何时进行初始化和清除(WHEN)，应由编译器来决定。

1.1 构造函数

1.1.1 意义

对象的“生”（初始化工作）是由编译器在创建对象处自动生成调用构造函数的代码来完成的。构造函数是类的特殊成员函数，它用来确保类的每个对象都能正确地初始化。

1.1.2 语法

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。

```
class Student {
  int ID;
```

```
public:
   Student(int id) { ID = id; }
   Student(int year, int order) {
         ID = year * 10000 + order;
    }
   ...
};
```

1.1.3 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，使用“数据成员(初始值)”的形式。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。

1.1.4 初始化列表和构造函数体的基本区别

1.1.4.1 实现过程区别

构造函数可以分两个阶段进行：（1）初始化阶段；（2）函数体阶段。

函数体阶段由函数体内所有的语句组成。不管成员是否在构造函数初始化列表中显式初始化，类的数据成员初始化总是在初始化阶段进行，初始化阶段先于计算阶段。构造函数初始化列表是对类的成员做初始化，而在构造函数体内只是对类的数据成员进行了一次赋值操作。

也就是说，构造函数函数体内对于成员数据的赋值意味着初始化已经完成，仅仅是进行了赋值。这一点可以联系《创建与销毁·二》当中对于声明、定义、初始化、赋值的讨论一同理解。这也可以用于理解，为什么常量成员数据仅仅可以在初始化列表中初始化，但是不可以在函数体内赋值。（因为常量成员仅能进行初始化，却不能在初始化之后再赋值（因为常量成员不能更改）。而一个常量成员，即使在初始化列表中没有被显式初始化，也已经被初始化过了，一个初始化过的常量自然不能在函数体里被修改。）更加详细的讨论在《创建与销毁·二》。

1.1.4.2 效率区别

综前文所述，初始化列表显然避免了函数体内不必要的一些赋值过程，这在类的成员数据较为复杂时能够显著地提高效率。

1.1.5 初始化列表的初始顺序

初始化列表的成员是按照声明的顺序初始化的，而不是按照出现在初始化列表中的顺序。比如下列代码：

```
#include <iostream>
using namespace std;
class Student {
 int ID1;
 int ID2;
public:
   Student(int id) : ID2(id), ID1(ID2) { }
```

```
    void print(){cout<<"ID1 exist "<<this->ID1<<endl;
        cout<<this->ID2<<endl;
    }
};
int main(){
    Student test(100);
    Test.print();
    return 0;
}
//输出：ID1 exist 32767

100
```

在这个例子中，ID1 在 ID2 之前声明，但其初始化依赖于 ID2。初始化列表会按照声明顺序进行初始化，先用未知的 ID2 的值对 ID1 进行初始化，再用可知的 id 来初始化 ID2。从而使得 ID1 的值不可预测。

这里的不可预测在不同的编译器上表现不同。可能会有如上输出，也可能是 warning（不是 error，这里可以参考《第三次作业第二题》的解析）一般来说我们会参考 g++编译器的输出，vs 编译器有时会更严格。

1.2 委派构造函数

1.2.1 定义

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”。

1.2.2 意义

委派构造函数也是 c++11 中对 c++的构造函数的一项改进，其目的就是减少程序员写构造函数的时间。通过委派其他构造函数，多构造函数的类编写起来就很简单容易。

1.2.3 实例

对比上下两个例子。

```
class Info {
public:
 Info() : type(1), name('a') {
  InitRest();
 }
 Info(int i) : type(i), name('a') {
  InitRest();
 }
 Info(char e) : type(1),, name(e) {
  InitRest();
 }
private:
```

```
 void InitRest() { //其他初始化 }
 int type;
 char name;
} ;

class Info {
public:
 Info() { InitRest(); } //称为目标构造函数（被调用）

 Info(int i) : Info() { type = i; } //委派构造函数（调用者）

 Info(char e) : Info() { name = e; }
private:
 void InitRest() { // other int }
 int type {1};
 char name {'a'};
};
```

委派构造函数不能有初始化列表，因为 C++中，构造函数不能同时使用委派和初始化列表。只能在函数体内为 type，name 等成员赋值。

在构造函数比较多时，可以有不止一个委派构造函数。目标构造函数也可以是委派构造函数，可以在委派构造函数中形成链状的委派构造关系。

```
class Info {
public :
 Info() : Info(1) {}//这个不带参数的构造函数的目标函数就在下方，而目标函数也

有委派构造函数，从而形成了委派构造函数链。

 Info(int i) : Info(i, 'a') {} //可以委派成链状，但不能形成环。

 Info(char e) : Info(1, e) {}
private:

 Info(int i, char e) : type(i), name(e) {}//info()调用 info(1)，info(1)调用 info(1,'a')


 int type;
 char name;
};
```

这个例子其实蛮有意思的是，我们之前很少有见到构造函数放在 private 里面的例子。这个例子却实现了这一点。

1.3 就地初始化

首先需要指出的是，在课程 PPT 中对于声明、初始化、定义这三个概念有些混用。三者的广义概念界定在《创建与销毁·二》中进行了阐述。

C++11 之前，类中的一般成员变量不能在类声明时进行初始化（定义），它们的初始化操作（定义）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。（类似于给类内对象提供了缺省值。）你可以认为就地初始化是一种特殊的机制，他可以使得定义和初始化在类内得以进行。

```
class A {
private:
    int a = 1; //声明+初始化（定义）

double b {2.0}; //声明+初始化（定义）

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
};
```

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。

1.4 默认构造函数

1.4.1 定义

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，被称为“默认构造函数”，也称“缺省构造函数”。

1.4.2 语法

使用默认构造函数（没有参数）来生成对象时，对象定义的格式如下。

```
ClassName a;    //调用默认构造函数

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

注意区别和下方代码区别开来。

```
ClassName c();  //这声明了一个返回值为 ClassName，不带参数的函数
```

1.4.3 编译器的额外操作

基于之后涉及到的组合的概念，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。

```
#include <iostream>
using namespace std;
class A {
public:
    A() { cout << "A()" << endl; }
};
class B {
public:
```

```
    A a;
    B() { cout << "B()" << endl;}
};
B b;
int main() { return 0; }
```

这里会先输出 A 的默认构造函数对应的输出，再输出 B 的默认构造函数对应的输出。

也就是说，如果类 A 里面有另一个类 B 作为这个类的成员，调用类 A 的默认构造函数时会先调用 B 的构造函数。先构造成员-再构造类。

1.4.4 隐式定义的默认构造函数

有时候我们没有手动定义默认构造函数，但我们仍然能够按上述方式定义变量。这是因为编译器帮我们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;
```

关于下列程序的说法，正确的是

```
#include <iostream>
using namespace std;
class A {
public:
    A() { cout<<"A()"<<endl; }
    A(int x)
        { cout << "A(int)" << endl; }
};
class B {
    A a;
public:
    B(int x=1): a(x) {}
};
int main(){
    B b;
    return 0;
}
```

A 输出A(int)

B 输出A()

C 编译错误

提交

16

注意这个例子，b 究竟是调用了哪一个构造函数？

我们将代码修改如下：

```
#include <iostream>
using namespace std;
class A {
public:
    A() { cout<<"A()"<<endl; }
    A(int x)
      { cout << "A(int)" << endl; }
};
class B {
    A a;
public:
    B(int x=1): a(x) {}
    B():a(1){}
```

```
};
int main(){
    B b;
    return 0;
}
```

发现会因为函数调用不明确而 error，故而可以确定上述选择题是调用了缺省的构造函数而非隐式生成的默认构造函数。这一点在函数重载有讨论。

某种意义上，缺省的构造函数本质上已经实现了默认构造函数的功能。

另一方面，若用户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
```

1.4.5 显式声明默认构造函数

出于某些需要，我们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。

```
class A {
private:
    int a = 1;
    double b {2.0};
public:
    A() = default;    // C++11 起
    A(int i):a(i) {}
};
A a;
```

1.4.6 显式删除危险构造函数

有时候，我们也可以显式地声明禁用某些带有风险的构造函数。这种禁用不仅可以禁用编译器合成的默认构造函数，也可以用来禁止一些自动类型转换带来的构造函数调用

```
#include <iostream>
using namespace std;
class A {
private:
    int a = 1;
    double b {2.0};
    char c = 'c';
```

```
public:
    A() = default;
    A(int i):a(i) {cout<<i<<endl;}
};
int main(){
    A a('c');
    return 0;
}
输出结果 99
```

这一代码存在风险，本意一定是希望他报错，但是实则不会。从正确性上讲，这样的代码没有问题，char 和 int 可以类型转换，故而将‘c’转为了 int，调用了参数为 int 的构造函数。但是从工程的角度讲，这是很危险的行为。因为在开发者看来，用字符初始化应该是未定义的行为。

故而显示地禁用某一构造函数。

```
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
```

1.5 对象数组的初始化（在 main 中）

无参定义对象数组，必须要有默认构造函数

A a[50]; // 定义了一个具有 50 个元素的 A 类对象数组

如果构造函数带有参数

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

带有多个参数

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

1.6 析构函数

1.6.1 概述

对象的“死”（清除和释放资源）是由编译器在对象作用域结束处自动生成调用析构函数代码来完成的，动态分配的内存就是一种典型的需要释放的资源。

当执行到“包含对象定义范围结束处”时，编译器自动调用对象的析构函数。清除对象占用的资源是无条件的，不需要任何选项。因此，析构函数没有参数，且只有一个，即清除方式唯一。

这在《创建与销毁·二》的 3.4.1 的例子中有所体现。

参数对象的构造与析构实例

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

class A {
public:
    const char* s;
    A(const char* str):s(str) {
        cout << s << " A constructing" << endl;
    }
    ~A() { cout << s << " A destructing" << endl; }
};

void fun(A b) {
    cout << "In fun: b.s=" << b.s << endl;
}

int main() {
    A a("a");
    fun(a);
    return 0;
}
```

运行结果：

```
a A constructing
In fun: b.s=a
a A destructing
a A destructing
```

构造一次，析构两次?

没有显式定义拷贝构造函数，系统调用了隐式生成的拷贝构造函数，这当然是没有输出的。形参的构造用的是拷贝构造函数，可是析构却是共用的一个析构函数。

1.6.2 语法

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。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。

```
class ClassRoom {
   int num;
   int* ID_list;
public:
   ClassRoom() : num(0), ID_list(nullptr) {}
   ...
   ~ClassRoom() {  // 析构函数
      if (ID_list) delete[] ID_list; // 释放内存
   }
};
```

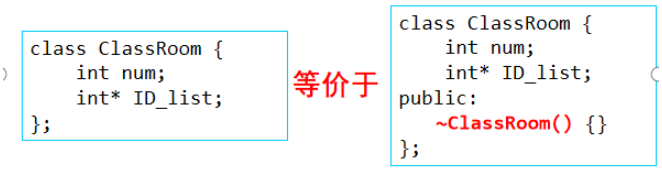
注意到这个例子也体现了对于指针，delete 之前应先检测是否为空指针。

1.6.3 析构顺序

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。例如在组合当中，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。（这点与构造函数恰恰相反）以及最基本的，先构造的后析构。

1.6.4 默认析构函数

和构造函数类似，当用户没有自定义析构函数时，编译器会自动合成一个隐式的析构函数。



问题在于隐式定义的析构函数不会 delete 指针成员，可能造成内存泄露。

二、对象的析构与构造

2.1 局部对象的构造与析构

2.1.1 定义

非静态的局部对象：在程序执行到该局部对象的代码时被初始化。在局部对象生命周期结束、即所在作用域结束后被析构。这里注意与静态的局部对象进行对比。详见《创建与销毁·二》。

作用域：该变量能够被引用的区域，例如，{}将会形成一个作用域。

非静态全局变量：在 main()函数调用之前进行初始化。在同一编译单元中，按照定义顺序进行初始化。

编译单元：通常同一编译单元就是同一源文件。

不同编译单元间，对象初始化顺序不确定。

在 main()函数执行完 return 之后，对象被析构。

2.2.2 全局变量的局限性

尽量少用全局对象。首先，全局对象的构造顺序不能完全确定，所以全局对象之间不能有依赖关系，否则会出现问题。其次，全局对象会增大代码的耦合性，导致程序难以复用或者测试。

解决方案：使用参数来替代全局对象。

下列程序的运行结果是

```
#include <iostream>
using namespace std;
class A {
    const char* s;
public:
    A(const char* str):s(str) {
        cout << s << " A constructing" << endl;
    }
    ~A() { cout << s << " A destructing" << endl; }
};
class B {
public:
    B() { cout << "B constructing" << endl; }
    ~B() { cout << "B destructing" << endl; }
};
A global_obj("global");
int main() {
    cout << "Entering main..." << endl;
    B local_obj;
    cout << "Exiting main..." << endl;
    return 0;
}
```

A
global A constructing
Entering main...
B constructing
Exiting main...
B destructing
global A destructing

B
Entering main...
global A constructing
B constructing
Exiting main...
global A destructing
B destructing

C
global A constructing
Entering main...
B constructing
Exiting main...
global A destructing
B destructing

提交

34

这道题就是强调了全局变量在 main()函数执行完 return 之后被析构。而 local_obj 相当于是 main 的局部变量，会在全局变量后析构。

三、引用

3.1 定义与语法

同一个内存单元的两个不同名字。

格式具名变量的别名：类型名 & 引用名 变量名

例：int v0; int & v1 = v0; v1 是变量 v0 的引用。引用必须在定义时进行初始化，且不能修改引用指向，这点和指针不同。

被引用变量名可以是类的成员变量，如 int & m = s.m;

3.2 结合函数使用

函数参数可以是引用类型，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，使用得当也可避免许多不必要的形参拷贝。如调用以下函数将交换实参的值：

```
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，但不得指向函数的非静态的临时变量。换言之，你可以返回临时变量（这相当于把它复制了一份 return 出去，它本身会被销毁），但不能返回非静态临时变量的引用。静态局部变量是可以作为返回值的，并且有着神奇的操作。（这一神奇的操作涉及到静态局部变量的作用，详见《创建与销毁·二》1.2.2.2）

右侧代码的输出结果是

A 1_4

B 4_7

C 4_4

D 该代码会访问非法内存

```
class Int {
public:
    int data;
    Int() { data = 1; }
    Int(int i): data(i) {}
};
void func1(Int& a, Int b) {
    a.data += b.data;
}
Int& func2(Int& a, Int b) {
    func1(a, b);
    Int tmp(a.data + b.data);
    return tmp;
}
int main() {
    Int a, b(3);
    Int& f = func2(a, b);
    cout << a.data << "_";
    cout << f.data << endl;
    return 0;
}
```

提交

42

这段代码存在非法内存访问。注意到 func2 定义的局部变量 tmp 是非静态的，故而 func2 结束之后立刻析构。而 f 被定义为了 tmp 的引用，tmp 已经被销毁了，这块内存空间已经被释放。

被释放的内存，访问是危险的，但并非不可访问。现在一般的编译器都是允许访问，但访问的结果不确定，有可能是运行错误，有可能是不确定的值。这个可能需要等到学汇编会了解的更清楚。

故而这段代码在 glot 编译器上的结果为 4_0。

然而我们对局部变量 tmp 加上 static 修饰。

```
#include <iostream>
using namespace std;
class Int {
public:
   int data;
   Int() { data = 1; }
   Int(int i): data(i) {}
};
void func1(Int& a, Int b) {
   a.data += b.data;
}
Int& func2(Int& a, Int b) {
   func1(a, b);
   Int static tmp(a.data + b.data);
   return  tmp;
}
int main() {
   Int a, b(3);
   Int &f = func2(a, b);
   cout << a.data <<"_"<<f.data << endl;
   return 0;
}
```

```
输出：4_7
```

非常神奇，我们不仅通过 static 修饰保留了局部变量，似乎还改变了这个局部变量。

```
#include <iostream>
using namespace std;
class Int {
public:
    int data;
    Int() { data = 1; }
    Int(int i): data(i) {}
};
void func1(Int& a, Int b) {
    a.data += b.data;
}
Int& func2(Int& a, Int b) {
    func1(a, b);
    Int static tmp(a.data + b.data);
    return  tmp;
}
int main() {
    Int a, b(3);
    Int &f = func2(a, b);
    cout << a.data <<"_"<<f.data<<endl;
    f.data++;
    cout << a.data <<"_"<<func2(a,b).data<<"_"<<a.data<<endl;
    return 0;
}
输出：4_7

4_8_7
```

这似乎违反了静态局部变量只能够被定义它的函数体操作这一点，实则不然。我们之前叙述的是，静态局部变量不能在函数外被使用，但是但它的内存空间可以。上例是绕过了语言层面，直接对于内存进行了操作。

对于变量的操作，是指例如编写一行 C 语言代码，明确地操作这个变量，比如变量 x++。但是操作可以通过指针，比如定义一个指针 p 来指向这个地址，然后对*p 进行各类操作。这时候编译器并不知道*p 指向的内存空间究竟怎么来的，它只知道是一段内存空间，进行了一系列操作。

简而言之，你不能在函数体外使用静态局部变量的名字，但是可以使用这块内存空间。

又比如下面这段代码：

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

```
class Int {
public:
    int data;
    Int() { data = 1; }
    Int(int i): data(i) {}
};
void func1(Int& a, Int b) {
    a.data += b.data;
}
Int& func2(Int& a, Int b) {
    func1(a, b);
    Int static tmp(a.data + b.data);
    return tmp;
}
int main() {
    Int a, b(3);
    Int &f = func2(a, b);
    cout << a.data <<"_"<<f.data<<endl;
    func2(a,b).data++;
    cout << a.data <<"_"<<func2(a,b).data<<"_"<<a.data<<endl;
    return 0;
}
```

输出：4_7

7_8_10

我企图直接对 func2(a,b).data++；因为我返回类型为 tmp 的引用，也成功实现了对于 tmp 的内存空间的操作。

```
#include <iostream>
using namespace std;
class Int {
public:
    int data;
    Int() { data = 1; }
    Int(int i): data(i) {}
};
void func1(Int& a, Int b) {
    a.data += b.data;
}
Int& func2(Int& a, Int b) {
    func1(a, b);
    Int static tmp(a.data + b.data);
    return  tmp;
}
int main() {
```

```
    Int a, b(3);
    Int &f = func2(a, b);
    cout << a.data <<"_"<<f.data<<endl;
    tmp.data++;
    cout << a.data <<"_"<<func2(a,b).data<<"_"<<a.data<<endl;
    return 0;
}
```

输出：

```
error: use of undeclared identifier 'tmp'
    tmp.data++;
    ^
1 error generated.
```

3.3 引用的其他特点

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。

引用的优势：更灵活地支持运算符重载。

引用的特性：创建时必须初始化、初始化后便不能指向其他对象，不存在空引用。

四、运算符重载

4.1 意义

用户自定义类，没有对常用的运算符进行定义，比如想要表示两个类对象相加，无法采用 a+b 这种方式。可以采取定义一个 add 函数的方式，解决这种问题。但这种实现方式，在调用的时候，会和基础类型差别很大，缺少编程的一致性。需要过多地区分自定义类和基础类别，调用起来也不方便。

因此，我们引入运算符重载。

4.2 语法

运算符重载需要按规则声明执行该运算的函数。例如 + 对应 operator+（operator 就是运算的意思，相当于我人为定义了对于 A 类的+号）

运算重载一般有两种方式（注意参数不同），且只能用一种

全局函数型运算符重载

```
    A operator+(A& a1, A& a2) {
  A new_a(a1.data + a2.data);
```

```
    return new_a;
}
```

成员函数型运算符重载

```
class A{
    int data;
public:
    A operator+(A& a) {
 A new_a(data + a.data);
return new_a;}
};
```

注意理解参数的不同。定义为全局函数型运算符重载之后，需要指出对于哪两个对象进行操作。但是定义为成员函数型运算符重载时，我这一函数是某一对象的成员函数，只需指明另一对象。

使用成员函数型重载

```
#include <iostream>
using namespace std;
class A {
public:
    int data;
A(int i) { data = i; }
A& operator+=(A& a) { data += a.data; return *this;}
A operator+(A& a) {
A new_a(data + a.data);
return new_a;
}
};
int main() {
    A a1(2), a2(3);

    a1 += a2; // 调用 operator+=()

    cout << a1.data << endl;

    cout << (a1 + a2).data << endl; // 调用 operator+()

    return 0;
}
```

显然对于+=和+都应该重载，不可能只重载+就解决问题。

这里的 a1+a2 相当于调用函数 a1.operator+(a2);

使用全局函数型重载

```
#include <iostream>
using namespace std;
class A {
```

```
public:
    int data;
    A(int i) { data = i; }
};
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;
    cout << (a1 + a2).data << endl; // 调用 operator+()
    return 0;
}
```

这里的 a1+a2 相当于调用函数 operator+(a1,a2);

4.3 具体的重载实例

4.3.1 可重载类型

双目算术运算符

+(加)，-(减)，*(乘)，/(除)，%(取模)

关系运算符

==(等于)，!=(不等于)，<(小于)，>(大于>，<=(小于等于)，>=(大于等于)

逻辑运算符

||(逻辑或)，&&(逻辑与)，!(逻辑非)

单目运算符

+(正)，-(负)，*(指针)，&(取地址)

自增自减运算符

++(自增)，--(自减)

位运算符

|(按位或)，&(按位与)，~(按位取反)，^(按位异或)，<<(左移)，>>(右移)

赋值运算符

=, +=, -=, *=, /= , % = , &=, |=, ^=, <<=, >>=

空间申请与释放
new, delete, new[] , delete[]

其他运算符
()(函数调用)，->(成员访问)，,(逗号)，[](下标)

4.3.2 前后缀重载
4.3.2.1 声明
前缀运算符重载声明
ClassName operator++();
ClassName operator--();

后缀运算符重载声明
ClassName operator++(int dummy);
Dummy 表示哑元，实际上哑元可以没有名字。

```
ClassName operator++(int dummy);
++a 等价于 operator++(a)
a++ 等价于 operator++(a,int)
```

```
ClassName operator--(int dummy);
int fun(int,int a){ return a/10*10; }
```

哑元的意义：设想没有这一哑元，那么 a++和++a 的声明完全相同。然而实际上，两个函数的功能完全不同，需要实现重载。这一实现就是借助哑元达成的。

编译器在编译 a++时，等价于 operator++(a,int)，从而调用了后缀运算符。编译器自动识别，就像是 a1+a2 等价于 a1.operator+(a2)，这种等价就是编译器自动等价。

4.3.2.2 语义区别
int a = ++b; //先完成 b+1 操作，再赋值
int a = b++; //先完成赋值，再 b+1 操作

4.3.2.3 前缀运算符重载实例

```
#include <iostream>
using namespace std;
class Test {
public:
  int data = 1;
  Test(int d) {data = d;}
  Test& operator++ () {
    ++data;
    return *this;
  }
};
```

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

4.3.2.4 后缀运算符重载

4.3.2.4.1 实例一

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

class Test {
public:
  int data = 1;
  Test(int d) {data = d;}
  Test operator++ (int) {
    Test test(data);
    ++data;
    return test;
  }
};
```

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

首先需要指出的是，函数的返回值并不一定需要使用。这一段就是典型的例子，尽管我返回的是个 Test 对象，但是没有人接收这一对象，他会在主函数的对应语句（也就是 test++;）结束后被释放。虽然在这个例子里面，test++并没有用到 Test 类型的返回值，可以改为 void，但是在其他情况下是需要用到这一个返回值的。

4.3.2.4.2 关于析构时机

对于此处局部对象的析构时机，结合第三次作业第二题的 f1 部分进行一定的解读。

```
Test f1(Test a)
{
   a.print("a");   return a;
}
Test A = f1(a);
```

对于这一赋值构造(构造外面的 A）+函数调用语句，构造 A 和析构返回值和形参的先后顺序：先执行完 f1 的函数体，暂不析构。然后执行完整个语句，具体到这个语句，就是外部对 A 的移动构造，再析构。

完成了对 A 的移动构造之后，就已经完成了两次对应移动构造的输出。分别是因为返回值优化被禁用而对返回值进行的规定的移动构造，以及返回的对象对于 A 的移动构造。接下来，这一语句执行完毕后才进行函数体的析构，也就是连续的两次析构。第一次析构掉形参，第二次析构掉返回值。（先构造则先析构）

至于为什么对 A 是移动构造

```
Test A = f1(a);//f1(a)是个右值，因为你没法对 f1(a)++; f1(a)是个右值，故而默认调用了移动构造函数
```

这里需要意识到，f1(a)本身是个右值，但是 A 就是个左值了。

还值得说明的是，上文的析构时机是建立在主函数的语句既有赋值又有函数调用的基础上，也就是函数的返回值有赋值作用。如果函数返回值没有赋值的作用，那么返回值会立刻在函数体结束后被析构，而不是主函数的对应语句结束后析构。譬如上文简单的写 a++就会让 test 立刻被析构。

4.3.2.4.3 实例二

```
#include <iostream>
using namespace std;
class Test {
public:
  int data = 1;
  Test(int d) {data = d;}
  Test operator++ (int) {//后缀重载
    Test test(data);
    ++data;
    return test;
  }
};
int main() {
  Test test(1);
  test=test++;
  cout<<test.data<<endl;
  return 0;
}
输出：1
```

为什么会输出 1？我们结合修改后的代码进行解释。

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

```
class Test {
public:
  int data = 1;
  Test(int d) {data = d;}
  Test operator++ (int) {
    Test new_test(data);
    ++data;
    return new_test;
  }
};
int main() {
  Test test(1);
  test=test++;
  cout<<test.data<<endl;
  return 0;
}
```

输出：1

先构建了 new_test，然后对原有的 test.data++，然后把 new_test 返回给了 test。注意到这一逻辑过程的先后顺序，我虽然对于 test.data++了，但这步之后用返回的 new_test（data 还是 1）对 test 进行了移动赋值，覆盖了++的效果，故而 test.data 还是 1。

另一方面，避免主函数的局部变量和函数体局部变量重名必然是个好习惯！

4.3.2.5 全局型重载

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

class A {
public:
    int data;
    A() { data = 0; }
    A(int i) { data = i; }
};
A operator++(A& a) {//前缀

    ++a.data;
    return a;
}
A operator++(A& a, int) { //哑元，后缀

    A new_a(a.data);
    ++a.data;
    return new_a;
```

```
}
int main() {
    A a(1);
    cout << (++a).data << endl; // 2
    cout << (a++).data << endl; // 2
    cout << a.data << endl; // 3
    return 0;
}
```

4.3.3 函数运算符()重载

在自定义类中也可以重载函数运算符() ，它使对象看上去如同是一个函数名，可以称之为对象函数。

```
#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;
}
ReturnType operator() (Parameters) {
    ...
}

ClassName Obj;

Obj(real_parameters); //注意这里显然不是在调用构造函数！
```

实际上在调用 Obj(real_parameters); 时，等价于调用了 Obj.operator()(real_parameters);

4.3.4 数组下标重载

函数声明形式：返回类型 operator[] (参数);

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;（也就是返回左值）如果返回类型不是引用，则只能出现在等号右边 Var = Obj[index];（返回了右值）

注意到，这里 Obj 是一个对象，而不是一个数组。这是对于一个类，定义了一个成员函数来重载数组下标。

```
#include <iostream> //为了使用 cout

#include <cstring>  //为了使用 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

   {
      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
   }
};
int main()
{
WeekTemperature beijing;
   beijing["mon"] = -3;
   beijing["tu"] = -1;
   cout    << "Monday Temperature: "
         << beijing["mon"] << endl;

   return 0;
}
输出结果：Monday Temperature: -3
```

const char*name:表示一个常量字符串，避免修改。这个字符串常量就是在[]里面的内容，比如[“test”]那你的参数就是 test。（注意到必须要有双引号才能是字符串）

此处，对于 private 做一说明: temp 是 private，所以直接访问 temp 这个名字不可行，但是可通过其他方式访问其引用，进而可修改内存空间。

这个名字被保护了，但是他的内存单元没有被保护。private 一般用于修饰类的内部属性（变量）和方法（函数）（即：不想暴露给外部的那些），它保证了它修饰的属性和方法不能在类的外部被【直接】访问，但可以通过类的一些 public 方法实现间接访问。

例如：

```
#include <iostream>
using namespace std;
class Test {
private: int i;
public:
Test(int j):i(j){}
void print(){cout<<this->i;}
};

int main() {
  Test test(1);
  int&b=test.i;
  b++;
  test.print();
  return 0;
}
```

这里将会访问失败，因为主函数内不可访问直接 private 成员。

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

class Test {
private: int i;
public:
Test(int j):i(j){}
void print(){cout<<this->i;}
int& setTest(){return this->i;}
};
```

```
int main() {
  Test test(1);
  int&b=test.setTest();
  b++;
  test.print();
  return 0;
}
```

此处，我通过 setTest 这一 public 方法得到了 i 的引用，再构造了另外一个引用。直接绕过了变量，对其指向的内存空间进行了操作。

4.3.5 只能成员函数型重载的运算符

=,[],(),->只能通过成员函数来重载。

这里其实可以联系到之后学习的拷贝赋值运算和移动赋值运算，因为本质上二者就是实现方式值得推究的两种运算符重载。

当我们没有显式地在类内定义这两个运算符（重载）时，编译器会自动生成缺失的部分。编译器无法获知（或者至少没有尝试去获知）类以外已经重载了 operator =，它在编译这个类时发现这个类没有重载 operator =，于是就给它补上了一个。但如果之后又在类以外重载了一个 operator =，这就会导致调用的时候不知道用哪个，产生歧义，综上 c++禁止了在类以外重载 operator= 。

换言之，编译器编译类时能看到类内重载的赋值运算符，就不会自动生成=的重载。但全局的编译器无法在编译时检测到,只有在链接的时候才能检测到。而此时会链接歧义函数。

仅仅是只是链接两个歧义函数并不会出错，在发生有歧义的调用的时候才会有问题。（也就是说，如果在类外重载 operator=，而不调用=,是不会 warning 或者 error 的。）比如我们甚至可以同时定义 int foo(int x, int y=1) 和 int foo(int x) 这两个函数，只有使用了 foo(x) 才会导致 CE。

```
#include <iostream>
using namespace std;
void func(int x){};
void func(int x,int y=0){}

int main() {
cout<<"1"<<endl;
return 0;
}
```

这不会有问题

```
#include <iostream>
using namespace std;
void func(int x){};
void func(int x,int y=0){}
int main() {
cout<<"1"<<endl;
func(1);
return 0;
}
```

这就有问题

4.3.6 流运算符重载

用户自定义的类，虽然可以像内置类型那样定义变量（对象），但想要使用流运算符输入、输出对象，则还需要为类定义流运算符重载。

4.3.6.1 语法

```
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。

备注：因为流运算符往往需要输出 private 数据，故而往往设为友元函数并在类内声明，并尽量在类外时实现。

4.3.6.2 实例

```
#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;  // 等价于 obj.operator<<(cout,obj)
   cin >> obj;   // 等价于 obj.operator>>(cin,obj)
   cout << obj;
   return 0;
}
```

注意到，函数体内用的都是 in 和 out，而不是 cin 和 cout，因为你在重载 out，你希望你的 out 对于所有的输出流都适用，那么不能写 cout。因为还有 fout 等等。如果写了 cout，没有问题，但是没法给 fout 这些用，因此写 out 习惯更好。

参数的意义：cout << asdfsad << sadf; out 即时 ostream 类对象 cout，src 即是 test 类对象 asdfsad。函数结束后，返回了一个新的 ostream&，接下来继续输出。