C/C++知识点盲区

1.指针函数与函数指针

先看下面的信号处置设置函数：

```
#include<signal.h>
void (*signal(int sig,void (*handler)(int)))(int);
```

指针函数的定义，指针函数就是返回指针的函数，定义如下：

类型名 *函数名（函数参数表列），例如，`int *fun(int ,int)`

由于*的优先级低于()的优先级，所以fun先和后面的()结合，意味着fun就是一个函数；接着与前面的*结合,这意味着这个函数的返回值是一个指针，由于前面还有一个`int`,也就是说fun是一个返回值为整形指针的函数。

返回值是函数指针的函数：

`int (* fun)(int a,int b);`

实际上一个函数指针不关心他的输入变量名字，只关心输入变量类型，因此输入变量名字可以省略掉：

`int (* fun)(int,int);`

这样就定义了一个函数指针，去掉变量名和最后的分号就是变量类型，因此fun这个函数指针的变量类型为`int (*)(int,int)`

还可以使用`typedef`定义：

`typedef int(* fun)(int,int);`

这样就可以直接利用fun去定义函数指针变量了，让这个函数指针指向某一个函数：

```
#include <stdio.h>
int add(int a, int b)
{
    return a + b;
}
int sub(int a, int b)
{
    return a - b;
}
int(*func(int a)) (int, int)
/*该函数的作用是定义一个函数，该函数的目的是返回函数的地址，我们肯定是要用一个函数指针类型的变量来接收的*/
{
    if (a == 1) {
        return add;
    }
    return sub;
}

```

```c
/* 定义函数指针类型 */
typedef int (* func_t)(int, int);

int main(int argc, const char *argv[])
{
    int k;
    func_t p1;
    int (*p2)(int, int);

    p1 = func(1);
    p2 = func(2);
    k = p1(1, 3);
    printf("k = %d\n", k);

    k = p2(1, 3);
    printf("k = %d\n", k);

    return 0;
}
```

总结一下：

如果一个函数的返回值为一个函数指针类型。我们可以分为两步来写：

第一步，先写出函数的返回值类型：

`int (*)(int,int)`

第二步，再写出一个其他返回类型的函数：

`int fun(int a)`

接下来，我们只需要将这个函数的`int`替换成`int (*)(int,int)`

`int (*fun(int a))(int int)`，将`fun(int a)`直接加到函数指针类型的星号后面即可。

再回头看signal函数的声明:

`void (*signal(int signal,void (*func)int))(int)`

我们先看最外面，可以知道该函数的返回类型是函数指针，其类型为`void (*)(int)`

再看里面，该函数的参数，参数1是signal，参数2是一个函数指针变量func。

2.`nullptr`和NULL的区别

我们声明空指针一般有以下三种办法：

```c
int *p1 = nullptr;
int *p2 = 0;
// 需要首先#inlcude <cstdlib>
int *p3 = NULL;
```

我们也可以使用NULL来初始化空指针，但是这样会导致编译器无法区分他是指针还是一个int类型的变量，比如说以下代码：

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

class Myclass
{
public:
        void printf(char *)
        {
                cout << "This is char\n" << endl;
        }
        void printf(int)
        {
                cout << "This is int\n" << endl;
        }

};

int main(int argc,char **argv)
{
        Myclass a;
        a.printf(NULL);
        a.printf(nullptr);
        return 0;
}
```

本质上来讲，`nullptr`是一个指针类型的变量值，该值代表着指针是空指针，我们用它只能来初始化指针，并不能初始化其他的Int类型的变量。但是NULL就不一样了，因为NULL本来就是0.

在C++中，在源文件中：

```cpp
#ifndef NULL
    #ifdef __cplusplus
        #define NULL 0
    #else
        #define NULL ((void *)0)
    #endif
#endif
```

3. 编写自己的头文件

由于我们在编写代码的时候有可能会出现先后包含了多个相同文件的问题，所以说，我们应当在书写头文件的时候进行适当的处理，使其可以在遇到多次包含的情况下依旧可以安全和正常的运行。

我们确保头文件多次包含仍能安全工作的常用技术是预处理器。***预处理器就是在编译之前执行的一段程序，可以部分地改变我们所写的程序。***

比如说`#include`就是一项与处理功能，当预处理器看到`#include`标记的时候，就会是使用指定的头文件的内容代替`#include`。

我们还经常使用到的一项预处理功能是***头文件保护符***，就是我们平时如何去解决头文件多次被包含的问题，这里依赖于预处理变量。预处理变量有两种状态：***已经定义***和***未定义***。

`#define`指令就是将一个名字的设定为预处理变量，另外的两个指令则分别检查某一个指定的预处理变量是否已经被定义：

`ifdef`指令当且仅当已定义时为真，`ifndef`当且仅当变量未定义的时候为真。一旦检查结果为真，则执行后续操作直到遇到`#endif`指令为止。

我们平时写头文件的时候一般是这样去写：

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include<string>

...

#endif
```

有一点需要我们去注意，预处理变量无视C++语言中关于作用域的规则。

4. `const`的用法

> 参考文章：https://zhuanlan.zhihu.com/p/134654903

4.1 常变量

变量使用`const`修饰，其值不得被改变。任何改变此变量的代码都会产生编译错误。`const`加在数据类型的前后均可。

```
void main(void)
{
    const int i = 10;    //i,j都用作常变量
    int const j = 20;
    i = 15;             //错误，常变量不能改变
    j = 25;             //错误，常变量不能改变
}
```

4.2 常指针

`const`和指针一起使用的时候有两种不同的情况：

`const`可以用来限制指针不可以改变，就是说，指针指向的内存地址不可以改变，但是可以随意的改变该地址指向的内存的内容。

```
int main(void)
{
 int i = 10;
 int *const j = &i;  //常指针，指向int型变量
 (*j)++;      //可以改变变量的内容
 j++;       //错误，不能改变常指针指向的内存地址
}
```

`const`也可以用来限制指针指向的内存不可以改变，但是指针指向的内存地址可以改变。

```
int main(void)
{
 int i = 20;
 const int *j = &i;  //指针,指向int型常量
 //也可以写成int const *j = &i;
 j++;    //指针指向的内存地址可变
 (*j)++;   //错误,不能改变内存内容
}
```

我们怎么判断`const`修饰的是指针本身还是指针指向的内存呢?

我们可以通过`const`后面修饰的内容来判断:

如果`const`后面修饰的直接是指针变量的话，那么说明，指针的内容不可以改变，也就是指针指向不能改变;

但是如果`const`后面修饰的是* 和指针变量的话，说明指针指向的内存内容不可以改变。

两种方式还可以结合起来，使得指针指向的内存以及内存的内容都不可以改变。

```
int main(void)
{
 int i = 10;
 const int *const j = &i; //指向int常量的常指针
 j++;       //错误，不能改变指针指向的地址
 (*j)++;    //错误，不能改变常量的值
}
```

4.3 `const`和引用

我们引用的时候可以使用`const`修饰符进行修饰，使得我们不能通过别名来修改变量，但是我们可以通过变量本身来修改变量的值。

```
void main(void)
{
    int i = 10;
    int j = 100;
    const int &r = i;
    int const &s = j;
    r = 20;          //错，不能改变内容
    s = 50;          //错，不能改变内容
    i = 15;          // i和r 都等于15
    j = 25;          // j和s 都等于25
}
```

4.4 `const` 和成员函数

声明成员函数的时候，末尾加`const`修饰，表示在成员函数内不得改变该对象的任何数据。该种模式常常内用来表示对象数据只读的访问模式。

```
class MyClass
{
    char *str ="Hello, World";
```

```
    MyClass()
    {
        //void constructor
    }
    ~MyClass()
    {
//destructor
    }

    char ValueAt(int pos) const    //const method is an accessor method
    {
        if(pos >= 12)
              return 0;
  *str = 'M';       //错误，不得修改该对象
        return str[pos];     //return the value at position pos
    }
}
```

4.5 const和重载

参考：https://www.cnblogs.com/qingergege/p/7609533.html

4.5.1 常成员函数和非常成员函数之间的重载

首先先回忆一下常成员函数

声明：<类型标志符>函数名（参数表）const；

说明：

（1）const是函数类型的一部分，在实现部分也要带该关键字。

（2）const关键字可以用于对重载函数的区分。

（3）常成员函数不能更新类的成员变量，也不能调用该类中没有用const修饰的成员函数，只能调用常成员函数。

（4）**非常量对象也可以调用常成员函数，但是如果有重载的非常成员函数则会调用非常成员函数**（就是说，在有重载的情况下，非常量对象调用函数的时候，会去调用非常成员函数）。

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

class Test
{
protected:
    int x;
public:
    Test (int i):x(i) { }
    void fun() const
    {
        cout << "fun() const called " << endl;
    }
    void fun()
    {
        cout << "fun() called " << endl;
```

```
    }
};

int main()
{
    Test t1 (10);
    const Test t2 (20);
    t1.fun();
    t2.fun();
    return 0;
}
```

```
[han@localhost test]$ ./test
fun() called
fun() const called
[han@localhost test]$
```

4.5.2 `const`修饰成员函数的重载

分两种情况，一种情况可以重载，另一种情况不可以重载。

5. `constexpr`的用法

`constexpr`主要用来将变量声明为该种类型，以便由编译器来验证变量的值是否是一个常量表达式。声明为`constexpr`的变量一定是一个常量，而且必须使用常量表达式来初始化。

```
constexpr int size = size() //注意，只有当size是一个constexpr函数时才是一条正确的声明
```

可以不可以写出一种函数，它既可以在编译期运行也可以在运行期运行，C++11引入的`constexpr`关键字很好的解决了这个问题。

> 尽管编译器运算会延长我们的编译时间，但是我们有的时候会利用它来加快程序的运行速度，但是在使用的时候，我们应该抱着谨慎的态度。有些人说，反正`constexpr`函数在运行期和编译器都可以执行，那我们为什么不可以给每一个函数都加上`constexpr`呢？我对此观点持保留意见，因为它会让我们的代码中充斥着不必要的关键字，影响阅读不说，他到底给我们编译器带来的好处能不能将坏的影响抵消掉还是要好好权衡的。

by刘元老师:

以上功能仅仅是`constexpr`的用法之一，但是这并不是我们创建这个关键字的目的，它将常量给固定了，并且赋予了常量数据类型，我们在C中，想要写常量，我们可以使用`const`，但是其实他并没有真正的固定下来，我们是可以对该常量进行修改的。

比如说:

```cpp
#include <stdio.h>

int main()
{
    const int a = 5;
    int *p = &a;
    *p = 6;

    printf("%d",*p);
    printf("%d",a);
    return 0;
}
```

最终的输出结果为：

```
66
```

我们可以发现我们是可以对常量进行修改的。

我们进行反汇编：

但是如果到了C++我们想要去定义一个：

- 不可更改的常量；
- 常量需要有类型；

我们不能简单的使用宏来实现，这个时候就得需要`constexpr`对于可以确定的类型，在编译期间直接给我们构造好，固定住，我们使用的时候，依然可以让座对应的类型使用，但是我们不能去修改他。

6. 为什么尽量不要使用`using namespace std`

其实底线就一条：如果你的头文件(`*.cpp`、`*.hpp`)又被外部使用，则尽量不要使用任何`using`语句引其他命名空间或者其他命名空间中的标识符。因为这样做可能会给使用你的头文件的人添加麻烦。更何况头文件之间都是相互套用的，假如说人人都在头文件中包含了若干个命名空间，到了第N层以后突然发现了一个命名冲突，这得往前回溯多少层才可以找到冲突。然而这个冲突本来是可以避免的。

其实在源文件`*.cpp`里面怎么使用`using`都是没有关系的，因为`*.cpp`里面的代码不影响到别人。甚至如果你的头文件仅仅是自己使用话，那么`using`也是没有问题的，但是为了养成良好的习惯，很多人仍然建议不要随便的使用`using`，以防写顺手。

7. 关于类声明参数`explicit`

```cpp
/*************************************************************************//**
 * \file   Test.cpp
 * \brief  Test the function of explicit!
 *
 * \author Kirito
 * \date   December 2022
 ****************************************************************************/
#include "A.h"

#include <iostream>

void dosomething(A a)
{
    std::cout << "Test the function of explicit! \n";
}

int main()
{
    A a;
    dosomething(a);
    /// 隐式转换发生在此处，如果我们传参传进来一个int类型的变量，函数会先判断可不可以对其隐式转换
    /// 为相应的变量类型，然后再去执行函数的功能；
    /// 我们也可以去为我们自定义的类，说明它可不可以进行隐式类型转换，如果构造函数声明为explicit
    /// 的话，就是告诉编译器，该处不可以进行隐式类型转换，但是并不影响其显式转换。
    dosomething(14);

    std::cout << "Hello World!\n";
}
```

```cpp
/*************************************************************************//**
 * \file   A.h
 * \brief  the class of A
 *
 * \author Kirito
 * \date   December 2022
 ****************************************************************************/

#pragma once
class A
{
protected:
    int _a;
public:
    // explicit A(int x = 0){};
    A(int x = 0) {}
};
```

8. 类

8.1 类的基本知识

- 对于使用`struct`和`class`关键字，使用`class`和`struct`定义类唯一的区别就是默认访问权限，`struct`的默认访问权限是`public`而`class`的默认访问权限是`private`。
- 类是允许其他类或者函数访问它的非公有成员的，方法是令其他类或者函数成为他的**友元**。

8.2 定义基类和派生类

- 作为继承关系中根节点的类往往都会定义一个虚析构函数。
- 基类中的成员函数分为两种：一种是基类希望其派生类进行覆盖的函数；另有一种是基类希望派生类直接继承而不要改变的函数。对于前者，我们往往将其定义为**虚函数**。当我们使用指针或者引用调用虚函数时，该调用将被动态绑定。根据引用或指针绑定的对象类型不同，该调用将会执行基类的版本，也可能执行某一个派生类的版本。

> **批注：**
>
> 动态绑定就是根据传进来的对象是基类的对象还是派生类的对象，来决定执行哪一个版本。

- 派生类对象以及派生类对象向基类的类型转换，就是我们可以将基类的指针或者引用绑定到派生类对象中的基类部分上。

```
Quote item;  // 基类对象
Bulk_quote bulk; // 派生类对象

Quote *p = &item; // p指向Quote对象
p = &bulk;        // p指向bulk的Quote部分
Quote &r = bulk;  // r绑定到bulk的Quote部分
```

8.3 关于构造函数

对于一个普通的类来讲，必须定义他自己的构造函数，***因为编译器只有在发现类内没有定义任何构造函数的时候，才会为我们生成一个默认的构造函数，一旦说我们定义了一个构造函数，无论你定义什么构造函数，编译器就会认为你要自己去构造，就不会自己生成指定过的构造函数。***

C++11标准中，我们可以使用`default`关键字来指定我们需要默认的行为，比如说我们需要默认的构造函数，就可以通过***该关键字来要求编译器生成构造函数。***

```
Sales_data = default;
```

----------***构造函数中的初始值列表***----------

构造函数中的初始值列表指的就是我们构造函数中的冒号***表达式***，冒号表达式的部分就相当于定义类内的变量的同时进行初始化，因为一般情况下我们定义变量的时候习惯于立即对其进行初始化，而非说你定义一个`int`的类型，然后再说去初始化其值：

```
int a;
a = 10;
```

所以说，我们定义一个类变量的时候，其初始化是在冒号表达式结束的时候结束的，***构造函数体内的表达式是对变量进行赋值操作。***

这个时候就会引入一个小问题：***就是如果类内有`const`类型或者引用的话的问题。***

我们知道如果是`const`或者引用类型的话，我们必须定义其的时候进行初始化，***所以说该种类型的数据变量，我们必须在冒号表达式中对其进行初始化，如果在构造函数体中对其进行赋值操作是错误的。***

----------***构造函数初始值列表的书写顺序要求***----------

最好令构造函数初始值的顺序与成员声明的顺序保持一致，而且如果可能的话，尽量避免使用某一些成员初始化其他成员。

```
class X {
    int i;
    int j;
public:
    X(int val): j(val),i(j) {}
};
```

我们像上面的方式去初始化的话，会出现错误信息，***因为我们成员的初始化顺序与他们在类定义的出现顺序保持一致***，也就是说在上面的代码中，`i`会比`j`先初始化，这个时候我们就会发现问题所在，我们会发现`i`是使用`j`来初始化的，但是`j`此时并未完成初始化，所以说这里会报错。我们在书写初始化列表的时候尽量不要去用别的成员来初始化其他的成员，就是为了防止上面情况的发生。

除了上述情况，虽说初始值列表中初始值的前后关系不会影响实际的初始化顺序。

----------***委托构造函数***----------

C++11标准扩展了构造函数初始值的功能，就是我们可以在冒号表达式中调用其他的构造函数来实现自己的职责，这就是委托构造函数，

```
class Sales_data {
public:
    Sales_data(std::string a, unsigned cnt, double price) :
        bookNo(s),unit_sold(cnt), revenue(cnt * price) {}

    // 以下构造函数全部是委托构造函数
    Sales_data():Sales_data("",0,0) {}
    Sales_data(std::string s):Sales_data(s,0,0) {}
    Sales_data(std::istream& is):Sales_data() {read(is, *this);}
}
```

----------***类的隐式转换构造函数***----------

> 参看第7点知识点

----------***使用default和delete***----------

我们可以使用`default`和`delete`来通知编译器是否生成或者删除默认的构造函数、拷贝构造函数、析构函数、拷贝复制运算函数。

8.4 类的静态成员

我们通过在成员的声明之前加上关键字`static`使得其与类关联在一起，和其他成员一样，静态成员可以是`public`的或者`private`的。

关键就在于，类的静态成员仅仅和类有关，和对象个体无关。

静态成员函数不与任何对象绑定在一起，他们不包含`this`指针，所以说我们不能在`static`函数体内使用`this`指针。

> 另外我们知道类中的所有的函数单独独立存在的，就是说我们实例化出来的所有的对象访问的成员函数其实都是一个，我们调用的时候，是将指向对象的指针传进去。

9. NDEBUG预处理变量

我们在编译文件的时候，可以选择定义预处理变量：

```
$g++ -D NDEBUG main.cpp
```

该条命令的作用等价于在main.cpp文件的一开始写`#define NDEBUG`。

10. `size_t` 和 `int`

> 让我们从定义开始。`int`是基本的有符号整数类型，并且保证至少有16位宽。`std::size_t`被定义为一个无符号整数，有足够的字节来表示任何类型的大小[2]。这意味着除了在C++的实现中`int`和`size_t`的宽度相同外，`size_t`总是能够比`int`存储更多的数字。`int`和`size_t`具有相同宽度的系统很可能很难处理，但这可能也是使它们有趣的原因。由于`size_t`有能力表示所有类型的大小--从而表示数组和向量的索引--***人们倾向于使用`size_t`来表示索引***，因为他们有保证可以表示他们想要的大小或索引。

使用`size_t`可能会提高代码的可移植性、有效性或者可读性，或许可以同时提高这三者。

可读性：当你看到一个对象声明为`size_t`类型，你就马上知道它代表字节的大小或者数组索引，而不是一个错误代码或者是一个普通的算数值，另外其表示的范围更大，我们不需要担心大小不够的问题。

11. 使用尾置返回类型

我们知道数组的类型是由数组的维数和数据的类型所组成，比如说下面示例：

```
int a[10];
// 该处声明了一个类型为int [10]的变量；
```

虽然说函数不可以返回数组，但是我们是可以返回指向数组的指针的，我们是如何声明数组的指针的呢?

```
typedef int arrT[10];
// 利用类型别名，我们就可以声明出数组的别名为int[10];
using arrT = int[10];
// 我们还可以使用新特性中的using,也相当于声明了一个别名

arrT* func(int i);

//该处声明了一个函数，该函数的返回值是arrT* ，即函数的指针

```

如果我们不使用这些别名去定义一个返回值类型为`int [10]`的话，我们要想定义一个函数就得像下面这样声明:

```
Type (*function(parameter_list)) [dimension]
```

具体例子:

```
int (*func(int i)) [10];
```

按照以下的顺序来理解:

- `func(int i)`是调用函数传进来的参数;
- `*func(int i)`意味着我们可以利用*运算符来获得一个变量;
- `*func(int i) [10]`意味着我们执行*运算符之后将会得到一个大小为10的数组;
- `int (*func(int i)) [10]`意味着数组中的元素是`int`类型,

我们应该可以看到这样声明的话会十分的麻烦，并且不容易让人理解，所以说，我们引进了尾置返回类型的方法来完整的表示一个函数

```
auto func(int i) -> int (*) [10];
// 该种声明方法，auto仅仅是占位符号,使用`->`指明真正的返回类型为int(*)[10];
```

11. OOP的核心思想是数据抽象、继承、和动态绑定

11.1 关于动态绑定

通常情况下，如果我们想要把引用或者指针绑定到一个对象上的话，则引用或者指针的类型应该与对象的类型保持一致。这想一想也是一定的，但是存在继承关系的类是一个重要的例外：***我们可以将基类的指针后者引用绑定到派生类的对象中。***

```
// 例如，Bulk_quote是Quote的一个派生类，那么下面这些操作是合法的
Bulk_quote bulk;
Quote* quote = & bulk;
Quote& quote1 = bulk;
```

可以将基类的指针或者引用绑定在派生类对象上意味着:

当使用基类的指针或者引用的时候，实际上我们并不清楚该引用或者指针所绑定对象的真实类型。该对象可能是基类的对象，也可以是派生类的对象。

这就涉及到动态类型和静态类型。

> 和内置指针一样，智能指针类也是支持派生类向基类的类型转换，这意味着我们可以将一个派生类对象的指针存储在一个基类的智能指针里面。

动态绑定就是在运行的时候，函数才可以知道传进来的参数的类型到底是什么。

11.2 虚函数

虚函数的一个关键就是可以利用动态绑定来决定我们去执行基类与派生类中的函数的版本。

> 就比如我们在基类中声明一个虚函数并定义，在派生类中将该虚函数进行重写，然后我们这里有一个函数，函数的参数是基类函数的引用或者指针，这样的话，如果我们传进去的是基类对象，那么里面关于对象成员函数的调用，全都调用基类的版本。但是如果我们传进去一个派生类参数，那么就会动态绑定到派生类重写的那个函数的版本。

关于虚函数重载的问题，一般来将，我们如果需要重新定义相应的虚函数，直接重载定义即可。***但是难免派生类中如果定义了一个函数与基类中的虚函数名字相同但是形参列表不同的话，这仍然是合法的的行为***。这个如果继续下去的话，就会导致不可预知的错误，所以说我们引入了`override`关键字，该关键字会通知编译器该函数是要覆盖掉基类中的虚函数，编译器会去检查是否覆盖，如果参数错误的话，就会报错，提醒程序员有地方写错了。

11.3 抽象函数

有的时候我们需要虚函数和各种派生类来完成指定的功能，***但是又不希望我们去实例化我们的基类，我们只希望去实例化派生出来的类。***这个时候我们只需要将基类中虚函数的定义给去除掉：

```
virtual void test() = 0;
```

这个时候，很显然，我们基类中的虚函数没有定义，所以说我们不能实例化一个含有纯虚函数的基类，该基类此时被称为抽象类。

11.4 虚析构函数

当我们`delete`一个动态分配的对象的指针时将执行析构函数。如果该指针指向继承体系中的某一个类型，则有可能出现指针的静态类型与被删除对象的动态类型不符合的情况。

比如说我们定义一个Quote*类型的指针，则该指针有可能实际上指向的是Bulk_quote类型的对象，如果这样的话，编译器就应该清楚它应该执行的是Bulk_quote的析构函数。***和其他函数一样，我们需要通过在基类中将析构函数定义为虚函数以确保执行正确的析构函数版***。否则的话，我们可能会错误的执行析构函数。

> 所以说我们在继承这一块，我们不仅需要在基类中声明必要的虚函数，还需要声明虚析构函数，以保证我们之后析构对象的时候，可以析构正确。

12. lambda表达式

一个`lambda`表达式表示一个可以调用的代码单元，我们可以将其理解为一个未命名的内联函数，因为如我们所见，`lambda`表达式总是出现在我们的函数内部，即内联函数。

与任何函数一样，`lambda`***表达式具有一个返回类型、一个参数列表和一个函数体。***

```
[capture list] (parameter list) -> return type { function body}
// capture list，捕获列表，就是lambda表达式所在函数中定义的局部变量的列表(通常为空)
// parameter list，就是传进来参数的列表
// return type，就是函数的返回类型，注意lambda表达式的返回类型书写的时候使用表达式后置的方法，就将对象的类型写在参数的后面，前面的其实也是使用auto作为占位符
```

我们在书写lambda表达式的时候，我们可以忽略参数列表和返回值类型，但是必须包含捕获列表和函数体.

```
auto f = [] { return 42;}
cout << f() << endl;// 打印42
```

当我们书写lambda表达式的时候，如果忽略返回类型，编译器会根据函数体内的代码推断出返回的值的类型，如果没有返回`return`语句的话，那么返回类型就是void。

```
// 以下内容是一个简单的程序
#include <iostream>

int main()
{
    auto f = [] { return 42;};
    std :: cout << f() << std:: endl;
  return 0;
}
```

```
// 以下内容是编译器看到的内容
#include <iostream>

int main()
{

  class __lambda_5_11
  {
    public:
    inline int operator()() const
    {
      return 42;
    }

    using retType_5_11 = auto (*)() -> int;
    // 我们可以从这里看到，编译器自动识别其返回类型，并且将retType_5_11定义为函数指针，该函数的返回类型是int类型
    inline operator retType_5_11 () const noexcept
    {
      return __invoke;
    };

    private:
    static inline int __invoke()
    {
      return __lambda_5_11{}.operator()();
    }

```

```
    public:
    // inline /*constexpr */ __lambda_5_11(__lambda_5_11 &&) noexcept = default;

  };

  __lambda_5_11 f = __lambda_5_11(__lambda_5_11{});
  std::cout.operator<<(f.operator()()).operator<<(std::endl);
  return 0;
}
```

通过编译器看到的内容，我们可以看到编译器是将lambda表达式定义为一个类，然后调用该类，已完成输出。

如果我们像下面这样定义lambda表达式：

```
#include <iostream>

int main()
{
    auto f = [] { std::cout << 43;};
    f();
    return 0;
}
```

编译器看来是这样的：

```
#include <iostream>

int main()
{

  class __lambda_5_11
  {
    public:
    inline void operator()() const
    {
      std::cout.operator<<(43);
    }

    using retType_5_11 = auto (*)() -> void;
    // 我们可以看到编译器识别将其返回值定义为void
    inline operator retType_5_11 () const noexcept
    {
      return __invoke;
    };

    private:
    static inline void __invoke()
    {
      __lambda_5_11{}.operator()();
    }

    public:
```

```
    // inline /*constexpr */ __lambda_5_11(__lambda_5_11 &&) noexcept =
default;

  };

  __lambda_5_11 f = __lambda_5_11(__lambda_5_11{});
  f.operator()();
  return 0;
}
```

13. 左值引用和右值引用

> 参考文章https://paul.pub/cpp-value-category/
>
> 对于左值和右值，我们可以简单的理解为：***左值对应了具有内存地址的对象，而右值对象仅仅是临时使用的值。***
>
> ```
> // 例如下面例子中，很显然s1和s2是左值，因为其具有相应的内存地址，"Hello"和"World"很显然是一个临时使用的值，用来初
> // 始构造我们的变量
> std::string s1 = "Hello";
> std::string s2 = "World";
> // s3是左值，而右边的表达式虽然其中每一个变量是左值，但是组合起来就变成了右值
> std::string s3 = s1 + s2;
>
> ```

在C++之前，引用分为`const`引用和非`const`引用。这两种引用在C++中都称为左值引用(rvalue reference)。

注意：我们是无法将非`const`左值引用指向右值的。

```
// 下面的代码是不会通过编译的
#include <iostream>
#include <string>
using namespace std;
int main()
{
        string s1 = "Hello, ";
        string s2 = "world! ";
        string &s3 = s1 + s2;
        cout << s3 << endl;
        return 0;
}
```

编译器会报错：

```
.\main.cpp: In function 'int main()':
.\main.cpp:11:18: error: cannot bind non-const lvalue reference of type
'std::__cxx11::string&' {aka 'std::__cxx11::basic_string<char>&'} to an
rvalue of type 'std::__cxx11::basic_string<char>'
  string &s3 = s1 + s2;
```

意思翻译翻译过来也就是你不能将非`const`左值引用指向右值的。

但是，`const`类型的左值引用是可以绑定到右值的。也就是说，下面的代码是可以通过编译的：

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
        string s1 = "Hello, ";
        string s2 = "world! ";
        const string &s3 = s1 + s2;
        cout << s3 << endl;
        return 0;
}
```

不过，由于这个引用是const的，因此你无法修改其值的内容。

C++11新增了右值引用，左值引用的写法是`&`，右值引用的写法是`&&`。

右值是一个临时的值，右值引用是指向右值的引用。右值引用延长了临时值的生命周期，并且允许我们修改其值。

例如：

```
std::string s1 = "Hello ";
std::string s2 = "world";
std::string&& s_rref = s1 + s2;    // the result of s1 + s2 is an rvalue
s_rref += ", my friend";           // I can change the temporary string!
std::cout << s_rref << '\n';       // prints "Hello world, my friend"
```

右值引用使得我们可以创建出以此为基础的函数重载，例如：

```
void func(X& x) {
    cout << "lvalue reference version" << endl;
}

void func(X&& x) {
    cout << "rvalue reference version" << endl;
}
```

```
X returnX() {
    return X();
}

int main(int argc, char** argv) {
    X x;
    func(x);
    func(returnX());
}
```

输出结果为：

```
lvalue reference version
rvalue reference version
```

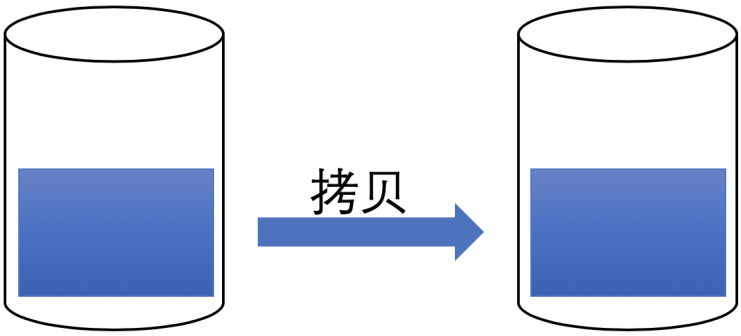
13.1 移动语义

我们知道，在C++中，我们可以为类定义拷贝构造函数和拷贝赋值运算符。

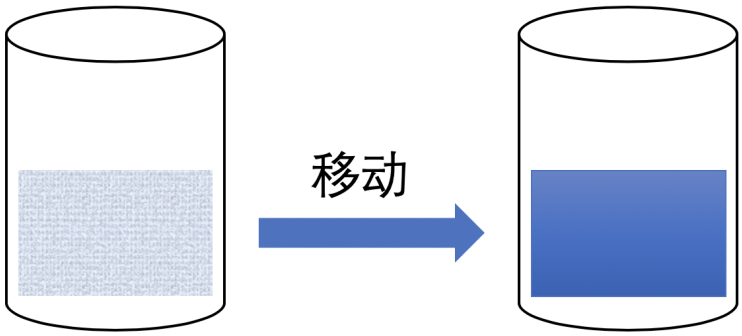
```
class X
{
public:
    X(const X& other) // copy constructor
    {
        m_data = new int[other.m_size];
        std::copy(other.m_data, other.m_data + other.m_size, m_data);
        m_size = other.m_size;
    }

    X& operator=(X other) // copy assignment
    {
        if(this == &other) return *this;
        delete[] m_data;
        m_data = new int[other.m_size];
        std::copy(other.m_data, other.m_data + other.m_size, m_data);
        m_size = other.m_size;
        return *this;
    }

    X& operator=(const X& other) // copy assignment
    {
        if(this == &other) return *this;
        delete[] m_data;
        m_data = new int[other.m_size];
        std::copy(other.m_data, other.m_data + other.m_size, m_data);
        m_size = other.m_size;
        return *this;
    }

private:
    int*   m_data;
    size_t m_size;
};
```

> 当然，如果你为类定义了拷贝构造函数和拷贝赋值运算符，你通常还应当为其定义析构函数。这称之为Rule of Three。

拷贝意味着会将原来的数据复制一份新的出来。这么做的好处是：新的数据和原先的数据是相互独立的，修改其中一个不会去影响另一个。但是坏处是：这么做会消耗运算时间和存储空间。例如你有一个包含了10^10个元素的集合数据，将其拷贝一份就不那么轻松了。



但是移动操作就会轻松很多，因为他不涉及新数据的产生，仅仅是将原先的数据更改其所有者。



在C++11中，我们可以这样为类定义移动构造函数和移动赋值运算符：

```
X(X&&);
X& operator=(X&&);
```

我们继续上面的定义：

```
X(X&& other)
{
    m_data = other.m_data;
    m_size = other.m_size;
    // 注意下面的语句，就相当于转移数据所有权，局部变量other已经没有用了
    other.m_date = nullptr;
    other.m_size = 0;
}

X& operator=(X&& other)
{
    if(this == &other) return *this;

    delete[] m_data;

    m_data = other.m_data;
    m_size = other.m_size;

    other.m_data = nullptr;
    other.m_size = 0;

    return *this;
}
```

现在，该类有了拷贝和移动两种操作，那编译器如何知道该选择哪个呢？***答案是，根据传入的参数类型：如果是左值引用，则使用拷贝操作；如果是右值引用，则使用移动操作。***

```
X createX(int size)
{
  return X(size);
}

int main()
{
  X h1(1000);                // regular constructor
  X h2(h1);                  // copy constructor (lvalue in input)
  X h3 = createX(2000);      // move constructor (rvalue in input)

  h2 = h3;                   // assignment operator (lvalue in input)
  h2 = createX(500);         // move assignment operator (rvalue in input)
}
```

还有一点就是，如果是左值，我们也是可以调用移动操作的，我们此时需要借用`std::move`。

13.1 线程的所有权的转移：

> ***该处的`std::move`我们可能在C++11线程的学习中使用很多，因为有的时候由于我们的需求，需要将一个线程的所有权转交给另一个线程。这个时候就需要用到`std::move`来帮助我们实现这个目的。***

```
void some_function();
void some_other_function();

std::thread t1(some_function);

std::thread t2 = std::move(t1);

t1 = std::thread(some_other_function);
// 注意，这里我们将临时产生的线程的控制权转移给了t1,但是我们并没有显示的去调用`std::move()`转移其所有权，这是因为，所有者是一个临时对象，是一个右值，移动赋值操作符会隐式的调用。
```

13.2 智能指针的控制权的转移：

> ***还有我们学习智能指针的时候，知道智能指针`unique_ptr`也成为独享指针，即不能同时有多个智能指针指向同一块内存，那如果我们在函数之间传递智能指针怎么办？***

```
void pass_up(unique_ptr<int> up)
{
 cout << "In pass_up: " << *up <<endl;
}

void main()
{
 auto up = make_unique<int>(123);
 pass_up(up);
}
```

上述代码会出现错误，原因在于我们在传递指针的时候，会有一个复制up的操作，显然这个操作是不允许的，所以会报错。

这个时候我们可以选择直接传给函数指针指向的资源：

`pass_up(*up);`

除此之外，还可以使用第二种方法：

`pass_up(up.get());`

其中`up.get()`获得的是资源的裸指针。

以上方法仅仅让我们去访问对应的资源，但是如果我们想要通过函数直接改变`unique_ptr`本身怎么办？

我们可以将函数的参数设置为智能指针的引用：

```
void pass_up(unique_ptr<int> &up)
{
 cout << "In pass_up: " << *up <<endl;
 up.reset();
}

void main()
{
 auto up = make_unique<int>(123);
 pass_up(up);
 if(up == nullptr) cout << "up is reset."<<endl;
}
```

最后还有一种方法就是利用`std::move`来转移`unique_ptr`的控制权。转移之后，原来的智能指针会变成`nulptr`

13.3 移动语义给我们带来的好处

有了右值引用和移动操作之后，STL里面的集合操作就会变得更加的高效，例如：

```cpp
std::string str = "Hello";
std::vector<std::string> v;

v.push_back(str);                // ①
v.push_back(std::move(str));     // ②
```

这里的1是将复制一个字符串添加到集合当中，而2则是将已有的对象移动进集合当中，如果我们移动个上千万个数据，这自然是更高效的。

14. C++ 三法则和五法则

14.1 C++ 三法则

三法则讲述的是，如果一个类定义了以下任何一项，那么它可能应该明确定义所有三个：

- 析构函数，***destructor***；
- 拷贝构造函数，***copy constructor***；
- 拷贝复制运算符，***copy assignment operator***；

为什么呢？

其实都是为深拷贝所服务的，如果我们想要深拷贝的话，我们就需要去自定义一个拷贝构造函数和拷贝复制运算符函数。既然我们涉及到深拷贝了，那么一定存在指针，所以需要析构函数来释放内存，否则就会导致内存泄漏。

14.2 C++ 五法则

五法则讲述的则是：

- 析构函数，***destructor***；
- 复制构造函数，***copy constructor***；
- 复制赋值运算符，***copy assignment operator***；
- 移动构造函数，***move constructor***；
- 移动赋值运算符，***move assignment operator***；