-基类、子类构造函数的调用，拷贝、赋值函数的调用

-函数定义与声明，全局变量和全局函数的定义与声明

Account( const char \*name, double opening\_bal ) : \_name( name ), \_balance( opening\_bal )更高效？

函数指针：

int (\*pFunc) (int a, int b);

int test(int a) { return a; }

int main(int argc, const char \* argv[])

{ typedef int (\*fp)(int a);

fp f = test;

cout<<f(2)<<endl; return 0; }

Functor / 函数对象:

1. **class** CAverage
2. {
3. **public**:
4. double **operator**()(int a1, int a2, int a3)
5. { //重载()运算符
6. **return** (double)(a1 + a2 + a3) / 3;
7. }
8. };
9. int main()
10. {
11. CAverage average; //能够求三个整数平均数的函数对象
12. cout << average(3, 2, 3); //等价于 cout << average.operator(3, 2, 3);
13. **return** 0;
14. }

函数对象与函数指针比较函数对象可以把附加对象保存在函数对象中是它最大的优点。

Virtual constructor / virtual destructor

构造函数不能是虚函数

1. 从vptr角度解释

虚函数的调用是通过虚函数表来查找的，而虚函数表由类的实例化对象的vptr指针(vptr可以参考C++的虚函数表指针vptr)指向，该指针存放在对象的内部空间中，需要调用构造函数完成初始化。如果构造函数是虚函数，那么调用构造函数就需要去找vptr，但此时vptr还没有初始化！

## [析构函数](https://so.csdn.net/so/search?q=%E6%9E%90%E6%9E%84%E5%87%BD%E6%95%B0&spm=1001.2101.3001.7020)可以且常常是虚函数

      这个原理上就很好理解啦，因为此时 vtable 已经初始化了，完全可以把析构函数放在虚函数表里面来调用。

**C++类有继承时，析构函数必须为虚函数。**如果不是虚函数，则使用时可能存在**内存泄漏**的问题。

#include 指示符读入指定文件的内容 它有两种格式 #include <> #include "my\_file.h" 如果文件名用尖括号 < 和 > 括起来 表明这个文件是一个工程或标准头文件 查 找过程会检查预定义的目录 我们可以通过设置搜索路径环境变量或命令行选项来修改这些 目录 在不同的平台上这些方法大不相同 建议你请教同事或查阅编译器手册以获得更进 一步的信息 如果文件名用一对引号括起来 则表明该文件是用户提供的头文件 查找该 文件时将从当前文件目录开始。

#ifdef DEBUG

cout << "word read: " << word << "\n";

#endif

$ CC -DDEBUG main.C

// 相等与不相等操作

bool operator==( const IntArray& ) const;

bool operator!=( const IntArray& ) const;

// 赋值操作符

IntArray& operator=( const IntArray& );

不可否认 如果 IntArray 构造函数被继承 了 那么就没有这个问题 实际上 那样的话我们会有其他更严重的问题 但现在没有足够的 篇幅向你证明这一点

inline IntArrayRC::IntArrayRC( int sz) : IntArray( sz ) {}

inline IntArrayRC::IntArrayRC( const int \*iar, int sz ) : IntArray( iar, sz ) {}

what is the benefit of exception handling?

在整型文字常量前面加一个 0 该值将被解释成一个八进制数 而在前面加一个 0x 或 0X 则会使一个整型文字常量被解释成十六进制数

逗号表达式是一系列由逗号分开的表达式 这些表达式从左向右计算 逗号表达式的结 果是最右边表达式的值。

函数重载 function overloading 允许多个函数共享同 一个函数名 但是针对不同参数类型提供共同的操作。

C++保证 随着栈的展开 尽管局部类对象的生命期是 因为抛出异常而被结束 但是这些局部类对象的析构函数也会被调用。

因为静态成员函数除了静态数据成员\_interestRate 之外不访问任何其他的数据成员

class Account {

public:

static void raiseInterest( double incr );

static double interest() { return \_interestRate; }

private:

static double \_interestRate; };

inline void Account::raiseInterest( double incr ) { \_interestRate += incr; }

与非嵌套类一样 嵌套类可以有与自身同样类型的成员

// Not ideal configuration: evolving class definition

class List {

public:

class ListItem {

friend class List; // 友元声明

ListItem( int val = 0 ); // 构造函数

ListItem \*next; // 指向自己类的指针 int value; };

// ... private: ListItem \*list; ListItem \*at\_end;

};

As the name suggests, Preprocessors are programs that process our source code before compilation. There are a number of steps involved between writing a program and executing a program in C / C++. Let us have a look at these steps before we actually start learning about Preprocessors.

图示

描述已自动生成

You can see the intermediate steps in the above diagram. The source code written by programmers is first stored in a file, let the name be “**program.c**“. This file is then processed by preprocessors and an expanded source code file is generated named “program.i”. This expanded file is compiled by the compiler and an object code file is generated named “program.obj”. Finally, the linker links this object code file to the object code of the library functions to generate the executable file “program.exe”.

Preprocessor programs provide preprocessor directives that tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a ‘#’ (hash) symbol. The ‘#’ symbol indicates that whatever statement starts with a ‘#’ will go to the preprocessor program to get executed. Examples of some preprocessor directives are: *#include*, *#define*, *#ifndef* etc. Remember that the **#** symbol only provides a path to the preprocessor, and a command such as include is processed by the preprocessor program. For example, #include will include extra code in your program. We can place these preprocessor directives anywhere in our program.

**There are 4 Main Types of Preprocessor Directives:**

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

Hashset, unordered\_set

static\_cast, dynamic\_cast

template <typename T>

T myMax(T x, T y)

{

return (x>y)? x: y;

}

template <class T> void bubbleSort(T a[], int n)

{

for (int i = 0; i < n - 1; i++)

for (int j = n - 1; i < j; j--)

if (a[j] < a[j - 1])

swap(a[j], a[j - 1]);

}

template <class T, class U> class A {

T x;

U y;

public:

A() { cout << "Constructor Called" << endl; }

};

Speed of execution: C++ programs excel in execution speed. Since, it is a compiled language, and also hugely procedural. Newer languages have extra in-built default features such as garbage-collection, dynamic typing etc. which slow the execution of the program overall. Since there is no additional processing overhead like this in C++, it is blazing fast.

C++ 11

- C++11 赋予 auto 关键字新的含义，使用它来做自动类型推导。auto n = 10; auto is int

- decltype 是“declare type”的缩写，译为“声明类型”。auto 要求变量必须初始化，而 decltype 不要求。这很容易理解，auto 是根据变量的初始值来推导出变量类型的，如果不初始化，变量的类型也就无法推导了。decltype 可以写成下面的形式：

decltype(exp) varname;

- typedef unsigned int uint\_t;

using uint\_t = unsigned int;

template <typename Val>

using str\_map\_t = std::map<std::string, Val>;

// ...

str\_map\_t<int> map1;

- using

/\* C++98/03 \*/

template <typename T>

struct func\_t

{

typedef void (\*type)(T, T);

};

// 使用 func\_t 模板

func\_t<int>::type xx\_1;

/\* C++11 \*/

template <typename T>

using func\_t = void (\*)(T, T);

// 使用 func\_t 模板

func\_t<int> xx\_2;

- lambda

auto display = [](int a,int b) -> void{cout << a << " " << b;};

**- overload override**

1. Overload（重载）

　　重载的概念最好理解，在同一个类声明范围中，定义了多个名称完全相同、参数（类型或者个数）不相同的函数，就称之为Overload（重载）。重载的特征如下：

（1）相同的范围（在同一个类中）；

（2）函数名字相同；

（3）参数不同；

（4）virtual 关键字可有可无。

2. Override（覆盖）

　　覆盖的概念其实是用来实现C++多态性的，即子类重新改写父类声明为virtual的函数。Override（覆盖）的特征如下：

（1）不同的范围（分别位于派生类与基类）；

（2）函数名字相同；

（3）参数列表完全相同；

（4）基类函数必须有virtual 关键字。

3. Overwrite（改写）

　　改写是指派生类的函数屏蔽（或者称之为“隐藏”）了与其同名的基类函数。正是这个C++的隐藏规则使得问题的复杂性陡然增加，这里面分为两种情况讨论：

（1）如果派生类的函数与基类的函数同名，但是参数不同。那么此时，不论有无virtual关键字，基类的函数将被隐藏（注意别与重载混淆）。

（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual关键字。那么此时，基类的函数被隐藏（注意别与覆盖混淆）。

unordered\_set 容器，可直译为“无序 set 容器”，即 unordered\_set 容器和 set 容器很像，唯一的区别就在于 set 容器会自行对存储的数据进行排序，而 unordered\_set 容器不会。

**STL**

[**array:**](https://www.geeksforgeeks.org/array-class-c/)Static contiguous array (class template)

array<**int**,6> ar = {1, 2, 3, 4, 5, 6};

[**vector:**](https://www.geeksforgeeks.org/vector-in-cpp-stl/)Dynamic contiguous array (class template)

vector<**int**> g1;

[**deque:**](https://www.geeksforgeeks.org/deque-cpp-stl/)Double-ended queue (class template)

[**forward\_list:**](https://www.geeksforgeeks.org/forward-list-c-set-1-introduction-important-functions/)Singly-linked list (class template)

Forward list in STL implements singly linked list. It differs from the [**list**](https://www.geeksforgeeks.org/list-cpp-stl/) by the fact that the forward list keeps track of the location of only the next element while the list keeps track of both the next and previous elements, thus increasing the storage space required to store each element. The drawback of a forward list is that it cannot be iterated backward and its individual elements cannot be accessed directly.

[**list:**](https://www.geeksforgeeks.org/list-cpp-stl/)Doubly-linked list (class template)

Lists are [sequence containers](https://www.geeksforgeeks.org/containers-cpp-stl/) that allow non-contiguous memory allocation. As compared to vector, the list has slow traversal, but once a position has been found, insertion and deletion are quick.

[**Set:**](https://www.geeksforgeeks.org/set-in-cpp-stl/)Collection of unique keys, sorted by keys   
(class template)

Sets are a type of associative container in which each element has to be unique because the value of the element identifies it.

**Properties:**

1. **Storing order –**The set stores the elements in **sorted** order.
2. **Values Characteristics** – All the elements in a set have **unique values**.
3. **Values Nature**– The value of the element cannot be modified once it is added to the set, though it is possible to remove and then add the modified value of that element. Thus, the valuesare **immutable**.
4. **Search Technique** – Sets follow the **Binary search tree** implementation.
5. **Arranging order –**The values in a set are **unindexed**.

[**unordered\_set:**](https://www.geeksforgeeks.org/unorderd_set-stl-uses/)Collection of unique keys, hashed by keys. (class template)

An **unordered\_set** is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized. All operations on the **unordered\_set** takes constant time **O(1)** on an average which can go up to linear time **O(n)** in worst case which depends on the internally used hash function, but practically they perform very well and generally provide a constant time lookup operation.

**Difference between Set and Unordered Set**

| Set | Unordered Set |
| --- | --- |
| Set stores elements in a sorted order | Unordered Set stores elements in an unsorted order |
| Set stores/acquire unique elements only | Unordered Set stores/acquire only unique values |
| Set uses Binary Search Trees for implementation | Unordered Set uses Hash Tables for implementation |
| More than one element can be erased by giving the starting and ending iterator | We can erase that element for which the iterator position is given |
| **set<datatype> Set\_Name;** | **unordered\_set<datatype> UnorderedSet\_Name;** |

[**Map:**](https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/)Collection of key-value pairs, sorted by keys, keys are unique (class template).

[**multiset:**](https://www.geeksforgeeks.org/multiset-in-cpp-stl/)Collection of keys, sorted by keys (class template)

[**multimap:**](https://www.geeksforgeeks.org/multimap-associative-containers-the-c-standard-template-library-stl/)Collection of key-value pairs, sorted by keys   
(class template)

[**unordered\_map:**](https://www.geeksforgeeks.org/unordered_map-in-stl-and-its-applications/)Collection of key-value pairs, hashed by keys, keys are unique. (class template)

[**unordered\_multiset:**](https://www.geeksforgeeks.org/unordered_multiset-and-its-uses/)Collection of keys, hashed by keys (class template)

[**unordered\_multimap:**](https://www.geeksforgeeks.org/unordered_multimap-and-its-application/)Collection of key-value pairs, hashed by keys (class template)

**unordered\_set**  
总的来说，unordered\_set 容器具有以下几个特性：

1. 不再以键值对的形式存储数据，而是直接存储数据的值；
2. 容器内部存储的各个元素的值都互不相等，且不能被修改。
3. 不会对内部存储的数据进行排序（这和该容器底层采用哈希表结构存储数据有关，可阅读《[C++ STL无序容器底层实现原理](http://c.biancheng.net/view/7235.html" \t "_blank)》一文做详细了解）

**Garbage Collector in Java and C#**

**Shared memory, mutex, Semaphore**

**Mutex is for inter thread communication**

**Exceptional handling**

**Solarflare**

C++11引入了auto和decltype关键字，使用他们可以在编译期就推导出变量或者表达式的类型，方便开发者编码也简化了代码。

**auto** a **=** 10; *// 10是int型，可以自动推导出a是int*

cont **int** **&**i **=** 1;

**int** a **=** 2;

**decltype**(i) b **=** 2; *// b是const int&*

auto不能用作函数参数

在类中auto不能用作非静态成员变量

auto不能定义数组，可以定义指针

C++11 中 auto 和 decltype 结合再借助「尾置返回类型」还可推导函数的返回类型。⽰例：

// 利⽤ auto 关键字将返回类型后置

template<typename T, typename U>

auto add1(T x, U y) -> decltype(x + y) {

return x + y;

}

从 **C++14** 开始⽀持仅⽤ auto 并实现返回类型推导，见下⽂ C++14 章节。

C++11的一大亮点就是引入了Lambda表达式。利用Lambda表达式，可以方便的定义和创建匿名函数。

[**https://www.cnblogs.com/DswCnblog/p/5629165.html**](https://www.cnblogs.com/DswCnblog/p/5629165.html)

右值引用是C++11中新增加的一个很重要的特性，他主是要用来解决C++98/03中遇到的两个问题，第一个问题就是临时对象非必要的昂贵的拷贝操作，第二个问题是在模板函数中如何按照参数的实际类型进行转发。

[**https://blog.csdn.net/qq\_20853741/article/details/119155511**](https://blog.csdn.net/qq_20853741/article/details/119155511)

## Move constructor and move assignment operator

// move constructor

    ArrayWrapper (ArrayWrapper&& other)

        : \_p\_vals( other.\_p\_vals  )

        , \_size( other.\_size )

    {

        other.\_p\_vals = NULL;

        other.\_size = 0;

    }

1. The parameter is a non-const rvalue reference
2. other.\_p\_vals is set to NULL

### But why do we need to set other.\_p\_vals = NULL? The reason is the destructor--when the temporary object goes out of scope, just like all other C++ objects, its destructor will run. When its destructor runs, it will free \_p\_vals. The same \_p\_vals that we just copied! If we don't set other.\_p\_vals to NULL, the move would not really be a move--it would just be a copy that introduces a crash later on once we start using freed memory. This is the whole point of a move constructor: to avoid a copy by changing the original, temporary object!

可以发现左值引用真是用的很不爽，**右值引用的出现解决了这个问题**，在STL的很多容器中，都实现了以**右值引用为参数**的移动构造函数和移动赋值重载函数，或者其他函数，最常见的如std::vector的push\_back和emplace\_back。参数为左值引用意味着拷贝，为右值引用意味着移动。

### std::function, std::bind??

std::unique\_ptr

**char** **\*const** ptr; *// 指针本身是常量*

**const** **char\*** ptr; *// 指针指向的变量为常量*

### constexpr

constexpr是c++11新引入的关键字，用于编译时的常量和常量函数，这里直接介绍constexpr和const的区别：

两者都代表可读，const只表示read only的语义，只保证了运行时不可以被修改，但它修饰的仍然有可能是个动态变量，而constexpr修饰的才是真正的常量，它会在编译期间就会被计算出来，整个运行过程中都不可以被改变，constexpr可以用于修饰函数，这个函数的返回值会尽可能在编译期间被计算出来当作一个常量，但是如果编译期间此函数不能被计算出来，那它就会当作一个普通函数被处理。如下代码：

**#include**<iostream>

**using** **namespace** std;

**constexpr** **int** **func**(**int** i) {

**return** i **+** 1;

}

**int** **main**() {

**int** i **=** 2;

func(i);*// 普通函数*

func(2);*// 编译期间就会被计算出来*

}

### 新增数据结构

* std::forward\_list：单向链表，只可以前进，在特定场景下使用，相比于std::list节省了内存，提高了性能

std**::**forward\_list**<int>** fl **=** {1, 2, 3, 4, 5};

**for** (**const** **auto** **&**elem : fl) {

cout **<<** elem;

}

* std::unordered\_set：基于hash表实现的set，内部不会排序，使用方法和set类似
* std::unordered\_map：基于hash表实现的map，内部不会排序，使用方法和set类似
* std::array：数组，在越界访问时抛出异常，建议使用std::array替代普通的数组
* std::tuple：元组类型，类似pair，但比pair扩展性好

### 基于范围的for循环

直接看代码

vector**<int>** vec;

**for** (**auto** iter **=** vec.begin(); iter **!=** vec.end(); iter**++**) { *// before c++11*

cout **<<** **\***iter **<<** endl;

}

**for** (**int** i : vec) { *// c++11基于范围的for循环*

cout **<<** "i" **<<** endl;

}

### 左值、右值

**概念1**：

左值：可以放到等号左边的东西叫左值。

右值：不可以放到等号左边的东西就叫右值。

**概念2**：

左值：可以取地址并且有名字的东西就是左值。

右值：不能取地址的没有名字的东西就是右值。

**举例**：

**int** a **=** b **+** c;

a是左值，a有变量名，也可以取地址，可以放到等号左边, 表达式b+c的返回值是右值，没有名字且不能取地址，&(b+c)不能通过编译，而且也不能放到等号左边。

**Make:** [**https://www.gnu.org/software/make/manual/make.html**](https://www.gnu.org/software/make/manual/make.html)

**Premake:** PreMake可以自动生成多个平台的MakeFile，比起其它的MakeFile生成软件（qmake, cmake），它最大的特点应该是它的工程文件其实是使用LUA语言编写的脚本，这样一来，它的工程文件可以拥有非常强大的表达能力。

**Jenkins: CI/CD**

**Train: MS thing, integrate SCM, CI/CD etc**

**C++11 multi-thread programing, std::future, 条件变量**

**GitHub**

**Linux Sim**

**Multi-thread and Multi-process program**

**Java**

**Java script**

**Phython**

**Linux C++**

**C++ 11**

**C++ shared memory**

**Build/Compile/Debug C++ in Linux env**

**Web 3**