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;
}
```

```
/* 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：

```
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：

```
#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：

```
#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:

```
#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`

```
/*****************************************************************//**
 * \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";
}
```

```
/*****************************************************************//**
 * \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。

11.5 static成员初始化

一般情况下，一个`static`类成员是静态分布的，而不是每一个类对象的一部分。也就是说***static成员声明充当类外定义的声明，也可以这样理解，`static`类成员的定义和声明是分开的。***

```
class Node{
    //...
    static int node_count;    // 声明
};

int Node::node_count = 0; // 定义
```

当然了，在极少数的情况下，在类内声明初始化`static`成员也是有可能的。条件是`static`成员必须是整型或者枚举类型的`const`，或者是字面值类型的`constexpr`。

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;
    // 就相当于 typedef int (* retType_5_11)()
    // 编译器看到的都是using retType_5_11 = auto (*)() -> int
    // 这里的auto声明是函数的返回值类型后置写法，auto仅仅是一个占位符，后面的int才是函数的返回类型
    // 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;
}
```

12.1 参考刘元老师笔记（http://mtw.so/6lHAIT）

在泛型算法中，我们可以传进来一个***谓词***，来帮助泛型算法更好的进行运算。

例如：我们可以在`find_if`函数中传入第三个参数，这个参数是一个接受且只接受一个参数的可调用对象（这里需要注意，第三个参数是只能接受一个参数的可调用对象，如果我们传进来含有多余参数的可调用对象，就会报错），其可以帮助`find_if`判断是否找到了我们需要的元素：

```
bool getOdd(int numToPredicate) { return numToPredicate == 0; }
int main()
{
    vector<int> vec({1, 2, 3});
    find_if(vec.cbegin(), vec.cend(), getOdd);
}
```

但是有的时候，我们需要向函数里面传递两个参数才可以确定我们是否找到了我们需要的元素，在不进行额外的操作的时候，我们是无法使用的，这个时候，***我们就可以使用到lambda表达式，因为lambda表达式可以捕获局部变量，这样的话我们就实现了往里面传入多个参数的功能。***

12.2 值的捕获

类似于参数传递，变量的捕获方式也 可以是值或者引用。采用值捕获的前提是变量可以拷贝。与参数不同，被捕获的变量的值是在lambda创建时的拷贝，而不是调用的时候拷贝，编译器会在编译的时候，创建一个相对应的类，然后在类中重载调用运算符（）重载的时候，其中写的是我们lambda函数的主体部分。比如说下面的例子：

```
// 我们书写了含有lambda表达式的式子
stable_sort(words.begin(), words.end(),
    [](const string& str1, const string& str2)
        { return str1.size() < str2.size(); });

// 编译器看来是这样的：
class ShorterString
{
public:
    bool operator()(const string& str1, const string& str2) const
    {
        return str1.size() < str2.size();// 我们所写lambda表达式的主体
  }
};
```

12.3 值捕获

值的捕获又分为两种，一种是lambda表达式中可以修改的，另一种是lambda表达式中不可以修改的。这二者通过`mutable`关键字来进行区分：

```
// 下面lambda表达式内是不可以修改捕捉到的值的
int test;
auto f = [test] {test = 5; };
// 上面的代码会报错，原因是`cannot assign to a variable captured by copy in a non-mutable lambda`
// 翻译过来就是无法赋值给不可变lambda中由copy捕获的变量

// 下面的代码成功运行
int test;
auto f = [test]() mutable{ test = 5; }
```

这大概已经是`mutable`的使用方法了，但是为什么呢?

原因其实也很简单，我们接下来来看一下：

我们将下面的代码以编译器的视角来看一下, 我们先查看不加关键字`mutable`的情况，为什么选择下面这么长的代码，因为这段代码在测试的时候发现了右值引用相关的知识点，很有趣的：

```
// 测试代码，测试不加关键字mutable的情况
#include <iostream>
using namespace std;
class Test
{
public:
        Test(int val) : _val(val) { cout << "Test::Test()" << endl; }
        Test(const Test& other) : _val(other._val) { cout << "Test::Test(const Test&)" << endl; }
        ~Test() { cout << "Test::~Test()" << endl; }
```

```
        Test& operator=(const Test& other) { this->_val = other._val; cout <<
"Test::operator=(const Test& other)" << endl; return *this; }
private:
        int _val;
};

int main()
{
        Test test(5);
        auto func_value = [test]() { return test; };
        func_value();
}
```

下面的代码是编译器看见的代码：

```
#include <iostream>

using namespace std;

class Test
{
  public:
  inline Test(int val)
  : _val{val}
  {
    std::operator<<(std::cout, "Test::Test()").operator<<(std::endl);
  }
  inline Test(const Test & other)
  : _val{other._val}
  {
    std::operator<<(std::cout, "Test::Test(const
Test&)").operator<<(std::endl);
  }

  inline ~Test() noexcept
  {
    std::operator<<(std::cout, "Test::~Test()").operator<<(std::endl);
  }
  inline Test & operator=(const Test & other)
  {
    this->_val = other._val;
    std::operator<<(std::cout, "Test::operator=(const Test&
other)").operator<<(std::endl);
    return *this;
  }
  private:
  int _val;
};

int main()
{
  Test test = Test(5);

  class __lambda_21_27
  {
```

```
    public:
    inline Test operator()() const  //这一行是关键所在
    {
      return Test(test);
    }
    private:
    Test test;
    public:
    // inline __lambda_21_27 & operator=(const __lambda_21_27 &) /* noexcept */
= delete;
    __lambda_21_27(const Test & _test)
    : test{_test}
    {}
  };
  __lambda_21_27 func_value = __lambda_21_27{test};
  func_value.operator()();
  return 0;
}
```

从上面的代码中我们可以看到编译器是先将lambda表达式先转换成一个类，然后通过实例化该类以及调用相关的函数从而实现了lambda表达式的功能。

好了我们回到为什么不加关键字`mutable`，lambda表达式就无法修改捕获到的值呢?

从上面的代码中我们可以看到，如果我们捕捉一个变量的话，lambda相应的类就会生成一个相应的私有成员变量，然后其拷贝构造函数声明方式为：

```
__lambda_21_27(const Test & _test)
    : test{_test}
    {}
```

这里很正常，因为我们平时声明拷贝构造函数就是声明为`const`的引用,然后将其值再初始化给`test`，***然后还有就是，我们捕捉到的值是左值，并不是右值。这一点其实就不用看，如果你和我意见不一样，在下面的引用捕获你可以体会到是左值。***

关键看第40行的代码：

```
    public:
    inline Test operator()() const  //这一行是关键所在
    {
      return Test(test);
    }
```

我们可以看到这里重载了调用运算符`()`，并且将其声明为`const`，这就是一切的来源了，就是因为其声明为`const`的原因，才导致我们不可以在lambda表达式中去修改相应的变量，虽然说，我们已经通过copy得到了一份副本，但是对于这一份副本，我们声明为`const`,就是不可以修改。

好，看到这里你应该可以猜到如果加上关键字`mutable`会发生什么了，没错就是将重载函数的`const`声明去掉了：

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

```
class Test
{
public:
        Test(int val) : _val(val) { cout << "Test::Test()" << endl; }
        Test(const Test& other) : _val(other._val) { cout << "Test::Test(const
Test&)" << endl; }
        ~Test() { cout << "Test::~Test()" << endl; }
        Test& operator=(const Test& other) { this->_val = other._val; cout <<
"Test::operator=(const Test& other)" << endl; return *this; }
private:
        int _val;
};
int main()
{
        Test test(5);
        auto func_value = [test]() mutable { test = Test(6); };
        func_value();
}
```

编译器看到的是下面代码：

```
#include <iostream>

using namespace std;
class Test
{
  public:
  inline Test(int val)
  : _val{val}
  {
    std::operator<<(std::cout, "Test::Test()").operator<<(std::endl);
  }
  inline Test(const Test & other)
  : _val{other._val}
  {
    std::operator<<(std::cout, "Test::Test(const
Test&)").operator<<(std::endl);
  }
  inline ~Test() noexcept
  {
    std::operator<<(std::cout, "Test::~Test()").operator<<(std::endl);
  }
  inline Test & operator=(const Test & other)
  {
    this->_val = other._val;
    std::operator<<(std::cout, "Test::operator=(const Test&
other)").operator<<(std::endl);
    return *this;
  }

  private:
  int _val;
};
int main()
{
```

```
  Test test = Test(5);

  class __lambda_21_27
  {
    public:
    inline /*constexpr */ void operator()()  // 我在这里我在这里
                                                                                // 我
们在这里可以很清晰的看到，该函数没有const所以我们是可以对lambda中的
                                                                                  //
变量进行修改的
    {
      test.operator=(Test(Test(6)));
    }
    private:
    Test test;
    public:
    // inline __lambda_21_27 & operator=(const __lambda_21_27 &) /* noexcept */
= delete;
    __lambda_21_27(const Test & _test)
    : test{_test}
    {}
  };
  __lambda_21_27 func_value = __lambda_21_27{test};
  func_value.operator()();
  return 0;
}
```

重点在于第38行：

```
    inline /*constexpr */ void operator()()  // 我在这里我在这里
                                                                                // 我
们在这里可以很清晰的看到，该函数没有const所以我们是可以对lambda中的
                                                                                  //
变量进行修改的
    {
      test.operator=(Test(Test(6)));
    }
```

我们也可以看到这里对`test`进行了修改，怎么修改的呢，是通过调用重载赋值运算符`=`来修改的，这里有一处代码实践了之前学到的知识：

> 我们是可以通过`const`引用来引用右值的！！！

我们可以看到重载赋值运算符函数里面的参数`Test(Test(6))`,首先，我们是先构造一个临时变量这才是右值，然后为什么外面又要嵌套一层呢，因为这里是右值，编译器又将其强转化为Test类型的变量，这里是编译器的处理。我们其实也可以直接使用`Test(6)`进行传参，因为我们是可以通过`const`引用来引用右值的：

```
// 我们可以看到对应的重载赋值运算符的参数类型就是const 的引用
// 这下我们对为什么重载赋值运算符的参数是 const Test & other有了进一步的认识
// 因为我们这样可以实现接受右值的传参
inline Test & operator=(const Test & other)
  {
    this->_val = other._val;
    std::operator<<(std::cout, "Test::operator=(const Test&
other)").operator<<(std::endl);
    return *this;
  }
```

12.4 引用捕获

看完上面的简述之后，我们接下来继续看另一种捕获，引用捕获：

```
#include<iostream>
#include<string>
int main()
{
    std::string test;
    auto f = [&test]{ test = "test";};
    f();
    std::cout << test;
}
```

使用起来很方便，我们可以直接想引用正常变量的时候引用他。

接下来我们可以查看编译器是怎么看的：

```
#include<iostream>
#include<string>
int main()
{
  std::basic_string<char> test = std::basic_string<char>();

  class __lambda_6_13
  {
    public:
    inline /*constexpr */ void operator()() const // 这里虽然说是const修饰，但是这里
是引用，所以说我们可以对其进行修改

// 我们只是不可以修改引用指向哪里，但是并不代表我们不可以修改引用对

// 的值，当然了如果我们想要修改其引用指向哪里，一样的，添加关键字
                                                    // mutable
    {
      test.operator=("test");
    }

    private:
    std::basic_string<char> & test;

    public:
    __lambda_6_13(std::basic_string<char> & _test)
```

```
    : test{_test}
    {}

  };
  __lambda_6_13 f = __lambda_6_13{test};
  f.operator()();
  std::operator<<(std::cout, test);
  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
```

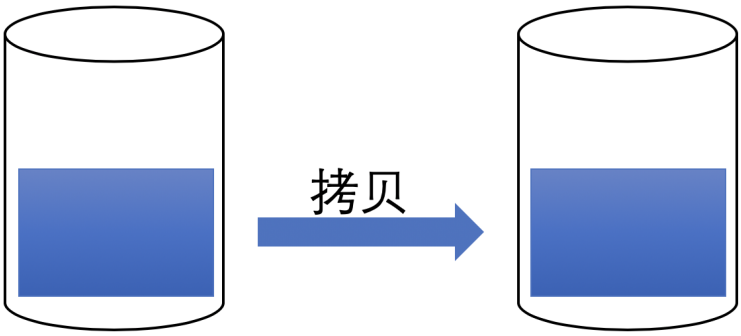
13.1 移动语义

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。

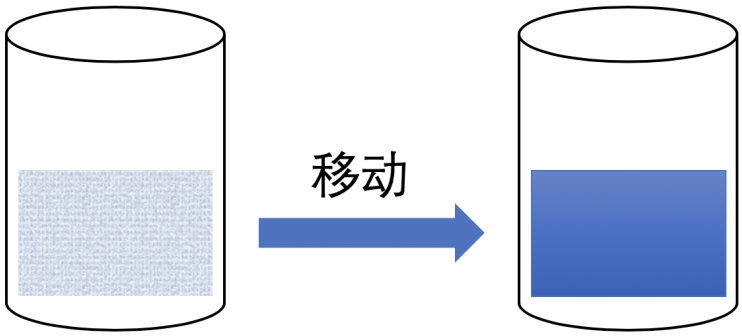
```
class X
{
public:
    X(const X& other) // copy constructor
    {
        m_data = new int[other.m_size];
        std::copy(other.m_data, other.m_data + other.m_size, m_data);
        m_size = other.m_size;
    }

    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： 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***;

15. 常量

C++支持如下两种不变概念。

- `const`: 大致的意思是"我承诺不会去改变这个值（***即我们不能通过该变量去修改对应的值***）"。主要用于说明接口，这样在于我们在把变量传入函数的时候就不必担心变量会在函数内被改变了。编译器负责确认并运行`const`的承诺。
- `constexpr`: 大致的意思是"在编译的时候求值"。***这才是真正的常量。作用是允许将数据置于只读内存（不太可能被破坏）中以及提高性能。***

```
const int dmv = 17; // dwv 是一个命名的常量
int var = 17;       // var不是常量
constexpr double max1 = 1.4*square(dwv); // 如果square(17)是常量表达式，则正确，这个需要square函数声明为constexpr
constexpr double max2 = 1.4*square(var); // 错误，因为var不是常量表达式
const double max3 = 1.4*square(var); // 正确，可以在运行的时候求值
double sum(const vector<double>&); // 此处声明的是一个函数，说明该函数不会去修改参数的值
vector<double> v{1,2,3,4,5}; // v不是常量
const double s1 = sum(v); // OK,在运行的时候求值
constexpr double s2 = sum(v); // 错误, constexpr需要在编译的时候求值，即所有的值都是常量表达式
```

如果某一个函数用在常量表达式中，即该表达式在编译的时候求值，则该函数必须定义为`constexpr`.例如：

```
constexpr double square(double x) { return x*x; }
```

当一个函数定义为`constexpr`的时候，该函数就是可以在编译器可以运行的函数，在编译器运行的函数，你想想，他的能耐能有多大，所以说，定位为`constexr`的函数的限制十分的多：

- 函数必须返回一个值，不能是void;
- 函数体内只能有一条语句，return；
- 函数调用之前必须被定义；
- 函数必须使用`constexpr`进行声明；

`constexpr`的作用是指示或者确保在编译的时候求值，而`const`的主要任务是规定接口的不可修改性。

16. noexcept的使用方法

> 搬运博客： https://blog.csdn.net/null_10086/article/details/119517852

16.1 C++98中的异常规范：

throw 关键字除了可以用在函数体中抛出异常，还可以用在函数头和函数体之间，指明当前函数能够抛出的异常类型，这称为异常规范，有些教程也称为异常指示符或异常列表。请看下面的例子：

```
double func1 (char param) throw(int);
```

函数 func1 只能抛出`int`类型的异常。如果抛出其他类型的异常，try 将无法捕获，并直接调用`std::unexpected`。

如果函数会抛出多种类型的异常，那么可以用逗号隔开，

```
double func2 (char param) throw(int, char, exception);
```

如果函数不会抛出任何异常，那么只需写一个空括号即可，

```
double func3 (char param) throw();
```

同样的，如果函数 func3 还是抛出异常了，try 也会检测不到，并且也会直接调用`std::unexpected`。

虚函数中的异常规范:

C++ 规定，派生类虚函数的异常规范必须与基类虚函数的异常规范一样严格，或者更严格。只有这样，当通过基类指针（或者引用）调用派生类虚函数时，才能保证不违背基类成员函数的异常规范。请看下面的例子：

```
class Base {
  public:
    virtual int fun1(int) throw();
    virtual int fun2(int) throw(int);
    virtual string fun3() throw(int, string);
};

class Derived: public Base {
  public:
    int fun1(int) throw(int);     //错！异常规范不如 throw() 严格
    int fun2(int) throw(int);     //对！有相同的异常规范
    string fun3() throw(string); //对！异常规范比 throw(int, string) 更严格
}
```

异常规范与函数定义和函数声明

C++ 规定，异常规范在函数声明和函数定义中必须同时指明，并且要严格保持一致，不能更加严格或者更加宽松。请看下面的几组函数：

```
// 错！定义中有异常规范，声明中没有
void func1();
void func1() throw(int) { }

// 错！定义和声明中的异常规范不一致
void func2() throw(int);
void func2() throw(int, bool) { }

// 对！定义和声明中的异常规范严格一致
void func3() throw(float, char *);
void func3() throw(float, char *) { }
```

异常规范在 C++11 中被摒弃

异常规范的初衷是好的，它希望让程序员看到函数的定义或声明后，立马就知道该函数会抛出什么类型的异常，这样程序员就可以使用`try-catch`来捕获了。如果没有异常规范，程序员必须阅读函数源码才能知道函数会抛出什么异常。

不过这有时候也不容易做到。例如，`func_outer()`函数可能不会引发异常，但它调用了另外一个函数`func_inner()`，这个函数可能会引发异常。再如，编写的一个函数调用了老式的一个库函数，此时不会引发异常，但是老式库更新以后这个函数却引发了异常。

其实，不仅仅如此，

1. 异常规范的检查是在运行期而不是编译期，因此程序员不能保证所有异常都得到了 catch 处理。
2. 由于第一点的存在，编译器需要生成额外的代码，在一定程度上妨碍了优化。
3. 模板函数中无法使用。比如下面的代码，

   ```
   template<class T>
   void func(T k) {
       T x(k);
       x.do_something();
   }
   ```

   赋值函数、拷贝构造函数和 do_something() 都有可能抛出异常，这取决于类型 T 的实现，所以无法给函数 `func` 指定异常类型。
4. 实际使用中，***我们只需要两种异常说明：抛异常和不抛异常，也就是 throw(...) 和 throw()。***

所以 C++11 摒弃了 throw 异常规范，而引入了新的异常说明符`noexcept`。

> 动态异常规定：就是列出函数中可能直接或者间接排除的异常

16.2 C++ 11中的异常规范

`noexcept`紧跟在函数的参数列表后面，他只用来表明两种状态：***"抛出异常和不抛出异常"***。

```
void func_not_throw() noexcept;// 保证不抛出异常
void func_not_throw() noexcept(true);// 和上面的式子是一样的

// 默认情况下都是要抛出异常的
void func_not_throw() noexcept(false); // 可能会抛出异常
void func_not_throw();
```

对于一个函数，使用`noexcept`时候的规范：

- `noexcept` 说明符号要么出现在该函数的所有声明语句和定义语句中，要么一次也不出现；
- 函数指针以及函数指针指向的函数必须具有一致的异常说明；

对于函数指针中的`noexcept`说明符号，请看下面的示例：

```
void (* func_not_throw)() noexcept(false);

// 此处的函数声明就是声明了一个指向可能抛出异常的函数

// 但是需要注意一点的是，我们不能在typedef或者类型别名的时候出现noexcept说明符号
// 比如说：
// typedef int (*pf)() noexcept;
// 上面的写法是错误的
```

- 在成员函数中，`noexcept` 说明符需要跟在 `const` 及引用限定符之后，而在 final、override 或虚函数的 =0 之前。
- 如果一个虚函数承诺了它不会抛出异常，则后续派生的虚函数也必须做出同样的承诺；与之相反，如果基类的虚函数允许抛出异常，则派生类的虚函数既可以抛出异常，也可以不允许抛出异常。

还有一点就是如果我们在声明了`noexcept`的函数中抛出异常的话，程序会直接调用`std::terminate`，并不能捕获到指定的异常。

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

void func_not_throw() noexcept {
    throw 1;
}

int main() {
    try {
        func_not_throw(); // 直接 terminate，不会被 catch
    } catch (int) {
        cout << "catch int" << endl;
    }
    return 0;
}

// 上面的程序中我们是不可以捕获到int的，因为我们声明的函数为不会产生异常，一旦产生异常就会直接调用std::terminate
// 所以说我们在使用noexcept说明符的时候需要格外的注意
```

`noexcept`说明符还有另外的一种使用方法：

`noexcept`运算符进行编译时候检查，如果说表达式声明为不抛出任何异常就返回`true`，简单地说就是为了判断某一个函数是否声明`noexcept`.

比如说下面的示例：

```cpp
void f() noexcept {
}

void g() noexcept(noexcept(f())) { // g()是否是 noexcept 取决于 f()
    f();
}
// 由于f函数是noexcept类型的，所以说该运算符会返回true，那么g函数也就会被声明为noexcept(true)
// 即不会抛出异常
```

还有很重要的是***对于析构函数，我们都是默认为`noexcept`类型的。C++ 11 标准规定，类的析构函数都是 `noexcept` 的，除非显示指定为 `noexcept(false)`。***

```cpp
class A {
  public:
    A() {}
    ~A() {} // 默认不抛出异常
};

class B {
  public:
    B() {}
    ~B() noexcept(false) {} // 可能会抛出异常
};
```

在为某个异常进行栈展开的时候，会依次调用当前作用域下每个局部对象的析构函数，如果这个时候析构函数又抛出自己的未经处理的另一个异常，将会导致 `std::terminate`。所以析构函数应该从不抛出异常。

16.3 `noexcept`的使用建议

我们所编写的函数**默认都不使用**，只有遇到以下的情况你再思考是否需要使用，

1. **析构函数**

   这不用多说，必须也应该为 `noexcept`。

2. **构造函数（普通、复制、移动），赋值运算符重载函数**

   尽量让上面的函数都是 `noexcept`，这可能会给你的代码带来一定的运行期执行效率。

3. **还有那些你可以 100% 保证不会 throw 的函数**

   比如像是`int`，`pointer` 这类的 `getter`，`setter` 都可以用 `noexcept`。因为不可能出错。但请一定要注意，不能保证的地方请不要用，否则会害人害己！切记！如果你还是不知道该在哪里用，可以看下准标准库 Boost 的源码，全局搜索 `BOOST_NOEXCEPT`，你就大概明白了。

16.4 使用`noexcept`说明符的好处

1. **语义**

   从语义上，`noexcept`对于程序员之间的交流是有利的，就像 `const`限定符一样。

2. **显示指定 `noexcept`的函数，编译器会进行优化**

   因为在调用 `noexcept` 函数时不需要记录 exception handler，所以编译器可以生成更高效的二进制码（编译器是否优化不一定，但理论上`noexcept` 给了编译器更多优化的机会）。另外编译器在编译一个 `noexcept(false)` 的函数时可能会生成很多冗余的代码，这些代码虽然只在出错的时候执行，但还是会对 Instruction Cache 造成影响，进而影响程序整体的性能。

3. **容器操作针对 `std::move` 的优化**

   举个例子，一个 `std::vector<T>`，若要进行 `reserve` 操作，一个可能的情况是，需要重新分配内存，并把之前原有的数据拷贝（copy）过去，但如果 T 的移动构造函数是 `noexcept` 的，则可以移动（move）过去，大大地提高了效率。

```
#include <iostream>
#include <vector>
using namespace std;
class A {
  public:
    A(int value) {
    }
    A(const A &other) {
        std::cout << "copy constructor";
    }
    A(A &&other) noexcept {
        std::cout << "move constructor";
    }
};

int main() {
    std::vector<A> a;
    a.emplace_back(1);
```

```
    a.emplace_back(2);
    return 0;
}
```

上述代码可能输出：

```
move constructor
```

但如果把移动构造函数的 noexcept 说明符去掉，则会输出：

```
copy constructor
```

你可能会问，为什么在移动构造函数是 `noexcept` 时才能使用？这是因为它执行的是 Strong Exception Guarantee，发生异常时需要还原，也就是说，你调用它之前是什么样，抛出异常后，你就得恢复成啥样。但对于移动构造函数发生异常，是很难恢复回去的，如果在恢复移动（move）的时候发生异常了呢？但复制构造函数就不同了，它发生异常直接调用它的析构函数就行了。

> 可以查看 https://www.yhspy.com/2019/11/22/C-%E4%B8%AD%E7%9A%84%E7%A7%BB%E5%8A%A8%E6%9E%84%E9%80%A0%E4%B8%8E-noexcept/ 这一片文章有讲解移动构造函数和 `noexcept` 之间的关系的。

STL 为了保证容器类型的内存安全，在大多数情况下，只会调用被标记为不会抛出异常，即被标记为 `noexcept` 或 `noexcept(true)` 的移动构造函数，否则，便会调用其拷贝构造函数来作为代替。

示例程序：

```
// 本程序是cppference.com网站上的示例代码

#include <iostream>
#include <utility>
#include <vector>

void may_throw();
void no_throw() noexcept;
auto lmay_throw = []{};
auto lno_throw = [] () noexcept {};

class T
{
public:
    ~T(){} // dtor prevents move ctor
           // copy ctor is noexcept
};

class U
{
public:
    ~U(){} // dtor prevents move ctor
           // copy ctor is noexcept(false)
    std::vector<int> v;
};
```

```
class V
{
public:
    std::vector<int> v;
}

int main()
{
    T t;
    U u;
    V v;

    std::cout << std::boolalpha
        << "Is may_throw() noexcept? " << noexcept(may_throw()) << '\n'
      // 由于may_throw函数默认都是抛出异常的，所以说会输出false
        << "Is no_throw() noexcept? " << noexcept(no_throw()) << '\n'
      // 由于我们声明了该函数不抛出异常，所以说会输出true
        << "Is lmay_throw() noexcept? " << noexcept(lmay_throw()) << '\n'
      // 我们书写的lambda表达式没有声明，说明是默认的抛出异常,输出false
        << "Is lno_throw() noexcept? " << noexcept(lno_throw()) << '\n'
      // lambda表达式具有声明，所以说不抛出异常，输出true
        << "Is ~T() noexcept? " << noexcept(std::declval<T>().~T()) << '\n'
      // 这里的declval函数是可以获得指定类型的右值引用,所以说std::declval<T>()函数会获得一个T类型的右值引用
      // 然后利用该右值引用来获得T的析构函数，最后在进行判断析构函数的类型，输出结过为true，符合我们的预期
        // note: the following tests also require that ~T() is noexcept because
        // the expression within noexcept constructs and destroys a temporary
        << "Is T(rvalue T) noexcept? " << noexcept(T(std::declval<T>())) <<
'\n'
      // 这里的话是先获得一个右值引用，然后调用T类的默认移动构造函数，默认的移动构造函数是noexcept类型的
      // 这里我们可以在T类中添加会抛出异常的构造函数和拷贝构造函数或者移动构造函数，下面的结果会发生变化
        << "Is T(lvalue T) noexcept? " << noexcept(T(t)) << '\n'
        << "Is U(rvalue U) noexcept? " << noexcept(U(std::declval<U>())) <<
'\n'
        << "Is U(lvalue U) noexcept? " << noexcept(U(u)) << '\n'
        << "Is V(rvalue V) noexcept? " << noexcept(V(std::declval<V>())) <<
'\n'
        << "Is V(lvalue V) noexcept? " << noexcept(V(v)) << '\n';
}
```

其执行结果为：

```
Is may_throw() noexcept? false
Is no_throw() noexcept? true
Is lmay_throw() noexcept? false
Is lno_throw() noexcept? true
Is ~T() noexcept? true
Is T(rvalue T) noexcept? true
Is T(lvalue T) noexcept? true
Is U(rvalue U) noexcept? false
Is U(lvalue U) noexcept? false
Is V(rvalue V) noexcept? true
Is V(lvalue V) noexcept? false
```

17. `decval`和`decltype`

> 参看文章：https://stdrc.cc/post/2020/09/12/std-declval/
>
> 参看文章：https://www.jianshu.com/p/6606796de366
>
> https://blog.csdn.net/u014609638/article/details/106987131

暂时先记着，二者搭配可以获得指定的类中的函数的返回值。

18. 函数指针的声明

```
// 本代码的目的是测试函数指针的一些写法

#include <iostream>
using namespace std;

typedef int(FUNC)(int, int);
typedef int(*FUNC_P)(int, int);

int fun(int a, int b)
{
        cout << "Hello, world!" << endl;
        return 0;
}

int main()
{
        // 上面的typedef只是声明了一个函数类型和一个函数指针类型
        // 第一种方式：
        FUNC *fp = nullptr;
        fp = fun;
        fp(1,2);
        // 第二种方式：
        FUNC_P fp1 = nullptr;
        fp1 = fun;
        fp1(3,4);
        // 第三种方式：直接通过指针类型创建，不使用typedef预定义
        int(*fp2)(int, int) = nullptr;
        fp2 = fun;
        fp2(1,2);
}
```

19. 各种初始化

19.1 `explicit` 禁用构造函数定义的类型转换

我们常见的一些类的隐式初始化变量：

- ***通过一个实参调用的构造函数定义了从构造函数参数类型向类类型隐式转换的规则；***
- ***拷贝构造函数定义了用一个对象初始化另一个对象的隐式转换***

```
#include <iostream>

// Cat提供两个构造函数
class Cat {
 public:
    int age;
    // 接收一个参数的构造函数定义了从int型向类类型隐式转换的规则, explicit关键字可以组织这种转换
    Cat(int i) : age(i) {}
    // 拷贝构造函数定义了从一个对象初始化另一个对象的隐式转换
    Cat(const Cat &orig) : age(orig.age) {}
};

int main() {
    Cat cat1 = 10;     // 调用接收int参数的拷贝构造函数
    Cat cat2 = cat1;   // 调用拷贝构造函数

    std::cout << cat1.age << std::endl;
    std::cout << cat2.age << std::endl;
    return 0;
}

// 输出:
10
10
```

> ***explicit***做的是什么事情呢？就是禁止使用上面所述的两种隐式转换。

智能指针将构造函数声明为`explicit`，所以说智能指针只能直接初始化：

```
#include<memory>

class Cat{
public:
    int age;
  Cat() = default;
  // 必须显式调用拷贝构造函数
  explicit Cat(const Cat& cat) : age(cat.age) {};
}

int main()
{
  Cat cat1;
  Cat cat2(cat1);
  // Cat cat3 = cat1; // 因为我们使用explicit关键字限制了拷贝构造函数的隐式调用
```

```
  // 这里我们实际上是隐式调用了拷贝构造函数

  // std::shared_ptr<int> sp = new int(8);
  // 这里也是错误的，因为我们该行代码实际上是调用隐式构造函数来进行初始化，由于智能指针也是将构造函数声明为`explicit`
  std::shared_ptr<int> sp(new int(8)); // 这样书写就是显示调用拷贝构造函数
}
```

19.2 只允许一步隐式类型转换

表面意思就是你可以进行隐式转换，但是你只能转换一次：

```
class Cat {
 public:
    std::string name;
    Cat(std::string s) : name(s) {}  // 1. 允许string到Cat的隐式类型转换
};

int main() {
    // 2. 错误: 不存在从const char[8]到Cat的类型转换, 编译器不会自动把const char[8]转成string, 再把string转成Cat
    // Cat cat1 = "tomocat";

    // 3. 正确: 显式转换成string, 再隐式转换成Cat
    Cat cat2(std::string("tomocat"));

    // 4. 正确: 隐式转换成string, 再显式转换成Cat
    Cat cat3 = Cat("tomocat");
}
```

19.3 列表初始化

> 搬运文章： https://segmentfault.com/a/1190000039844285

1. C++ 98/ 03与C++11的列表初始化

在C++98/03中，普通数组和POD（Plain Old Data，即没有构造、析构和虚函数的类或结构体）类型可以使用花括号`{}`进行初始化，即列表初始化。但是这种初始化方式仅限于上述提到的两种数据类型：

```
int main() {
    // 普通数组的列表初始化
    int arr1[3] = { 1, 2, 3 };
    int arr2[] = { 1, 3, 2, 4 };  // arr2被编译器自动推断为int[4]类型

    // POD类型的列表初始化
    struct data {
        int x;
        int y;
    } my_data = { 1, 2 };
}
```

C++11新标准中列表初始化得到了全面应用，不仅兼容了传统C++中普通数组和POD类型的列表初始化，还可以用于任何其他类型对象的初始化：

```cpp
#include <iostream>
#include <string>

class Cat {
 public:
    std::string name;
    // 默认构造函数
    Cat() {
        std::cout << "default constructor of Cat" << std::endl;
    }
    // 接受一个参数的构造函数
    Cat(const std::string &s) : name(s) {
        std::cout << "normal constructor of Cat" << std::endl;
    }
    // 拷贝构造函数
    Cat(const Cat &orig) : name(orig.name) {
        std::cout << "copy constructor of Cat" << std::endl;
    }
};

int main() {
    /*
     * 内置类型的列表初始化
     */
    int a{ 10 };        // 内置类型通过初始化列表的直接初始化
    int b = { 10 };     // 内置类型通过初始化列表的拷贝初始化
    std::cout << "a:" << a << std::endl;
    std::cout << "b:" << b << std::endl;

        // 1.C++ 11 新特性
    /*
     * 类类型的列表初始化
     */
    Cat cat1{};                  // 类类型调用默认构造函数的列表初始化
    std::cout << "cat1.name:" << cat1.name << std::endl;
    Cat cat2{ "tomocat" };         // 类类型调用普通构造函数的列表初始化
    std::cout << "cat2.name:" << cat2.name << std::endl;

    // 注意列表初始化前面的等于号并不会影响初始化行为，这里并不会调用拷贝构造函数
    Cat cat3 = { "tomocat" };      // 类类型调用普通构造函数的列表初始化
    std::cout << "cat3.name:" << cat3.name << std::endl;
    // 先通过列表初始化构造右侧Cat临时对象，再调用拷贝构造函数(从输出上看好像编译器优化了，直接调用普通构造函数而不会调用拷贝构造函数)
    Cat cat4 = Cat{ "tomocat" };
    std::cout << "cat4.name:" << cat4.name << std::endl;

    /*
     * new申请堆内存的列表初始化
     */
    int *pi = new int{ 100 };
    std::cout << "*pi:" << *pi << std::endl;
    delete pi;
    int *arr = new int[4] { 10, 20, 30, 40 };
    std::cout << "arr[2]:" << arr[2] << std::endl;
    delete[] arr;
```

```
}

// 输出:
a:10
b:10
default constructor of Cat
cat1.name:
normal constructor of Cat
cat2.name:tomocat
normal constructor of Cat
cat3.name:tomocat
normal constructor of Cat
cat4.name:tomocat
*pi:100
arr[2]:30
```

2. vector中圆括号和花括号的初始化

总的来说，***圆括号是通过调用vector的构造函数进行初始化的***，如果***使用了花括号那么初始化过程会尽可能会把花括号内的值当做元素初始值的列表来处理***。如果初始化时使用了花括号但是提供的值又无法用来列表初始化，那么就考虑用这些值来调用vector的构造函数了。

```
#include <string>
#include <vector>

int main() {
    std::vector<std::string> v1{"tomo", "cat", "tomocat"};  // 列表初始化：包含3个string元素的vector
    // std::vector<std::string> v2("a", "b", "c");          // 错误：找不到合适的构造函数

    std::vector<std::string> v3(10, "tomocat");             // 10个string元素的vector，每个string初始化为"tomocat"
    std::vector<std::string> v4{10, "tomocat"};             // 10个string元素的vector，每个string初始化为"tomocat"

    std::vector<int> v5(10);      // 10个int元素，每个都初始化为0
    std::vector<int> v6{10};      // 1个int元素，该元素的值时10
    std::vector<int> v7(10, 1);   // 10个int元素，每个都初始化为1
    std::vector<int> v8{10, 1};   // 2个int元素，值分别是10和1
}
```

3. 初始化习惯

尽管C++11将列表初始化应用于所有对象的初始化，但是***内置类型习惯于用等号初始化，类类型习惯用构造函数圆括号显式初始化，vector、map和set等容器类习惯用列表初始化***。

```
#include <string>
#include <vector>
#include <set>
#include <map>

class Cat {
 public:
```

```
    std::string name;
    Cat() = default;
    explicit Cat(const std::string &s) : name(s) {}
};

int main() {
    // 内置类型初始化(包括string等标准库简单类类型)
    int i = 10;
    long double ld = 3.1415926;
    std::string str = "tomocat";

    // 类类型初始化
    Cat cat1();
    Cat cat2("tomocat");

    // 容器类型初始化(当然也可以用圆括号初始化, 列表初始化用于显式指明容器内元素)
    std::vector<std::string> v{"tomo", "cat", "tomocat"};
    int arr[] = {1, 2, 3, 4, 5};
    std::set<std::string> s = {"tomo", "cat"};
    std::map<std::string, std::string> m = {{"k1", "v1"}, {"k2", "v2"}, {"k3",
"v3"}};
    std::pair<std::string, std::string> p = {"tomo", "cat"};

    // 动态分配对象的列表初始化
    int *pi = new int {10};
    std::vector<int> *pv = new std::vector<int>{0, 1, 2, 3, 4};

    // 动态分配数组的列表初始化
    int *parr = new int[10]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
}
```

19.4 initializer_list形参

前面提到C++11支持所有类型的初始化，对于类类型而言，虽然我们使用列表初始化它会自动调用匹配的构造函数，但是我们也能显式指定接受初始化列表的构造函数。C++11引入了`std::initializer_list`，允许构造函数或其他函数像参数一样使用初始化列表，这才真正意义上为类对象的初始化与普通数组和 POD 的初 始化方法提供了统一的桥梁。

> Tips:
>
> - 类对象在被列表初始化时会优先调用列表初始化构造函数，如果没有列表初始化构造函数则会根据提供的花括号值调用匹配的构造函数
> - C++11新标准提供了两种方法用于处理可变数量形参， 第一种是我们这里提到的`initializer_list`形参（所有的形参类型必须相同），另一种是可变参数模板（可以处理不同类型的形参）

```
#include <initializer_list>
#include <vector>

class Cat {
 public:
    std::vector<int> data;
    Cat() = default;
    // 接受初始化列表的构造函数
```

```
    Cat(std::initializer_list<int> list) {
        for (auto it = list.begin(); it != list.end(); ++it) {
            data.push_back(*it);
        }
    }
};

int main() {
    Cat cat1 = {1, 2, 3, 4, 5};
    Cat cat2{1, 2, 3};
}
```

初始化列表除了用于对象构造函数上，还可以作为普通参数形参：

```
#include <initializer_list>
#include <string>
#include <iostream>

void print(std::initializer_list<std::string> list) {
    for (auto it = list.begin(); it != list.end(); ++it) {
        std::cout << *it << std::endl;
    }
}

int main() {
    print({"tomo", "cat", "tomocat"});
}
```

20. Pair

`Class Pair`可以将两个`value`视为一个单元。C++标准库中多处使用到了该`Class`，尤其是容器`map`、`multimap`、`unordered_map`、`unordered_multimap`就是使用`pair`来管理其中的键值对元素。还有如果有函数需要返回两个`value`的话，也是需要使用到`pair`，例如说`minmax()`。

`Pair`对应的代码如下：

```
namespace std{
  template <typename T1, typename T2>
  struct pair{
    // member
    T1 first;
    T2 second;
  };
}
```

我们可以看到`Pair`实际上是一个`struct`而不是说一个类，所以说程序是可以访问并且处理其两个值的。

其含有的一些成员函数如下面所示：

操作函数	影响
pair<T1,T2> p	Default 构造函数，建立一个 pair，其元素类型分别为 T1 和 T2，各自以其 default 构造函数初始化
pair<T1,T2> p(val1,val1)	建立一个 pair，元素类型分别为 T1 和 T2，以 *val1* 和 *val1* 为初值
pair<T1,T2> p(rv1,rv2)	建立一个 pair，元素类型分别为 T1 和 T2，以 *rv1* 和 *rv2* 进行搬移式初始化（move initialized）
pair<T1,T2> p(piecewise_construct, t1,t2)	建立一个 pair，元素类型分别为 tuple T1 和 T2，以 tuple *t1* 和 *t2* 的元素为初值
pair<T1,T2> p(p2)	Copy 构造函数，建立 p 成为 *p2* 的拷贝
pair<T1,T2> p(rv)	Move 构造函数，将 *rv* 的内容移至 p（允许隐式类型转换）
p = p2	将 *p2* 赋值给 *p*（始自 C++11；允许隐式类型转换）
p = rv	将 *rv* 的值 move assign 给 *p*（始自 C++11；允许隐式类型转换）
p.first	获得 pair 内的第一 value（直接成员访问）
p.second	获得 pair 内的第二 value（直接成员访问）
get<0>(p)	等价于 p.first（始自 C++11）
get<1>(p)	等价于 p.second（始自 C++11）
p1 == p2	返回“是否 *p1* 等于 *p2*”（等价于 p1.first==p2.first && p1.second==p2.second）
p1 != p2	返回“是否 *p1* 不等于 *p2*”（!(p1==p2)）
p1 < p2	返回“是否 *p1* 小于 *p2*”（比较 first，如果相等则比较 second）
p1 > p2	返回“是否 *p1* 比 *p2* 更大”（亦即 p2<p1）
p1 <= p2	返回“是否 *p1* 小于等于 *p2*”（!(p2<p1)）
p1 >= p2	返回“是否 *p1* 大于等于 *p2*”（!(p1<p2)）
p1.swap(p2)	互换 *p1* 和 *p2* 的数据（始自 C++11）
swap(p1,p2)	同上（是个全局函数）（始自 C++11）
make_pair(val1,val2)	返回一个 pair，带有 *val1* 和 *val2* 的类型和数值

测试代码0：

```
#include <iostream>
#include <utility>

// 如果我们想要将pair按照执行格式输出，并且实现泛型的话，我们就需要类似于下面的书写方式
template<typename T1, typename T2>
std::ostream& operator << (std::ostream & strm, const std::pair<T1, T2> &p)
{
        return strm << "[" << p.first << ", " << p.second << "]";
}

int main()
{
        std::pair<int, double> p(1,1.0);

        std:: cout << p << std::endl;
}
```

C++11 新标准：

从C++11开始，我们可以对`pair`使用一种`tuple-like`接口。所以说我们可以获得`pair`的元素个数（`pair`就是2）以及指定元素的类型,还可以使用`get()`获得`first`或者`second`。

测试代码1：

```cpp
#include <iostream>
#include <utility>

int main()
{
        typedef std::pair<int, double> IntDoublePair;

        IntDoublePair p(1,2.0);

        int a = std::get<0>(p);
        double b = std::get<1>(p);

        int size = std::tuple_size<IntDoublePair>::value;

        std::tuple_element<0, IntDoublePair>::type c = 0;

        std::cout << "pair的第一个元素的值为：" << a << "pair的第二个元素的值为：" <<
b << std::endl;

        std::cout << "pair的size为：" << size << std::endl;
        std::cout << "测试获得pair的元素类型：" << c << std::endl;
}
```

21. Tuple

`Pair`是将两个`value`视为一个单元，而`Tuple`则是将任意个不同类型的`value`视为一个单元。