

c++传统知识 .....	3
c++小型知识点 .....	3
常见疑难问题 .....	12
OOP 进阶 .....	13
头文件与 namespace .....	22
c++文件与字符串操作 .....	24
标准 API .....	29
标准库详解 .....	32
c++内存管理 .....	48
内存管理入门 .....	48
STL 内存管理 .....	50
返回值的内存管理 .....	55
引用与指针 .....	58
现代内存与指针管理技术 .....	59
std::move 与右值引用 .....	61
new /delete 运算符 .....	67
malloc 类函数行为研究 .....	68
c++内存管理之性能优化 .....	69
c++新特征 .....	72
c++新特性 .....	72
std::forward 与完美转发 .....	81
{}-Initialization .....	82
Using 的用法 .....	84
range-based for loop .....	85
继承中的新特性 .....	87
c++指令编程 .....	88
宏编程，泛型编程与元编程 .....	92
C/C++宏编程 .....	92
C/C++泛型编程 .....	94
可变参数与可变参数模板 Variadic template .....	96
C++元编程 .....	104
第三方 c++库 .....	105
多线程与多核编程 .....	114
C++多线程并发基础入门教程 .....	114
硬件与并发 .....	118
多线程与 OOP .....	119
定时器线程 .....	123
线程的资源管理 .....	124
多线程的内存模型 .....	126
c++的锁 .....	128
高性能多线程 .....	130
OpenMP .....	132
C/C++ 多线程编程/ 绑定 CPU .....	132
编译优化与体系结构 .....	132

基本概览 .....	132
cache 友好编程 .....	133
内核与网络 .....	136
c++致命错误 .....	136
c++ debug 工具 .....	137
c++性能优化概述 .....	137

wujianjunml@outlook.com

gcc 任何警告都不允许。

## c++传统知识

### c++小型知识点

#### 标准输入重定向

代码：

```
#include<iostream>
using namespace std;
int main(){
    cout << "start ......" << endl;
    int x;
    int sum = 0;
    while(cin>>x){
        sum += x;
    }
    cout << "sum=" <<sum<< endl;
}
```

数据文件：



运行：

```
$ ./t.exe <data.txt
start .....
sum=16
```

#### 新字符类型

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5     cout << "start ......" << endl;
6
7     wchar_t name1[] = L"吴小帅abc"; // 每个字符2个字节
8     cout << sizeof(name1) << endl;
9     cout << sizeof(*name1) << endl;
10
11    char32_t name2[] = U"吴小帅abc"; // 每个字符4个字节
12    cout << sizeof(name2) << endl;
13    cout << sizeof(*name2) << endl;
14 }
```

OUTPUT    TERMINAL    DEBUG CONSOLE    PROBLEMS    2

```
PS D:\wujianjun\code\c++\tmp\cpptest\src\basicapi> g++ 't.cpp' -o t
start .....
14
2
28
4
```

```
30     char *names[] = {"abc", "123"};
31     for(auto c : names){
32         std::cout << c << endl;
33     }
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

abc  
123

### How to typedef array for different sizes

```
10
11     typedef char MM_typecode[4];
12     MM_typecode matcode;
13     cout << sizeof(matcode) << endl;
14     matcode[0] = 'a';
15     matcode[1] = 'b';
16     matcode[2] = 'c';
17     matcode[3] = 'd';
18     for(auto c : matcode){
19         std::cout << c << endl;
20     }
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
39
40
41
42
43
44
45
46
47
48
49
49
50
51
52
53
54
55
56
57
58
59
59
60
61
62
63
64
65
66
67
68
69
69
70
71
72
73
74
75
76
77
78
79
79
80
81
82
83
84
85
86
87
88
89
89
90
91
92
93
94
95
96
97
98
99
100
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2

PS D:\wujianjun\code\c++\tmp\cpptest\src\basicapi> g++
4
a
b
c
d

### 全局对象与静态对象

比如，我们要定义一个 class 的全局对象，那么在 class 的.h 文件里，用 extern 声明：

```
1  #ifndef _LOG_H_
2  #define _LOG_H_
3
4  #include <iostream>
5  #include <string>
6  #include <iostream>
7
8  enum ENUM_LOGEND {END};
9
10 const ENUM_LOGEND END = END;
11
12 class LOG{
13 public:
14     void print(std::string log_str);
15
16     // 虽然下面几行看起来像函数，但是指针函数实现时会报编译错误，且iostream也是这样重载<<的
17     LOG& operator<<(std::string val);
18     LOG& operator<<(long val);
19     LOG& operator<<(int val);
20     LOG& operator<<(float val);
21     LOG& operator<<(double val);
22     LOG& operator<<(ENUM_LOGEND val);
23
24
25 private:
26     std::ostringstream logoss;
27 };
28
29 extern LOG logout; // 声明一个全局对象，其定义请将接时在别的地方找
30
31 #endif
```

然后在.cpp 中定义：

```

18 LOG& LOG::operator<<(string val){
19     logoss << val;
20     return *this;
21 }
22
23 LOG& LOG::operator<<(long val){
24     logoss << val;
25     return *this;
26 }
27
28 LOG& LOG::operator<<(int val){
29     logoss << val;
30     return *this;
31 }
32
33 LOG& LOG::operator<<(double val){
34     logoss << val;
35     return *this;
36 }
37
38 LOG& LOG::operator<<(float val){
39     logoss << val;
40     return *this;
41 }
42
43 LOG& LOG::operator<<(ENUM_LOGEND val){
44     time_t t = time(0);
45     char now_t_str[32] = {0};
46     strftime(now_t_str, 32, "%Y-%m-%d %H:%M:%S", localtime(&t));
47
48     cout<<"====INFO====|"<<now_t_str<<"|"<<logoss.str()<<endl;
49
50     logoss.clear();
51     logoss.str("");
52
53     return *this;
54 }
55
56 LOG logout = LOG();
57

```

这样一来，别的程序只要 include 这个 class 的.h 文件就可以使用这个全局对象了。

### Evaluate a string with a switch in C++

I want to evaluate a string with a switch but throws me the following error.

```

string a;
cin>>a;
switch (string(a))
{
    case "Option 1":
        cout<<"It pressed number 1"<<endl;
        break;
    case "Option 2":
        cout<<"It pressed number 2"<<endl;
        break;
    case "Option 3":
        cout<<"It pressed number 3"<<endl;
        break;
    default:
        cout<<"She put no choice"<<endl;
        break;
}

```

Serhiy:

As said before, **switch can be used only with integer values**. So, you just need to convert your "case" values to integer. You can achieve it by using **constexpr** from c++11, thus some calls of **constexpr functions can be calculated in compile time**. something like that...

```

switch (str2int(s))
{
    case str2int("Value1"):
        break;
    case str2int("Value2"):
        break;
}

```

where str2int is like :

```

constexpr unsigned int str2int(const char* str, int h = 0)
{
    return !str[h] ? 5381 : (str2int(str, h+1) * 33) ^ str[h];
}

```

二维数组还是一维数组

刚学习 C++ 那会这个问题曾困扰过我，后来慢慢形成了不管什么时候都用一维数组的习惯。

```
// 一维数组的申请和释放
int *m = new int[n_row * n_col];
for( int i = 0; i < n_row; ++i ){
    for( int j = 0; j < n_col; ++j ){
        m[i * n_col + j] = i*j; // 一维数组访问
    }
}
delete [] m; // 一维数组释放内存
m = nullptr;

// 二维数组的申请和释放
int **m = new int*[n_row];
for( int i = 0; i < n_row; ++i ) // 多次new, 内存并不连续!!!!!!
    m[i] = new int[n_col];
for( int i = 0; i < n_row; ++i ){
    for( int j = 0; j < n_col; ++j ){
        m[i][j] = answer; // 二维数组访问
    }
}
for( int i = 0; i < n_row; ++i ) // 二维数组释放内存
    delete [] m[i];
delete [] m;
m = nullptr;

// 对一维数组进行三维索引
// 本质上还是一维数组,但是实现了形式上的二维数组调用
int ***m = new int*[n_row]; // 二维数组指针,m[i]指向block中的某一段内存的起始地址
int *block = new int[n_row * n_col]; // 代价就是需要一块空间保存每行的其实地址
for( int i = 0; i < n_row; ++i )
    m[i] = &block[i * n_col];
for( int i = 0; i < n_row; ++i ){
    for( int j = 0; j < n_col; ++j ){
        m[i][j] = answer; // 二维数组形式的访问
    }
}
delete [] block; // 释放内存
block = nullptr;
delete [] m;
m = nullptr;
```

测试代码如上，结果发现：一维数组的效率完爆二维数组

为什么工程中最好不要用二维数组？

<https://www.zhihu.com/question/302824062>

作者 Milo Yip:

用二级指针的好处是下标不用自行计算(可写成 A[i][j])。但对矩阵数组也用二级指针的话，会有不必要的内存(存储指针的数组)和运行时开销(每次访问元素多一次寻址)。

指针与引用：

The screenshot shows a code editor with the following code in a file named 'main.cpp':

```
1085
1086 int main()
1087 {
1088     // 指针
1089     string str = "cccccc";
1090     string *str_p = &str;
1091     cout<<str_p->c_str()<<endl;
1092     cout<<*str_p<<endl;
1093     // 引用
1094     string& str_i = str;
1095     cout<<str_i.c_str()<<endl;
1096     cout<<str_i<<endl;
1097
1098     return 0;
1099 }
```

Below the code, there is a terminal window showing the output of the program:

```
[Running] cd "d:\wujianjun\code\my\zacpp\zacpp\src\"  
cccccc  
cccccc  
cccccc  
cccccc
```

数组与指针

指针++是向前移动指针类型对应的字节数，指针--即向后移动指针类型对应的字节数。

设 `ptr` 是一个指向地址 1000 的 `int`(4 字节)指针，执行 `ptr++` 运算之后，`ptr` 将指向位置 1004。

如果 `ptr` 指向一个地址为 1000 的字符，上面的运算会导致指针指向位置 1001，。

**指针 `p+n` 其实是 `p` 加上 `n`\*类型占用字节数。**

### What is the fastest portable way to copy an array in C++

In C++ you should use `std::copy` by default unless you have good reasons to do otherwise.

- ◆ C++ classes define their own copy semantics via the copy constructor and copy assignment operator, and of the operations listed, only `std::copy` respects those conventions.
- ◆ `memcpy()` uses raw, byte-wise copy of data (though likely heavily optimized for cache line size, etc.), and ignores C++ copy semantics (it's a C function, after all...).
- ◆ `cblas_dcopy()` is a function for use in linear algebra routines using double precision floating point values. It likely excels at that, but shouldn't be considered general purpose.

If your data is "simple" POD type struct data or raw fundamental type data, `memcpy` will likely be as fast as you can get. Just as likely, `std::copy` will be optimized to use `memcpy` in these situations, so you'll never know the difference.

The behavior of this function template is equivalent to:

```
1 template<class InputIterator, class OutputIterator>
2     OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result)
3 {
4     while (first!=last) {
5         *result = *first;
6         ++result; ++first;
7     }
8     return result;
9 }
```

例子如下：

```
2052 ~ int main()
2053 { 
2054     int a[10];
2055     for(int i = 0; i < 10 ; i++)
2056     {
2057         *(a + i) = i * 10;
2058     }
2059     for(int i = 0; i < 10 ; i++)
2060     {
2061         cout<< a[i] << ",";
2062     }
2063     cout << endl;
2064 
2065     int b[10];
2066     copy(a, a +10, b);
2067     for(int i = 0; i < 10 ; i++)
2068     {
2069         cout<< b[i] << ",";
2070     }
2071     cout << endl;
2072 }
2073 
```

问题 3    输出    调试控制台    终端    JUPYTER  
[Running] cd "d:\wujianjun\code\my\zancode"  
0,10,20,30,40,50,60,70,80,90,  
0,10,20,30,40,50,60,70,80,90,

### Why is string to number conversion so slow in C++?

This function reads an array of doubles from a string:

```
vector<double> parseVals(string& str) {
    stringstream ss(str);
    vector<double> vals;
    double val;
    while (ss >> val) vals.push_back(val);
    return vals;
}
```

When called with a string containing 1 million numbers, the function takes 7.8 seconds to

execute. I have found a great alternative in the [Boost/Spirit](#) library. The code is safe, concise and extremely fast (0.06 seconds on VC2012, 130x faster than stringstream).

```
#include <boost/spirit/include/qi.hpp>

namespace qi = boost::spirit::qi;
namespace ascii = boost::spirit::ascii;

vector<double> parseVals4(string& str) {
    vector<double> vals;
    qi::phrase_parse(str.begin(), str.end(),
        *qi::double_ >> qi::eoi, ascii::space, vals);
    return vals;
}
```

jxh:

On my Linux system, I have g++ 4.6.3, compiled with -O3. I used cygwin g++ 4.5.3, also compiled with -O3. On Linux, I got the following output.

```
elapsed: 0.46 stringstream
elapsed: 0.11 strtod
```

On cygwin, I got the following:

```
elapsed: 1.685 stringstream
elapsed: 0.171 strtod
```

## 函数对比

- ◆ float `strtof (const char* str, char** endptr);`

Parses the C-string `str` interpreting its content as a floating point number and returns its value as a float. The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes as many characters as possible that are valid, and interprets them as a numerical value. A pointer to the rest of the string after the last valid character is stored in the object pointed by `endptr`.

```
1255 #include <iostream>
1256 #include <string>
1257 #include <stdlib.h>
1258
1259 int main()
1260 {
1261     std::string d1 = "0.265656";
1262     std::string d2 = "1.0";
1263     std::cout << strtof(d1.c_str(), nullptr) << std::endl;
1264     std::cout << strtof(d2.c_str(), nullptr) << std::endl;
1265 }
```

输出  
[Running] cd "d:\wujianjun\code\my\zacpp\zacpp\src\" && g++ t  
0.265656  
1

- ◆ float `stof( const std::string& str, std::size_t* pos = nullptr );`

### const 进阶

引用必须初始化，我们可以定义一个**指针的引用**:

```
int ival = 1092;
int *pi = &ival;
int *&pi2 = pi; //ok
```

**const 引用可以用不同类型的对象初始化(只要能从一种类型转换到另一种类型即可)**，也可以是不可寻址的值，如文字常量。如下，左边正确，右边会报错：

```

double dval = 3.14159;
//下3行仅对const引用才是合法的
const int &ir = 1024;
const int &ir2 = dval;
const double &dr = dval + 1.0;

```

double dval = 3.14159;  
 //下3行仅对const引用才是错误的  
 int &ir = 1; // 错误,一般引用不能引用字面量  
 int &ir2 = dval; // 错误,一般引用必须严格类型一致,不能自动转换  
 double &dr = dval + 1.0; // 错误,一般引用必须绑定一个变量,不能是无名中间量。

**引用在内部存放的是一个对象的地址,它是该对象的别名。**对于不可寻址的值,如文字常量,以及不同类型的对象,编译器为了实现引用,必须生成一个临时对象,将该对象的值置入临时对象中,引用实际上指向该对象,但用户不能访问它。**例如:**

```

double dval = 3.14159;
//下3行仅对const引用才是合法的
const int &ir = 1024;
const int &ir2 = dval;
const double &dr = dval + 1.0;

```

**编译器将其转换为:**

```

double dval = 3.14159;
int tmp1 = 1024;
const int &ir = tmp1;
int tmp2 = dval; // double -> int
const int &ir2 = tmp2;
double tmp3 = dval + 1.0;
const double &dr = tmp3;

```

**编译器产生临时变量的时候引用必须为 const!!!!const 引用是指 const T &, 此时引用本身无法再被修改, const 引用表明保证不会通过此引用间接的改变被引用的对象!**

```

int i = 1;
const int &r = i; //r绑定i, 其后对r的操作实际上是针对i进行的
// r = 2;          //错误, 不能通过修改r来实现对i的修改
i = 2;            //i可修改

```

int i = 1;
 int &r = i; //r绑定i, 其后对r的操作实际上是针对i进行的
 r = 2; //正确,修改引用等同修改引用的对象
 i = 2; //i可修改

- ◆ const 引用可读不可改, 与绑定对象是否为 const 无关; 非 const 引用可读可改, 只可与非 const 对象绑定。

const 还可以修饰函数:

- ◆ int& fun(const int& a); 多数情况下我们都会传引用, 节省内存并且可以起到改变实参。不过有的时候我们并不希望改变实参的值, 就要加上 const 关键字。**一定要思考函数是否会修改参数, 如果不会修改的话一定要加上 const.**
- ◆ const int& fun(int& a); 返回值是引用的函数, **这个引用一定是成员变量或者是函数参数**, 返回的时候为了避免其成为左值被修改, 就需要加上 const 关键字来修饰。

```

class A
{
private:
    int data;
public:
    A(int num):data(num){}
    ~A(){}
    int& get_data(){
        return data;
    }
};
int main()
{
    A a(1);
    cout<< a.get_data()<< endl; //data=1
    a.get_data() = 3;
    cout<< a.get_data()<< endl; //data=3
    return 0;
}

```

class A
{
private:
 int data;
public:
 A(int num):data(num){}
 ~A(){}
 const int& get\_data(){
 return data;
 }
};
int main()
{
 A a(1);
 cout<< a.get\_data()<< endl; //data=1
 // a.get\_data() = 3; // 错误!!!
 return 0;
}

- ◆ int& fun(int& a) const{}: 函数内部只能读而不能修改类的成员。

```

class A
{
private:
    int data;
public:
    A(int num):data(num){}
    ~A(){}
    int get_data() const{
        // data = 10; // 错误
        return data;
    }
};

int main()
{
    A a(1); // 这是一个常成员
    int b = a.get_data();
    cout << b << endl; // 1
    b = b + 3;
    cout << b << endl; // 4
    return 0;
}

```

if a class variable is marked as *mutable*, and a "const function" could change this variable .

看下面的例子：

```

2813 #include <iostream>
2814 #include <string>
2815 using namespace std;
2816
2817 void f1(string& data)
2818 {}
2819
2820 int main()
2821 {
2822     string str("A");
2823     f1(str);
2824     const string& str2 = str;
2825     f1(str2);
2826 }

```

问题 ⑤ 编译 调试控制台 终端 JUPYTER  
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\src\" && g++ test.cpp -o test  
test.cpp: In function 'int main()':  
test.cpp:2825:5: error: binding reference of type 'std::\_\_cxx11::string\*' (aka 'std:  
f1(str);  
^~~~~  
test.cpp:2817:6: note: initializing argument 1 of 'void f1(std::\_\_cxx11::string\*)'  
void f1(string& data)  
^~

编译报错，参数加一个 **const** 即可。

```

2813 #include <iostream>
2814 #include <string>
2815 using namespace std;
2816
2817 void f1(const string& data)
2818 {}
2819
2820 int main()
2821 {
2822     string str("A");
2823     f1(str);
2824     const string& str2 = str;
2825     f1(str2);
2826 }

[Done] exited with code=0 in 0.704 seconds

```

## 基本类型的自动转换

c++中在做边界值比较时，如果类型不同，很有可能比较错误，比如 **float** 的 **0.01** 和 **double** 的 **0.01** 其实是不相等的，如下：

```

3800 template <class T>
3801 void compare(T v)
3802 {
3803     double low = 0;
3804     double high = 0.01;
3805     bool b = v >= low && v < high;
3806     cout << b << endl;
3807 }
3808
3809 int main()
3810 {
3811     float v1 = 0.01;
3812     compare<float>(v1);
3813     double v2 = 0.01;
3814     compare<double>(v2);
3815     cout << setprecision(4) << v1 << " | " << v2 << endl;
3816     cout << setprecision(8) << v1 << " | " << v2 << endl;
3817 }

问题 ① 输出 调试控制台 终端 JUPYTER
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\src\" &&
1
0
0.01|0.01
0.0099999998|0.01

```

## int 与 unsigned short

unsigned short 的取值范围是 0 - 65535, short 的取值范围是-32768 - 32767, 见如下链接:

<https://learn.microsoft.com/en-us/cpp/c-language/cpp-integer-limits?view=msvc-170>

小心把一个不在这个范围的值赋予时会出问题, 如下:

```
3826 int a = 65535;
3827 int b = 123456789;
3828 short int* m_data = new short int[2];
3829 m_data[0] = a;
3830 m_data[1] = b;
3831 cout << m_data[0] << "," << m_data[1] << endl;
3832
3833
```

问题 8 域出 调试控制台 终端 JUPITER  
[Running] cd "d:\wujianjun\code\my\zancode\zacpp\zacpp\src\" && g++ -I,-1,-13035

## 尾递归

什么是尾递归 <https://www.zhihu.com/question/20761771>

用普通递归和尾递归则分别是 1 到 n 求和:

```
// 普通递归
int recsum(int x){
    if(x == 1){
        return x;
    }
    else{
        return x + recsum(x - 1);
    }
}

// 尾递归
int tailrecsum(int x, int sum){
    if(x == 0){
        return sum;
    }
    else{
        return tailrecsum(x - 1, sum + x);
    }
}
```

当调用 recsum(5) 和 tailrecsum(5), 分别发生如下状况:

recsum(5)	tailrecsum(5, 0)
5 + recsum(4)	tailrecsum(4, 5)
5 + (4 + recsum(3))	tailrecsum(3, 9)
5 + (4 + (3 + recsum(2)))	tailrecsum(2, 12)
5 + (4 + (3 + (2 + recsum(1))))	tailrecsum(1, 14)
5 + (4 + (3 + 3))	tailrecsum(0, 15)
5 + (4 + 6)	
5 + 10	
15	

尾递归, 比线性递归多一个参数, 这个参数是上一次调用函数得到的结果; 尾递归由于将

外层方法的结果传递给了内层方法，那外层方法没有任何利用价值了，直接从栈里踢出去就行了，可以保证同时只有一个栈帧在栈里存活，节省了大量栈空间。

怎么写尾递归？形式上只要最后一个 return 语句是单纯函数就可以。如：`return tailrec(x+1);`，而 `return tailrec(x+1) + x;` 则不可以。再比如下面个是尾递归么？

```
public static int acc(int n){  
    if(n == 1){  
        return 1;  
    }  
    return n + acc(n - 1);  
}
```

答案是否定的。可能有的人会说，明明最后一个步骤就是调用 acc，为啥不是尾递归？实际上，这个方法的 return 先拿到 acc(n-1) 的值，然后再将 n 与其相加，所以求 acc(n-1) 并不是最后一步，因为最后还有一个 add 操作。累加的尾递归写法是下面这样子的：

```
public static int accTail(int n, int sum){  
    if(n == 1){  
        return sum + n;  
    }  
    return accTail(n - 1,sum + n);  
}
```

在计算机学里，尾调用是指一个函数里的最后一个动作是返回一个函数的调用结果的情形，即最后一步新调用的返回值直接被当前函数的返回结果。

## 常见疑难问题

### undefined reference

- ◆ 如果将函数的实现放在头文件中，那么每一个包含该头文件的 `cpp` 文件都将得到一份关于该函数的定义，那么链接器会报函数重定义错误。
- ◆ 如果将函数的实现放在 `cpp` 文件中，并且标记为 `inline`，那么该函数对其他编译单元不可见，也就是其他 `cpp` 文件不能链接该函数库，会出现的 `... undefined reference to ...`  
**The body of an inline function needs to be in the header so that the compiler can actually substitute it wherever required.**
- ◆ 将类的成员函数的实现放在头文件中不会出现重定义错误，是因为在类中定义成员函数默认为 `inline` 函数。

## OOP 进阶

### 几个重要函数

- ◆ 构造函数：构造函数的函数名与类名相同，没有返回值类型，也没有返回值。**当在类中定义任意构造函数后(包括拷贝构造函数)，编译器就不会为我们定义默认构造函数。**对象在创建时构造函数的调用顺序为：
  - 调用父类的构造函数；
  - 调用成员变量的构造函数，按照它们声明的顺序调用；
  - 调用类自身的构造函数；

C++ 初始化列表，知道这些就够了

<https://zhuanlan.zhihu.com/p/33004628>

```

class Base
{
public:
    Base(int val){
        m_num = 0;
        cout << "create Base(int val)" << endl;
    }
private:
    int m_num;
};

class BaseChild: public Base
{
public:
    BaseChild(){
        m_num = 0;
        cout << "create is BaseChild()" << endl;
    }
private:
    int m_num;
};

int main(int argc, char *argv[])
{
    BaseChild child; // 错误, 因为此时会调用父类的构造函数, 但是并没有合适的父类构造函数可以调用
}

```

BaseChild 继承 Base 时没有显式地指定 Base 的构造函数，所以编译报错。我们用什么办法不去调用默认构造函数，而是显式的调用 Base 带参构造函数呢。答案就是初始化列表。如下：

```

BaseChild():Base(1)
{
    cout << "create is BaseChild()" << endl;
}

```

初始化列表还可以对类本身的数据成员进行初始化：

```

BaseChild():Base(1), m_num(0){...}

```

如果是对象之间的赋值呢，例如：

```

BaseChild child = BaseChild();

```

其实，这又涉及了另外一个话题，赋值构造函数和编译器的优化。其具体执行顺序是：

1. 调用 BaseChild 构造函数，生成一个临时对象。

2. 给 child 对象赋值

3. 创建 child 对象后，删除临时对象。

而编译器有可能会优化代码为 BaseChild child()。总结一下：

- ◆ 当类 B 含有成员 a 为 A 类型时，在类 B 的所有构造函数中，必须使用初始化列表对 a 初始化。在构造函数体内赋值是不行的。
- ◆ 如果不在初始化列表中指定基类的构造函数，编译器会调用基类的默认构造函数。  
当类 B 继承类 A 时，在 B 中也需要使用构造函数初始化列表对 A 进行初始化，

注意下面这个例子：

```

1557 class Animal
1558 {
1559     public:
1560         Animal()
1561         { cout<<"Animal construct default "<<endl; data = nullptr; }
1562         Animal(int size)
1563         { cout<<"Animal construct size "<<endl; data = new float[size]; }
1564         ~Animal()
1565         { cout<<"Animal free stat..." << data <<endl;
1566             delete[] data; data = nullptr;
1567             cout<<"Animal free ok!!! " << data <<endl;
1568         };
1569         public:
1570             float* data;
1571     };
1572
1573 class Factory
1574 {
1575     public:
1576         Factory(){ cout<<"Factory construct default "<<endl; }
1577         Factory(int size)
1578         { cout<<"Factory construct size "<<endl;
1579             animal = Animal(size);
1580         }
1581         public:
1582             Animal animal;
1583     };
1584
1585 int main()
1586 {
1587     {
1588         Factory factory = Factory(5);
1589     }
1590     cout<<"======"<<endl;
1591     return 0;
1592 }

```

问题 ② 输出 调试控制台 终端

```

[Running] cd "d:\wujianjun\code\my\zancode\zacpp\zacpp\src\" && g++ test.cpp -o test
Animal construct default
Factory construct size
Animal construct size
Animal free stat...0x25b24f0
Animal free ok!!!
Animal free stat...0x25b24f0
=====

```

[Done] exited with code=0 in 4.099 seconds

```

1557 class Animal
1558 {
1559     public:
1560         Animal()
1561         { cout<<"Animal construct default "<<endl; data = nullptr; }
1562         Animal(int size)
1563         { cout<<"Animal construct size "<<endl; data = new float[size]; }
1564         ~Animal()
1565         { cout<<"Animal free stat..." << data <<endl;
1566             delete[] data; data = nullptr;
1567             cout<<"Animal free ok!!! " << data <<endl;
1568         };
1569         public:
1570             float* data;
1571     };
1572
1573 class Factory
1574 {
1575     public:
1576         Factory(){ cout<<"Factory construct default "<<endl; }
1577         Factory(int size):animal(size)
1578         { cout<<"Factory construct size "<<endl;
1579             // animal = Animal(size);
1580         }
1581         public:
1582             Animal animal;
1583     };
1584
1585 int main()
1586 {
1587     {
1588         Factory factory = Factory(5);
1589     }
1590     cout<<"======"<<endl;
1591     return 0;
1592 }

```

问题 ② 输出 调试控制台 终端

```

[Running] cd "d:\wujianjun\code\my\zancode\zacpp\zacpp\src\" && g++ test.cpp -o test
Animal construct size
Factory construct size
Animal free stat...0x26024f0
Animal free ok!!!
=====

```

[Done] exited with code=0 in 3.977 seconds

左边是在构造函数体内对成员做初始化的，可以看出首先调用了成员的默认构造函数，而且在析构时，对同一块内存析构了两次，最后导致 core dump。而右边在初始化列表中对成员做了初始化，程序运行果然如预期。

◆ 析构函数：析构函数的函数名为波浪号~加类名。析构函数只能有一个，不能被重载。

**当对象生命周期临近结束时，析构函数由C++编译器自动调用。**

```

class Student
{
    private:
        char *name;
        int height;
    public:
        Student(char * name, int height)      //构造函数
        {
            this->name = (char *)malloc(sizeof(name) + 1);
            strcpy(this->name, name);
            this->height = height;
        }
        Student(const Student &s)           //拷贝构造函数
        {
            // 深拷贝
            this->name = (char *)malloc(100);
            this->name = strcpy(this->name, s.name);
            this->height = height;
        }
        ~Student() // 析构函数
        {
            if (this->name != NULL)
            {
                free(this->name);
                this->name = NULL;
                this->height = 0;
            }
        }
};

```

析构函数的调用函数与构造函数相反

- 执行自身的析构函数；
  - 执行成员变量的析构函数；
  - 执行父类的析构函数；
- ◆ 拷贝构造函数：只有一个参数，是同类对象的引用。拷贝构造函数主要有以下场景：
- 对象作为函数的参数，以值传递的方式传给函数。
  - 对象作为函数的返回值，以值的方式从函数返回。
  - 使用一个对象给另一个对象初始化。

拷贝构造函数和赋值运算符的行为比较相似，区别在于：

- ◆ 拷贝构造函数是一种构造函数，那么它的功能就是创建一个新的对象实例；
  - ◆ 赋值运算符是执行某种运算，将一个对象的值复制给另一个对象(已经存在的)。
- 调用的是拷贝构造函数还是赋值运算符，主要是看是否有新的对象实例产生。**如果产生了新的对象实例，那调用的就是拷贝构造函数；如果没有，那就是对已有的对象赋值，调用的是赋值运算符。

```

125 class A {
126 public:
127     A() // 构造函数
128     { std::cout << this << " constructor" << std::endl; }
129
130     A(const A& orig) // 拷贝构造函数
131     { std::cout << this << " copy constructor" << std::endl; }
132
133     A& operator=(const A& orig) // 赋值运算符重载
134     {
135         std::cout << this << " operator=" << std::endl;
136         return *this;
137     }
138
139     ~A() // 析构函数
140     { std::cout << this << " destructor" << std::endl; }
141
142 };
143
144 int main()
145 {
146     A a1 = A(); //
147     A a2 = A();
148     A a3 = a1;
149     a3 = a2;
150     std::cout << "======" << std::endl;
151
152     return 0;
153 }
154
155

```

问题 ② 输出 调试控制台 终端

```

[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ main.cpp -o
0x61fe0f constructor
0x61fe0e constructor
0x61fe0d copy constructor
0x61fe0d operator=
=====
0x61fe0d destructor
0x61fe0e destructor
0x61fe0f destructor

```

- ◆ 注意，构造函数可以互相调用：

```

class Data
{
public:
    Data(const int& _x, const int& _y):x(_x),y(_y){}
};

class A
{
public:
    A(const int& x, const int& y):A(Data(x, y))
    {cout << "A(int i)" << endl;}

    A(const Data& d)
    {cout << "A(i)" << endl;}
};

int main()
{
    A a(10,20);
}

```

问题 ② 输出 调试控制台 终端

```

[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ main.cpp -o
A(i)
A(int i)

```

## 运算符重载

运算符重载如下：

```
70 class Vector
71 {
72 public:
73     Vector(int a);
74     int& operator[](int nIndex);
75 private:
76     int m_nGril[4];
77 };
78 Vector::Vector(int a)
79 {
80     m_nGril[0] = a*10, m_nGril[1] = a*20;
81     m_nGril[2] = a*30, m_nGril[3] = a*40;
82 }
83 // 重载数组下标运算符[],且返回的是引用
84 int& Vector::operator[](int nIndex)
85 {
86     cout << "operator[]" << endl;
87     return m_nGril[nIndex];
88 }
89
90 int main()
91 {
92     Vector vt(5);
93     cout << vt[2] << endl; // 调用重载的运算符
94 }
```

问题 ② 编译 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" &  
operator[]  
150

```
89 class Matrix
90 {
91 public:
92     Matrix(int, int); // 为 m_nMatrix,m_nRow,m_nCol 赋初值
93     int& operator()(int, int);
94 private:
95     int * m_nMatrix;
96     int m_nRow, m_nCol;
97 };
98 Matrix::Matrix(int nRow, int nCol)
99 {
100     m_nRow = nRow, m_nCol = nCol;
101     m_nMatrix = new int[m_nRow * m_nCol];
102     for(int i = 0; i < m_nRow * m_nCol; ++i)
103     {
104         *(m_nMatrix + i) = i;
105     }
106 }
107 // 重载圆括号运算符(),且返回的是引用
108 int& Matrix::operator()(int nRow, int nCol)
109 {
110     cout << "operator()" << endl;
111     return *(m_nMatrix + nRow * m_nCol + nCol);
112 }
113
114 int main()
115 {
116     Matrix mtx(10, 10);
117     cout << mtx(3, 4) << endl; // 调用重载的运算符
118 }
119 }
```

问题 ② 编译 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" & g++ main.c  
operator()  
34

C++中 operator ()的重载主要用于实现

- ◆ **Functor:** 对函数指针的面向对象化，具有诸多优点。
- ◆ C++的 operator[]只允许你声明一个参数，所以很难实现下面的用法：

```
import numpy as np
x = np.random.randn(10, 10)
x[5, 5] # correct
```

此时可以重载小括号：

```
template <typename T, std::size_t ROWS, std::size_t COLS>
struct Matrix
{
    T data[ROWS][COLS];
    T& operator()(int x, int y)
    {
        return data[x][y];
    }
};

Matrix<int, 10, 20> m;
m(5, 5) = 10;
std::cout << m(5, 5) << std::endl;
```

## 继承

当一个类派生自基类，该基类可以被继承为 **public**、**protected** 或 **private** 几种类型：

- ◆ **public:** 基类的公有成员也是派生类的公有成员，基类的保护成员也是派生类的保护成员，基类的私有成员不能直接被派生类访问。

- ◆ **protected:** 基类的公有和保护成员将成为派生类的保护成员。
- ◆ **private:** 当一个类派生自私有基类时，基类的公有和保护成员将成为派生类的私有成员。

通过派生类创建对象时必须要调用基类的构造函数。且会首先调用基类构造函数，再调用派生类构造函数。

子类调用父类的同名函数：

- ◆ 返回值类型相同，函数名相同，有 **virtual** 关键字，则由对象的类型决定调用哪个函数。
- ◆ **函数名相同，没有 virtual 关键字，则子类的对象没有办法调用到父类的同名函数，父类的同名函数被隐藏了，可以强制调用父类的同名函数 `class::function_name`。**
- ◆ 参数不同，函数名相同，有 **virtual** 关键字，则不存在多态性，父类的同名函数被隐藏了，可以强制调用父类的同名函数 `class::function_name`。

```

1810 class People{
1811 public:
1812     People(string name_): name(name_) {}
1813     void print(){ cout<< "People: name=" << name << endl;}
1814 public:
1815     string name;
1816 };
1817
1818 class Student: public People{
1819 public:
1820     Student(string name, float score_): People(name), score(score_) {}
1821     void print() { cout<< "Student:" << endl; People::print(); cout<< "score=" << score << endl;}
1822 public:
1823     float score;
1824 };
1825
1826 int main(){
1827
1828     Student stu("小明", 90.5);
1829     cout<< "-----" << endl;
1830     stu.print();
1831
1832     cout<< "-----" << endl;
1833     cout<< stu.name << endl;
1834 }

```

问题 ⑧ 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zancode\zacpp\zacpp\src\" && g++ test.cpp -o test && "d:\wujianjun\code\my\zancode\zacpp\zacpp\src\test"  
=====  
Student:  
People: name=小明  
score=90.5  
=====  
小明

## 虚函数

虚函数只能借助于指针或者引用来达到多态的效果。

纯虚函数在基类中没有定义，要求任何派生类都要定义自己的实现方法。语法格式为：

`virtual 返回值类型 函数名(函数参数) = 0;`

虚函数简单示例如下：

```

class A
{
public:
    virtual void foo()
    {
        cout<<"A::foo() is called"<<endl;
    }
};
class B:public A
{
public:
    void foo()
    {
        cout<<"B::foo() is called"<<endl;
    }
};
int main(void)
{
    A *a = new B();
    a->foo(); // 在这里, a虽然是指向A的指针, 但是被调用的函数(foo)却是B的!
    return 0;
}

```

子类继承父类时，

- ◆ 父类的纯虚函数必须重写，否则子类也是一个虚类不可实例化。
- ◆ 父类中虚函数，则子类也可以不重写，相当于原样继承了父类的虚函数。也可以重写，就相当于覆盖了父类的虚函数实现。

访问类中的成员函数有以下几种方式：

- ◆ 通过对对象名访问：派生类主要看同名函数在派生类中有没有定义：
  - 没有定义则使用基类中的函数；
  - 有定义，那么基类中的同名函数就被隐藏了，只使用派生类中的函数。
- ◆ 通过指向派生类对象的基类指针访问：
  - 基类函数有 `virtual` 关键字则使用派生类中的定义。
  - 基类函数没有 `virtual` 关键字：则使用基类的成员函数。

## 继承时的析构函数和拷贝构造函数

我们定义派生类的析构函数时，不用管基类部分的成员，只撤销派生类自己的成员即可。

编译器会自己调用基类的析构函数。我们不用在派生类的析构函数中显示调用它。

◆ 派生类的析构函数不负责撤销基类对象的成员。

◆ 编译器总是显式调用基类的析构函数。

但是如果一个指向基类的指针，实际指向了一个派生类时，`delete` 指针时只有基类的析构函数被调用，如果派生类中新增了动态分配的成员，则这部分内存未被释放，发生内存泄露。

```
3243 class Base
3244 {
3245     public:
3246     Base(string base_name):base_name(base_name){}
3247     ~Base(){cout << "Base:: deconstruct" << endl;}
3248     public:
3249     string base_name;
3250 };
3251 class A: public Base
3252 {
3253     public:
3254     A(string base_name,string a_name):Base(base_name),a_name(a_name){}
3255     ~A(){ cout << "A:: deconstruct" << endl; }
3256     public:
3257     string a_name;
3258 };
3259 int main()
3260 {
3261     A* a = new A("base","unknown");
3262     delete a; // 此时会自动调用基类的析构函数
3263     cout << "-----" << endl;
3264     Base *base = new A("base","unknown"); // 一个指向基类的指针，实际指向了一个派生类,
3265     delete base; // 此时只会调用基类的析构函数!!!!派生类的成员的内存就没有回收，造成内存泄露。
3266 }
```

问题 6 输出 调试控制台 终端 JUPYTER  
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\zacpp\src\" && g++ test.cpp -o test && "d:\wujianjun\code\my\zacode\zacpp\zacpp\src\" test  
A:: deconstruct  
Base:: deconstruct  
-----  
Base:: deconstruct

应该怎样处理这种情况？答案就是在基类中定义 `virtual` 析构函数。

```
3243 class Base
3244 {
3245     public:
3246     Base(string base_name):base_name(base_name){}
3247     virtual ~Base(){cout << "Base:: deconstruct" << endl;}
3248     public:
3249     string base_name;
3250 };
3251 class A: public Base
3252 {
3253     public:
3254     A(string base_name,string a_name):Base(base_name),a_name(a_name){}
3255     ~A(){ cout << "A:: deconstruct" << endl; }
3256     public:
3257     string a_name;
3258 };
3259 // 应该在基类的析构函数中回收基类成员的内存，在每个派生类的析构函数中回收派生类的成员的内存。
3260 // 基类的析构函数必须为虚函数。
3261 int main()
3262 {
3263     A* a = new A("base","unknown");
3264     delete a;
3265     cout << "-----" << endl;
3266     Base *base = new A("base","unknown");
3267     delete base;
3268 }
```

问题 5 输出 调试控制台 终端 JUPYTER  
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\zacpp\src\" && g++ test.cpp -o test && "d:\wujianjun\code\my\zacode\zacpp\zacpp\src\" test  
A:: deconstruct  
Base:: deconstruct  
-----  
A:: deconstruct  
Base:: deconstruct

◆ 应该在基类的析构函数中回收基类成员的内存，在派生类的析构函数中回收派生类的成员的内存。

◆ 基类的析构函数必须为虚函数。

子类的拷贝构造函数，必须在初始化列表中调用父类的构造函数。

```
3243 class Base
3244 {
3245     public:
3246         Base(string base_name):base_name(base_name){cout << "Base::constrcut" << endl;}
3247         Base(const Base& base){ this->base_name = base.base_name; cout << "Base::copy" << endl;}
3248         void print(){cout << "Base:: base_name=" << base_name << endl; }
3249     public:
3250         string base_name;
3251     };
3252 class A: public Base
3253 {
3254     public:
3255         A(string base_name,string a_name):Base(base_name),a_name(a_name){cout << "A::constrcut" << endl;}
3256         A(const Base& base):Base(base) // 注意,调用父类的拷贝构造函数
3257         { this->base_name = base.base_name; cout << "A::copy" << endl; }
3258         void print(){cout << "A:: base_name=" << base_name << ",a_name=" << a_name << endl; }
3259     public:
3260         string a_name;
3261     };
3262 void print(Base b) // 这里参数为子类时,首先会调用父类的拷贝构造函数,然后只能访问父类的函数了
3263 {
3264     b.print();
3265 }
int main()
{
    A a1("base","a1");
    cout << "-----" << endl;
    print(a1);
    return 0;
}
```

问题 ⑥ 输出 调试控制台 终端 JUPYTER  
[Running] cd "d:\wujianjun\code\my\zancode\zcpp\zcpp\src\" && g++ test.cpp -o test && "d:\wujianjun\code\my\za  
Base::constrcut  
A::constrcut  
-----  
Base::copy  
Base:: base\_name=base

**explicit** 关键字

构造函数被 **explicit** 修饰后，就不能再被隐式调用了。

先看一个例子：

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

class Point {
public:
    Point(int x = 0, int y = 0): x(x), y(y) {}
    // 没有默认值的构造函数无法完成隐式转换
    // Point(int x, int y): x(x), y(y) {}
public:
    int x, y;
};

void displayPoint(const Point& p) {
    cout << "(" << p.x << "," << p.y << ")" << endl;
}

int main()
{
    displayPoint(1); // 隐式转换, 调用有默认值的构造函数
    Point p = 1; // 隐式转换, 调用有默认值的构造函数
}
```

问题 ⑧ 输出 调试控制台 终端 JUPYTER  
[Running] (1,0)

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

class Point {
public:
    explicit Point(int x = 0, int y = 0): x(x), y(y) {}
    // 没有默认值的构造函数无法完成隐式转换
    // Point(int x, int y): x(x), y(y) {}
public:
    int x, y;
};

void displayPoint(const Point& p) {
    cout << "(" << p.x << "," << p.y << ")" << endl;
}

int main()
{
    displayPoint(1); // 隐式转换, 调用有默认值的构造函数
    Point p = 1; // 隐式转换, 调用有默认值的构造函数
}
```

问题 ⑧ 输出 调试控制台 终端 JUPYTER  
[Running] displayPoint() // 隐式转换, 调用有默认值的构造函数  
test.cpp:3967:6: note: in passing argument 1 of 'void displayPo  
void displayPoint(const Point& p) {

Effective C++ 中也写：

被声明为 **explicit** 的构造函数通常比其 **non-explicit** 兄弟更受欢迎，因为它们禁止编译器执行非预期（往往也不被期望）的类型转换。只有很少的情况下可以允许隐式转换，如下：

```
class MyString{
public:
    MyString(const char *p){};
};

int main()
{
    MyString s1 = "Brian"; // 隐式转换调用 MyString(const char *p)
    MyString s2("Fawlty"); // 正常调用 MyString(const char *p)
}
```

## C++ Prevent copy a member data

例子如下：

```
2750 class P
2751 {
2752     public:
2753         P():data(""){}
2754         P(string data_):data(data_){}
2755         P(const P& obj){ // 拷贝构造函数
2756             cout << "copy P:" << obj.data << endl;
2757             data = obj.data;
2758         }
2759         P& operator=(const P& obj){ // 赋值构造函数
2760             cout << "=" << endl;
2761             data = obj.data;
2762             return *this;
2763         }
2764         P(P&& obj){ // 移动构造函数
2765             cout << "move P" << endl;
2766             data = move(obj.data);
2767         }
2768         string data;
2769     };
2770 class Q
2771 {
2772     public:
2773         Q():p("") {}
2774         Q(P& p_):p(p_) {}
2775         const P& getP() const{
2776             return p;
2777         }
2778         P p;
2779     };
2780 int main(){
2781     P p("AA"); Q q(p);
2782     cout << "======" << endl;
2783     cout << &(q.p) << endl; // 注意，访问内部成员并没有copy!!!!!!应该是编译器优化导致的
2784     cout << &(q.getP()) << endl; // 注意，访问内部成员并没有copy!!!!!!
2785     cout << "======" << endl;
2786     P p1 = q.p; // 这里的 = 会导致copy!!! 也就是返回的是引用，但是如果赋值给非引用类型则需要拷贝
2787     cout << "======" << endl;
2788     P p2 = q.p;
2789 }
```

输出结果：

```
copy P:AA
=====
0x61fda0
0x61fda0
=====
copy P:AA
=====
```

## C++ static 静态成员函数

## 头文件与 namespace

<https://www.runoob.com/w3cnote/cpp-header.html>

一个 C++ 程序的内容可以分成不同的部分，分别放在不同的.cpp 文件里，.cpp 文件里的东西都是相对独立的，在编译时不需要互通，只需要在编译成目标文件后做一次链接就行了。比如，在文件 a.cpp 中定义了一个全局函数 void f(){}，而在文件 b.cpp 中需要调用这个函数。那么在文件 b.cpp 中，在调用 void f() 函数之前先声明一下这个函数 void f(); 就可以了。编译器在编译 b.cpp 的时候会生成一个符号表，像 void f() 这样的看不到定义的符号，就会被存放在这个表中。再进行链接的时候，编译器就会在别的目标文件中去寻找这个符号的定义。注意这里提到了两个概念，

- ◆ 定义：就是把一个符号完完整整地描述出来。
- ◆ 声明：则只是声明这个符号的存在，即告诉编译器，这个符号是在其他文件中定义的，我这里先用着，你链接的时候再到别的地方去找找看它到底是什么吧。

需要注意的是，一个符号在整个程序中可以被声明多次，但却要且仅要被定义一次。我们把所有的函数声明全部放进一个头文件中，当某一个.cpp 源文件需要它们时，它们就可以通过一个宏命令 #include 包含进这个.cpp 文件中。**#include 在预处理时把它后面所写的那个文件的内容，完完整整地，一字不改地包含到当前的文件中来。头文件的作用是被其他的.cpp 包含进去的，它们本身并不参与编译。头文件中应该只放变量和函数的声明，而不能放它们的定义，因为一个头文件的内容实际上是会被引入到多个不同的 .cpp 文件中并被编译。**

如果放了定义，那么也对于一个符号做了重复定义，编译器将报错。在头文件中可以：

- ◆ 使用 extern 声明变量或者函数：其声明的函数和变量可以在本模块外其他模块中使用。
- ◆ 声明函数原型：void f();。
- ◆ 写 const 对象的定义：**因为全局的 const 对象默认是没有 extern 的声明的，所以它只在当前文件中有效。即使它被包含到其他多个.cpp 文件中，这个对象对其他文件来说是不可见的，所以便不会导致多重定义。**同时，因为这些 .cpp 文件中的该对象都是从一个头文件中包含进去的，这样也就保证了 const 对象的值是相同的。**同理，static 对象的定义也可以放进头文件。**
- ◆ 写内联函数(inline)的定义：因为 inline 函数是编译器在遇到它的地方根据它的定义把它内联展开的，而并非是普通函数那样可以先声明再链接的(**内联函数不会链接**)。C++ 规定，**内联函数可以在程序中定义多次，但内联函数在一个.cpp 文件中只出现一次，并且在所有的.cpp 文件中，同一个内联函数的定义是一样的。**那么显然，把内联函数的定义放进一个头文件中是非常明智的做法。
- ◆ 写类(class)的定义：把类的定义放进头文件，在使用到这个类的.cpp 文件中去包含这个头文件，是一个很好的做法。还有另一种办法。**那就是直接把函数成员的实现代码也写进类定义里面，此时编译器会视这个函数为内联的。**注意一下，**如果把函数成员的定义写在类定义的头文件中，而没有写进类定义中，这是不合法的。**

设想一下，如果 a.h 中含有类 A 的定义，b.h 中含有类 B 的定义，由于类 B 的定义依赖了类 A，所以 b.h 中也 #include 了 a.h。现在有一个源文件，它同时用到了类 A 和类 B，于是程序员在这个源文件中既把 a.h 包含进来了，也把 b.h 包含进来了。这时，问题就来了：类 A 的定义在这个源文件中出现了两次！于是整个程序就不能通过编译了。可以如下：

```
#ifndef _TEST_H_
#define _TEST_H_

// ...

#endif
```

编译源程序会经过预处理、编译、汇编和连接几个过程。其中预处理器实现以下的功能：

- ◆ 文件包含：把源程序中的`#include` 的.h 文件找到并展开到`#include` 所在处。
- ◆ 条件编译：根据`#if` 和`#ifdef` 等编译命令将源程序中的某部分包含进来或排除在外。
- ◆ 宏展开：将源程序文件中出现的对宏的引用展开成相应的宏定义。

已知 a.h 声明了一系列函数，b.cpp 中实现了这些函数，在 c.cpp 中`#include "a.h"`，那么 c.cpp 是怎样找到 b.cpp 中的实现呢？其实这跟.cpp 和.h 文件名称没有任何直接关系。在源文件编译后生成了目标文件(.o 文件)中，这些函数和变量就视作一个个符号。在 link 的时候，需要说明需要连接哪个.o 文件，连接器会去在 b.cpp 生成的.o 文件中找到实现的函数。**通常，连接器会在每个.o 文件中都去找一下所需要的符号，而不是只在某个文件中找或者说找到一个就不找了。**如果在几个不同文件中实现了同一个函数/变量，就会提示 redefined。

命名空间的定义使用关键字 `namespace`，后跟命名空间的名称，如下所示：

```
namespace namespace_name {  
    // 代码声明  
}
```

通常情况下，在头文件中声明一个命名空间。：

```
//contosoData.h  
#pragma once  
namespace ContosoDataServer  
{  
    void Foo();  
    int Bar();  
}
```

如果函数实现位于一个单独的文件中，则限定函数名称。

**contosodata.cpp 中的函数实现应使用完全限定的名称，即使将指令放在 using 文件顶部：**

```
#include "contosodata.h"  
  
namespace ContosoDataServer{  
    void ContosoDataServer::Foo()  
    {  
        ContosoDataServer::Bar();  
    }  
  
    int ContosoDataServer::Bar()  
    {  
        return 0;  
    }  
}  
  
#include "contosodata.h"  
using namespace ContosoDataServer;  
  
// use fully-qualified name here  
void ContosoDataServer::Foo()  
{  
    // no qualification needed for Bar()  
    Bar();  
}  
  
int ContosoDataServer::Bar(){return 0;}
```

为了调用带有命名空间的函数或变量，需要在前面加上命名空间的名称，如下所示：

`name::code;` // code 可以是变量或函数

**一个命名空间的各个组成部分可以分散在多个文件中。**下面的命名空间定义可以是定义一个新的命名空间，也可以是为已有的命名空间增加新的元素：

```
namespace namespace_name {  
    // 代码声明  
}
```

注意几条：

- ◆ `using namespace xxx;` 尽量不许在头文件中使用。不让这么用，主要原因就是防止名字重复(即自定义变量名和 std 中名字重复)，因为头文件会被很多地方使用，你不知道这个 `using` 能覆盖多大范围。但在 cpp 文件中可以使用，但是，必须用在所有`#include` 之后。还有人想把 `using namespace std;` 放在自定义的命名空间中，你可以去试试(搞死你)。

如果你非常明确的想在一个头文件中使用 `using` 声明，应该怎么做？

- ◆ 使用 `typedef`:

```
std::map<std::string, long> clientLocations;  
  
typedef std::map<std::string, long> ClientNameToZip;  
ClientNameToZip clientLocations;
```

## c++文件与字符串操作

### 字符串 string

很多公司内部都有实现一个优化版本的 string, 比如 facebook 中的 fbstring。fbstring 的 find 使用了 boyer\_moore 算法等字符串搜索算法, 宣传是比 std 库的 string::find 字符串搜索性能提升超过 30 倍。

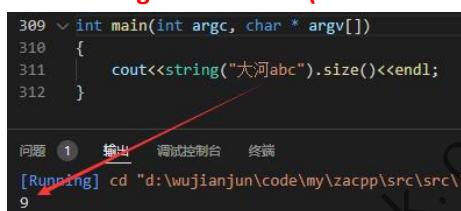
在 STL 中 vector 和 string 是比较特殊的, clear()之后是不会释放内存空间的, 也就是 size() 会清零, 但 capacity()不会改变。想释放空间的话, 用 swap 清空方法为: string().swap(str); c++11 里新加入的的 std::basic\_string::shrink\_to\_fit 也可以。

注意常用成员函数:

- ◆ `size_t size() const;`

Returns the length of the string, in terms of bytes.

**the value returned may not correspond to the actual number of encoded characters in sequences of multi-byte or variable-length characters (such as UTF-8).**



```
309 int main(int argc, char * argv[])
310 {
311     cout<<string("天河abc").size()<<endl;
312 }
```

问题 1 堆出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\"  
9

- ◆ `string substr(size_t pos = 0, size_t len = npos) const;`

Returns a newly constructed string object with its value initialized to a copy of a substring of this object. The substring is the portion of the object that starts at character position `pos` and spans `len` characters (or until the end of the string, whichever comes first).

- ◆ `void swap(string& str);`

After the call to this member function, the value of this object is the value `str` had before the call, and the value of `str` is the value this object had before the call.

- ◆ `size_t find(const string& str, size_t pos = 0) const;`

Searches the string for the `first occurrence` of the sequence specified by its arguments.

When `pos` is specified, the search only includes characters at or after position `pos`, ignoring any possible occurrences that include characters before `pos`.

Notice that unlike member `find_first_of`, whenever more than one character is being searched for, it is not enough that just one of these characters match, but the entire sequence must match.

- ◆ `size_t find_first_of(const string& str, size_t pos = 0) const;`

字符数组 s 中任意一个字符第一次出现的位置。

- ◆ `size_t rfind(const string& str, size_t pos = npos) const;`

Searches the string for the `last occurrence` of the sequence specified by its arguments.

When `pos` is specified, the search only includes sequences of characters that begin at or before position `pos`, ignoring any possible match beginning after `pos`.



```
533 string feature1="1,.....";
534 int pos1 = feature1.rfind(",");
535 cout<<pos1<<endl;
536 cout<<"|"<<feature1.substr(feature1.rfind(","))<<"|"<<endl;
537
538 string feature2="1,.....ABXY";
539 int pos2 = feature2.rfind("ABXY");
540 cout<<pos2<<endl;
541 cout<<"|"<<feature2.substr(feature2.rfind("ABXY"))<<"|"<<endl;
542
```

问题 8 堆出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ main.cpp -o main &&
6
|,|
7
|ABXY|

◆ `const char* c_str() const;`

Returns a pointer to an array that contains a null-terminated sequence of characters (i.e., a C-string) representing the current value of the string object.

◆ `void shrink_to_fit();`

Requests the string to reduce its capacity to fit its size.

This function has no effect on the string length and cannot alter its content.

```
309 int main(int argc, char * argv[])
310 {
311     string s1("Hello");
312     string s2(4, 'K');
313
314     string s4 = s1 + s2; // 连接
315
316     cout<<s1<<, "<<s2<<, "<<, "<<s4<<endl;
317
318     bool a = string("hello") != string("hello");
319     bool b = string("hello") == string("hello");
320     cout<<a<<, "<<b<< endl; // 相等判断
321
322     string s5("abcdef");
323     for(string::iterator iter = s5.begin(); iter < s5.end() ; iter++) // 正向迭代
324     {
325         cout<<*iter;
326     }
327     cout<<endl;
328
329     for(string::reverse_iterator riter = s5.rbegin() ; riter < s5.rend() ; riter++)
330     {
331         cout<<*riter;
332     }
333     cout<<endl; //fedcba
334 }
```

问题 ① 编辑 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ main.cpp -o main && "d:\wujianjun\main.exe"  
Hello,KKKK,,HelloKKKK  
0,1  
abcdef  
fedcba

◆ `find_first_not_of(const string& str)`

Searches the string for the first character that does not match any of the characters specified in its arguments.

◆ `find_last_not_of(const string& str)`

Searches the string for the last character that does not match any of the characters specified in its arguments.

```
3328 int main() {
3329     std::string str = " - M(13 )";
3330
3331     cout << str.find_first_not_of(" -") << endl;
3332     cout << str.find_last_not_of(" ") << endl;
3333 }
```

问题 ① 编辑 调试控制台 终端 JUPYTER  
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\zacpp\src"
3
6

Does `std::string::clear` reclaim the memory associated with a string?

Calling `std::string::clear()` merely sets the size to zero. The `capacity()` won't change. If you want to reclaim the memory allocated for a string, you'll need to do something along the lines of

```
std::string(str).swap(str);
```

Copying the string `str` will generally only reserve a reasonable amount of memory and swapping it with `str`'s representation will install the resulting representation into `str`. Obviously, if you want the string to be empty you could use

```
std::string().swap(str);
```

## 字符串流 **istringstream** 和 **ostringstream** 的用法

他们都定义在<sstream>中。 **ostringstream** 可以替代 **sprintf**, 避免申请大量的缓冲区。

◆ **ostringstream:** 用于执行字符串的输出操作。

- **ostringstream (ios\_base::openmode which = ios\_base::out);**
- **ostringstream (const string& str, ios\_base::openmode which = ios\_base::out);**
- **string str(); returns a string with a copy of the current contents of the stream.**
- **void str (string& s); sets s as the contents of the stream, discarding previous contents.**

```
391 int main()
392 {
393
394     ostringstream oss("abc");
395     oss << "0000" << endl; // 格式化, 此处endl也将格式化进osstr中
396     cout << oss.str(); // 方法str()将缓冲区的内容复制到一个string对象中, 并返回
397
398     //重置使用同一个ostringstream对象时, 调用clear()清除当前控制状态, 调用str("")除脏数据
399     oss.clear();
400     oss.str("");
401     cout << oss.str() << endl;
402
403     oss << "xxxxxx";
404     oss.str("_def"); // 覆盖原有内容, 且默认是ios_base::out, 这不会移动写入位置, 下一次 << 也还是从0开始.
405     cout << oss.str() << endl;
406     oss << "gg";
407     cout << oss.str() << endl;
408
409     return 0;
410 }
```

问题 ① 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ main.cpp -o main && "d:\wujianjun\code\my\zacpp\src\src\main" \_def  
\_def  
gg

◆ **istringstream:** 用于执行字符串的输入操作。

```
392 void test(string line){
393     istringstream iss(line);
394     string sid;
395     string y;
396     string day;
397     iss >> sid >> y >> day;
398     cout << sid << "|" << y << "|" << day << endl;
399 }
400 int main()
401 {
402     // 空格和任意多个空格, 可以分开
403     string line1 = "105560581@1518568726 0.6 2022-03-27";
404     test(line1);
405     string line2 = "105560581@1518568726 0.6 2022-03-27";
406     test(line2);
407     // 任意多个\t和\n, 也可以分开
408     string line3 = "105560581@1518568726\t0.6\t2022-03-27";
409     test(line3);
410     string line4 = "105560581@1518568726\n0.6\n2022-03-27";
411     test(line4);
412     return 0;
413 }
```

问题 ① 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ main.cpp -o main && "d:\wujianjun\code\my\zacpp\src\src\main"
105560581@1518568726[0.6]2022-03-27
105560581@1518568726[0.6]2022-03-27
105560581@1518568726[0.6]2022-03-27
105560581@1518568726[0.6]2022-03-27

注意>>a 会首先清空 a 已有的的内容:

```
422     string sid;
423     string y;
424     string day;
425
426     istringstream iss1("123456@123456 0.6 2022-03-27");
427     iss1 >> sid >> y >> day;
428     cout << sid << "|" << y << "|" << day << endl;
429
430     istringstream iss2("123@123 0.6 2022-03-27");
431     iss2 >> sid >> y >> day;
432     cout << sid << "|" << y << "|" << day << endl;
```

问题 ⑫ 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ test
123456@123456|0.6|2022-03-27
123@123|0.6|2022-03-27

## How to split string using istringstream with other delimiter than whitespace?

```
#include <iostream>
#include <string>
#include <sstream>

int main()
{
    std::istringstream iss { "Cpp|is|fun" };

    std::string s;
    while ( std::getline( iss, s, '|' ) )
        std::cout << s << std::endl;

    return 0;
}
```

## Fastest way to split a string

<https://cplusplus.com/forum/beginner/114790/>

what is the fastest way to split a string with a delimiter into a vector/array with C/C++ and STL?

I already tried different approaches - including strtok and stringstream - but it's always terrible slow compared to Java String.split().

NwN:

I don't know if it's the fastest way - but using std::string::find\_first\_of gives you a complexity of O(n) with minimal memory overhead if you make use of c++11's move semantics:

```
void split(std::string& line, char delim, std::vector<std::string>& words)
{
    std::string word;
    std::istringstream iss(line);
    while(getline(iss, word, delim)){
        words.push_back(word);
    }
    std::string delims(1, delim);
    if(is_endwith(line, delims))
    {
        words.push_back("");
    }
}

void string_split(std::string& line, char delim, vector<string>& words)
{
    size_t start = 0;
    size_t end = line.find_first_of(delim);

    while (end <= std::string::npos)
    {
        words.emplace_back(line.substr(start, end - start));
        if (end == std::string::npos)
            break;
        start = end + 1;
        end = line.find_first_of(delim, start);
    }
}
```

我们测试对比一下上面两种实现：

```
(base) [root@bigdata-dev01 zacpp]# g++ t.cpp -o t -O3
(base) [root@bigdata-dev01 zacpp]# ./t
总耗时1:4985 微秒
总耗时2:2092 微秒
(base) [root@bigdata-dev01 zacpp]# ./t
总耗时1:5074 微秒
总耗时2:2106 微秒
(base) [root@bigdata-dev01 zacpp]# ./t
总耗时1:5427 微秒
总耗时2:2350 微秒
(base) [root@bigdata-dev01 zacpp]#
```

可以看到 string\_split 快大约 60%。

## C++ 的 string 为什么不提供 split 函数？

<https://www.zhihu.com/question/36642771>

在 C++ 编程中，领导坚持用 `char` 而不用 `string`, `string` 有那么可怕吗？

<https://www.zhihu.com/question/348023215>

作者：黄兢成

C++没有二进制标准。对象的二进制布局、链接名字、内存分配释放，每个编译器和标准库都有不同。假如一种编译器编译出库，导出 C++ 接口，在另一种编译器去使用，就算链接通过，运行时也可能出一些诡异的问题。为了避免这些问题，预先编译的库，无论内部实现是否使用 C++，通常会导出 C 风格的接口。

### How to inherit from std::ostream?

I want to define MyOStream which inherits publicly from std::ostream. Let's say I want to call foo() each time something is written into my stream.

```
class MyOStream : public ostream {
public:
    ...
private:
    void foo() { ... }
}
```

I understand that the public interface of ostream is non-virtual, so how can it be done? I want clients to be able to use both operator<< and write() and put() on MyOStream and have use the extended ability of my class.

Ben:

I was spinning my head around how to do the same thing and i found out it's actually not that hard. Basically just subclass the ostream and the streambuf objects, and construct the ostream with itself as the buffer. the virtual overflow() from std::streambuf will be called for every character sent to the stream. To fit your example i just made a foo() function and called it.



```
570 class MLOG{
571 public:
572     MLOG(){}
573     void print();
574
575     template<typename T>
576     std::ostringstream& operator<< (T val);
577
578 private:
579     std::ostringstream loss;
580 };
581
582 void MLOG::print(){
583     cout<<"["<<loss.str()<<"]"<<endl;
584     // 打印后立即清空流
585     loss.clear();
586     loss.str("");
587 }
588
589 template<typename T>
590 std::ostringstream& MLOG::operator<< (T val){
591     loss << val;
592     return loss;
593 }
594
595 int main(int argc, char * argv[]){
596     MLOG mylog = MLOG();
597     mylog << "hhah" << 12.0 ;
598     mylog.print();
599
600     mylog << 1111 << 2222 ;
601     mylog.print();
602
603 }
```

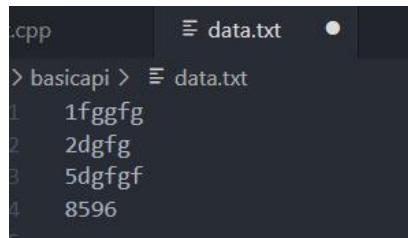
输出 [Running] cd "d:\wujianjun\code\my\zacpp\src\src" && [hhah] [11112222]

## 标准 API

**char \*fgets(char \*str, int n, FILE \*stream)**

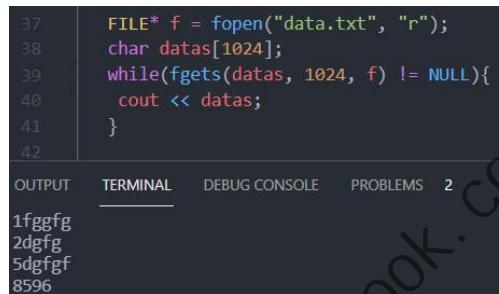
从指定的流 stream 读取一行，并把它存储在 str 所指向的字符串内。当读取 (n-1) 个字符时，或者读取到换行符时，或者到达文件末尾时，它会停止。

测试文件内容如下：



```
1fggfg
2dgfg
5dgfgf
8596
```

测试代码为：



```
37 FILE* f = fopen("data.txt", "r");
38 char datas[1024];
39 while(fgets(datas, 1024, f) != NULL){
40     cout << datas;
41 }
42
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2

```
1fggfg
2dgfg
5dgfgf
8596
```

**int sscanf(const char \*str, const char \*format, ...)**

从字符串 str 读取格式化 format 输入。



```
45 char lines[] = "123 xiaoming man 22 104.26";
46 int id;
47 char name[32];
48 char sex[32];
49 char age[32];
50 float salary;
51 sscanf(lines, "%d %s %s %s %f", &id, name, sex, age, &salary);
52 cout << id << endl;
53 cout << name << endl;
54 cout << sex << endl;
55 cout << age << endl;
56 cout << salary << endl;
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2

```
123
xiaoming
man
22
104.26
```

**int strncmp(const char \*str1, const char \*str2, size\_t n)**

把 str1 和 str2 进行比较，最多比较前 n 个字节。

该函数返回值如下：

如果返回值 < 0，则表示 str1 小于 str2。

如果返回值 > 0，则表示 str2 小于 str1。

如果返回值 = 0，则表示 str1 等于 str2。

**int strcmp ( const char \* str1, const char \* str2 );**

把 str1 所指向的字符串和 str2 所指向的字符串进行比较。

**int fscanf(FILE \*stream, const char \*format, ...)**

从流 stream 读取格式化输入。如果成功，该函数返回成功匹配和赋值的个数。如果到达文件末尾或发生读错误，则返回 EOF。

测试文件内容如下：



The terminal window shows the file 'data.txt' with the following content:

```
basicapi > cat data.txt
AA 10 6336.22
BB 2 55.62626
CC 899 0.6566
```

测试代码如下：



```
70 FILE* f = fopen("data.txt", "r");
71 for (int i=0; i<3; i++) { // 逐行读入矩阵
72     char name[32];
73     int id;
74     float pressure;
75     fscanf(f, "%s %d %f\n", name, &id, &pressure);
76     cout << name << " " << id << " " << pressure << endl;
77 }
78 
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2

```
AA 10 6336.22
BB 2 55.6263
CC 899 0.6566
```

`void *aligned_alloc( size_t alignment, size_t size )`

Defined in header <stdlib.h>

Allocate size bytes of uninitialized storage whose alignment is specified by alignment. **The size parameter must be an integral multiple of alignment.**

**The address of a block returned by `malloc` or `realloc` in GNU systems is always a multiple of eight (or sixteen on 64-bit systems). If you need a block whose address is a multiple of a higher power of two than that, use `aligned_alloc` or `posix_memalign`.**

As one example, I seem to recall that SSE2 instructions need their data aligned on 16-byte boundaries so you could use `aligned_alloc` to give you that even on systems where `malloc` only guarantees alignment to an 8-byte boundary.

`nullptr`

`void *aligned_alloc( size_t alignment, size_t size );`

**Allocate size bytes of uninitialized storage whose alignment is specified by alignment. The size parameter must be an integral multiple of alignment.**

`stdio.h`

定义了输入和输出的宏和函数。

- ◆ `EOF`: 这个宏是一个表示已经到达文件结束的负整数。
- ◆ `stderr`、`stdin` 和 `stdout`: 这些宏是指向 `FILE` 类型的指针，分别对应于标准错误、标准输入和标准输出流。
- ◆ `int fclose(FILE *stream)`: 关闭流 `stream`。刷新所有的缓冲区
- ◆ `FILE *fopen(const char *filename, const char *mode)`: 使用给定的模式 `mode` 打开 `filename` 所指向的文件。

`stdlib.h`

- ◆ `int atoi(const char *str)`: 把参数 `str` 所指向的字符串转换为一个整数（类型为 `int` 型）。
- ◆ `void *malloc(size_t size)`: 分配所需的内存空间，并返回一个指向它的指针。

- ◆ void free(void \*ptr): 释放之前调用 calloc、malloc 或 realloc 所分配的内存空间。
- ◆ int rand(void): 返回一个范围在 0 到 RAND\_MAX 之间的伪随机数。
- ◆ void srand(unsigned int seed): Initialize random number generator。

### cstdlib

经常发现有 #include <cstdlib> 的，也有 #include <stdlib.h> 的。

C 语言中是有 #include <stdlib.h> 的。在 C++ 中用 cstdlib，他实现了 stdlib.h 中的所有功能。  
即标准的 C++ 头文件没有.h 扩展名。

### string.h

- ◆ void \*memset(void \*str, int c, size\_t n): 复制字符 c (一个无符号字符) 到参数 str 所指向的字符串的前 n 个字符。
- ◆ int strcmp(const char \*str1, const char \*str2): 把 str1 所指向的字符串和 str2 所指向的字符串进行比较。
- ◆ char\* strcpy(char \*dest, const char \*src): 把 src 所指的由 NULL 结束的字符串复制到 dest 所指的数组中。src 和 dest 所指的内存区域不能重叠，且 dest 必须有足够的空间放置 src 所包含的字符串(包含结束符 NULL)，否则会造成缓冲溢出(buffer Overflow)。
- ◆ char\* strncpy(char \*dest, const char \*src, size\_t n): 将字符串 src 前 n 个字符拷贝到字符串 dest。如果 src 的前 n 个字节不含 NULL 字符，则结果不会以 NULL 字符结束。如果 src 的长度小于 n 个字节，则以 NULL 填充 dest 直到复制完 n 个字节。不像 strcpy(), strncpy() 不会向 dest 追加结束标记'\0'，这就引发了很多不合常理的问题。strncpy 并不帮你保证 \0 结束。如果 src 的内容比较少，而 n 又比较大的话，strncpy 将会把之间的空间都用\0 填充。这又出现了效率问题，如：strncpy(buf, "abcdefg", 79)，此时 strncpy 会填写 79 个 char，而不仅仅是"abcdefg"本身，还包含 71 个'\0'。
- ◆ void \*memcpy(void \*dest, const void \*src, size\_t n): 从存储 src 复制 n 个字节到存储区 dest。



```

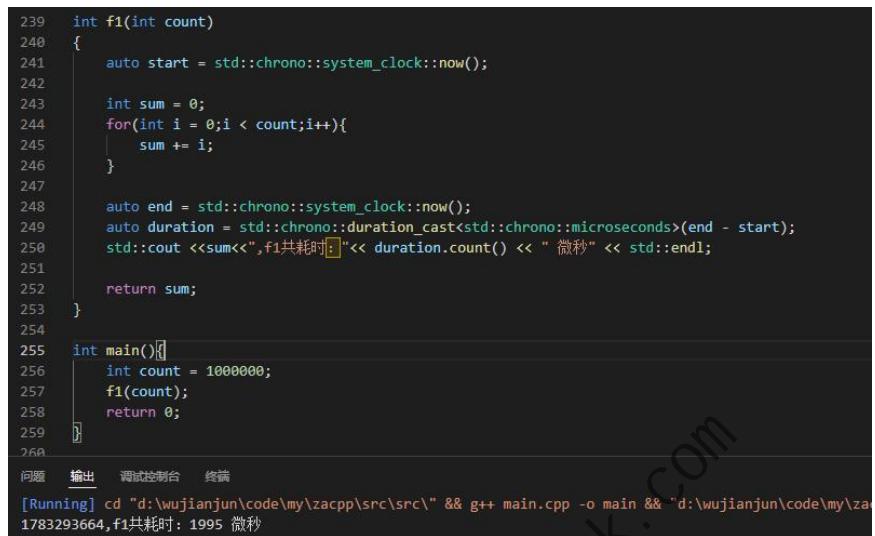
3675     float a = 0;
3676     float b = 10;
3677     memcpy(&a, &b, sizeof(float));
3678     cout << a << endl;
3679     return 0;
3680 }
```

问题 9 输出 调试控制台 终端 JUPYTER  
[Running] cd "d:\wujianjun\code\my\zacode"  
10

## 标准库详解

### 时间

std::chrono 是一个 time library。要使用 chrono 库，需要 #include<chrono>。  
chrono 是一个模版库，使用简单，功能强大，只需要理解三个概念：duration、time\_point、clock



```
239 int f1(int count)
240 {
241     auto start = std::chrono::system_clock::now();
242
243     int sum = 0;
244     for(int i = 0;i < count;i++){
245         sum += i;
246     }
247
248     auto end = std::chrono::system_clock::now();
249     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
250     std::cout << sum << ",f1共耗时:" << duration.count() << " 微秒" << std::endl;
251
252     return sum;
253 }
254
255 int main()
256 {
257     int count = 1000000;
258     f1(count);
259     return 0;
260 }
```

问题 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ main.cpp -o main && "d:\wujianjun\code\my\zacpp\src\src\main.exe"  
1783293664,f1共耗时: 1995 微秒

<time.h>有下面重要的函数：

- ◆ `char *strptime(const char *s, const char *format, struct tm * tm);`  
将字符串转换成 tm 结构。  
it converts the character string pointed to by `s` to values which are stored in the structure pointed to by `tm`, using the format specified by `format`.
- ◆ `size_t strftime(char *s, size_t max, const char *format,const struct tm *tm);`  
The strftime() function formats the tm according to the format specification format and places the result in the character array s of size max.
- ◆ 将 `time_t` 转为字符串。
- ◆ `time_t mktime(struct tm *timeptr)`  
把 `timeptr` 所指向的结构转换为自 1970 年 1 月 1 日以来持续时间的秒数。
- ◆ `double difftime(time_t time1, time_t time2);`  
返回 `time1` 和 `time2` 之间相差的秒数 (`time1 - time2`)
- ◆ `char *ctime(const time_t *time);`
- ◆ `struct tm *localtime(const time_t *timer);` 将 `time_t` 转换成 `tm` 结构时间。**localtime** 返回了一个 `tm` 指针，空间是由 `localtime` 自己控制的，所以如果连续调用这个函数会有问题。所以要记住，一旦调用了 `localtime` 函数，应该 马上取出 `tm` 结构中的内容。

## vector

c++ vector 删除指定元素

只使用 vector 的 erase 函数，**记住，该函数是迭代器失效，返回下一个迭代器。**

```
for(vector<int>::iterator niter = new_nums.begin();niter != new_nums.end();){  
    if(*niter == cur){  
        niter = new_nums.erase(niter);  
    }  
    else{  
        niter++;  
    }  
}
```

下面演示了 vector 切片操作

```
103 int main(){  
104     int n[] = {1, 2, 3, 4, 5, 6} ;  
105     int len = 5;  
106     vector<int> nums;  
107     for(int i=0; i < len; i++){  
108         nums.push_back(n[i]);  
109     }  
110     //print_vector(nums);  
111     for(int i =0; i < nums.size(); i++){  
112         vector<int> new_nums;  
113         new_nums.assign(&nums[0],&nums[i]); // 注意 i=0 是也将正常执行，只是结果为空  
114         vector<int> t_nums;  
115         t_nums.assign(&nums[i+1],&nums[nums.size()]); // 注意，这里并不会越界  
116         new_nums.insert(new_nums.end(),t_nums.begin(),t_nums.end());  
117         print_vector(new_nums);  
118     }  
119     /*  
120      vector<int> new_nums;  
*/  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL  
[Running] cd "d:\code\c++\vscode\leetcode\Permutations\" && g++ src.cpp -o src && "d:\code\leetcode\Permutations\src"  
2,3,4,5,  
1,3,4,5,  
1,2,4,5,  
1,2,3,5,  
1,2,3,4,
```

## vector 的地址

std::vector::data

Returns a direct pointer to the memory array used internally by the vector to store its owned elements. Because elements in the vector are guaranteed to be stored in contiguous storage locations in the same order as represented by the vector, the pointer retrieved can be offset to access any element in the array.

```
2469 |     vector<string> myv;  
2470 |     for(int i = 0 ;i < 10 ; i++)  
2471 |     {  
2472 |         for(int j = 0; j < 100 ; j++)  
2473 |         {  
2474 |             myv.push_back("A");  
2475 |         }  
2476 |     cout << myv.data() << endl;  
2477 | }
```

问题 8 编辑 调试控制台 终端 JUPYTER  
[Running] cd "d:\wujianjun\code\my\zancode\zacpp\zacd" && "d:\wujianjun\code\my\zancode\zacpp\zacd" && del zacd  
0x2682ca0  
0x1057f70  
0x1059f80  
0x1059f80  
0x1059f80  
0x2682490  
0x2682490  
0x2682490  
0x2682490  
0x2682490

可以看到随着数据的插入，vector 的地址在不停地变化。

## emplace\_back VS push\_back

都是向一个 vector 中增加一个元素。如果参数是左值，两个调用的都是 copy constructor  
如果参数是右值，两个调用的都是 move constructor

The image shows two side-by-side code editors. The left editor contains the definition of the `MyT` class with three constructors: copy constructor (left value), move constructor (const reference), and move constructor (non-const reference). The right editor shows the `main` function where `MyT` objects are added to vectors using both `emplace_back` and `push_back`. The output from both editors is identical, showing the sequence of copy and move operations.

```
2766 class MyT
2767 {
2768     public:
2769         MyT(string data_):data(data_){
2770             cout << "construct" << endl;
2771         }
2772         MyT(const MyT& obj){
2773             cout << "copy " << endl;
2774             this->data = obj.data;
2775         }
2776         MyT(MyT && obj){
2777             cout << "move " << endl;
2778             this->data = move(obj.data);
2779         }
2780     public:
2781         string data;
2782 };
2783 int main()
2784 {
2785     vector<MyT> vec1;
2786     vec1.reserve(3);
2787     vec1.emplace_back(MyT("a"));
2788     vec1.emplace_back(MyT("b"));
2789     vec1.emplace_back(MyT("c"));
2790     cout << "======" << endl;
2791     vector<MyT> vec2;
2792     vec2.reserve(3);
2793     vec2.push_back(MyT("b"));
2794     vec2.push_back(MyT("b"));
2795     vec2.push_back(MyT("c"));
2796 }
```

```
2768 class MyT
2769 {
2770     public:
2771         MyT(string data_):data(data_){
2772             cout << "construct" << endl;
2773         }
2774         MyT(const MyT& obj){
2775             cout << "copy " << endl;
2776             this->data = obj.data;
2777         }
2778         MyT(MyT && obj){
2779             cout << "move " << endl;
2780             this->data = move(obj.data);
2781         }
2782     public:
2783         string data;
2784 };
2785 int main()
2786 {
2787     MyT myt1("a");
2788     MyT myt2("b");
2789     MyT myt3("c");
2790     cout << "======" << endl;
2791     vector<MyT> vec1;
2792     vec1.reserve(3);
2793     vec1.emplace_back(myt1);
2794     vec1.emplace_back(myt2);
2795     vec1.emplace_back(myt3);
2796     cout << "======" << endl;
2797     vector<MyT> vec2;
2798     vec2.reserve(3);
2799     vec2.push_back(myt1);
2800     vec2.push_back(myt2);
2801     vec2.push_back(myt3);
2802 }
```

看起来二者没有区别。

## Best way to extract a subvector from a vector?

The image shows a code editor with a red arrow pointing to a section of code. The code defines a `MyT` class and demonstrates how to extract a subvector from a larger vector. The mistake shown is using a range-based for loop to iterate over a subvector's elements, which creates multiple copies of the elements. A red arrow points to the line `for(auto const& item : vec2){`.

```
2766 class MyT
2767 {
2768     public:
2769         MyT(string data_):data(data_){
2770             cout << "construct" << endl;
2771         }
2772         MyT(const MyT& obj){
2773             cout << "copy " << endl;
2774             this->data = obj.data;
2775         }
2776         MyT(MyT && obj){
2777             cout << "move " << endl;
2778             this->data = move(obj.data);
2779         }
2780     public:
2781         string data;
2782 };
2783 int main()
2784 {
2785     vector<MyT> vec1;
2786     vec1.reserve(5);
2787     vec1.emplace_back(MyT("a"));
2788     vec1.emplace_back(MyT("b"));
2789     vec1.emplace_back(MyT("c"));
2790     vec1.emplace_back(MyT("d"));
2791     cout << "======" << endl;
2792     vector<MyT> vec2(vec1.begin() + 1 , vec1.begin() + 4 );
2793     cout << "======" << endl;
2794     for(auto const& item : vec2){
2795         cout << item.data << ",";
2796     }
2797     cout << endl;
2798 }
```

## map 与 unordered\_map

<https://zhuanlan.zhihu.com/p/48066839>

**map 是基于红黑树实现。**红黑树作为一种自平衡二叉树，保障了良好的最坏情况运行时间，即它可以做到在  $O(\log n)$  时间内完成查找，插入和删除。另红黑树是一种二叉查找树，二叉查找树一个重要的性质是有序，且中序遍历时取出的元素是有序的。

**unordered\_map 是基于 hash\_table 实现，一般是由一个大 vector, vector 元素节点可挂接链表来解决冲突来实现。**hash\_table 最大的优点，就是把数据的存储和查找消耗的时间大大降低，几乎可以看成是常数时间；而代价仅仅是消耗比较多的内存。

至于实际性能对比，这里不妨写测试来看看。

先看看我们实现了 timer 类：

```
const int num = 100000;
// clock() 在 C 标准库 time.h 中，可以返回一段代码所耗的 CPU tick 数。
// CLOCKS_PER_SEC 是标准 C 的 time.h 中定义的一个常数，表示一秒钟内 CPU 运行的时钟周期数。
class timer {
public:
    clock_t start, end;
    string name;
    timer(string n) { // 本对象构造时记录时间起点
        start = clock(); name = n;
    }
    ~timer() { // 本对象析构时计算时间终点，打印耗时!!!!
        end = clock();
        printf("%s time: %.1f (ms)\n", name.c_str(), (end - start) * 1.0 / CLOCKS_PER_SEC * 1000);
    }
};
```

测试与结果代码如下：

```
447 template<typename T>
448 void insert(T & conta, string name) {
449     srand((unsigned)time(NULL));
450     timer t1(name);
451     for (int i = 0; i < num / 2; i++) {
452         int key = rand();
453         conta.insert(pair<int, int>(i, i));
454         conta.insert(pair<int, int>(key, i));
455     }
456 }
457
458 template<typename T>
459 void find(T & conta, string name) {
460     srand((unsigned)time(NULL));
461     timer t1(name);
462     for (int i = 0; i < num / 2; i++) {
463         int key = rand();
464         conta.find(key);
465         conta.find(i);
466     }
467 }
468
469 void test_map() {
470     map<int, int> m1;
471     insert<map<int, int>>(m1, "map insert");
472     find<map<int, int>>(m1, "map find");
473 }
474
475 void test_unordered_map() {
476     unordered_map<int, int> m2;
477     insert<unordered_map<int, int>>(m2, "unordered_map insert");
478     find<unordered_map<int, int>>(m2, "unordered_map find");
479 }
480
481 int main(){
482     test_map();
483     test_unordered_map();
484 }
```

问题 ① 输出 调试控制台 终端

```
[Running] cd "d:\wujianjun\code\my\zapp\src\src\" && g++ main.cpp -o main
map insert time: 42.0 (ms)
map find time: 29.0 (ms)
unordered_map insert time: 24.0 (ms)
unordered_map find time: 7.0 (ms)
```

可以看到 map 查找耗时是 unordered\_map 的 4 倍，insert 也是将近两倍。

另外内存占用情况为：map 内存 5096K，unordered\_map 内存 6712K。

## unordered\_map

Unordered map is an associative container that contains key-value pairs **with unique keys**.

**Internally, the elements are organized into buckets. Which bucket an element is placed into depends entirely on the hash of its key. Keys with the same hash code appear in the same bucket.** 当 bucket 内数据量在 8 以内使用链表来实现，数据量大于 8 则转换为红黑树结构。

unordered\_map 用到自定义的类型，需要对 key 定义 hash\_value 函数并且重载 operator ==

- ◆ template<class Key, class T, class Hash = std::hash<Key>> class unordered\_map;
- ◆ size(): 有多少 pair 对被 insert 到容器中。
- ◆ bucket( const key\_type& k ): Returns the index of the bucket for key k. Buckets are numbered from 0 to (bucket\_count-1).
- ◆ bucket\_size( size\_type n ): Returns the number of elements in the bucket with index n.
- ◆ **bucket\_count(): indicates how many buckets the underlying hash table has.**
- ◆ **load\_factor(): Returns the average number of elements per bucket 即 size() /bucket\_count().**
- ◆ **max\_load\_factor()/max\_load\_factor( float ml ): Manages the maximum load factor (number of elements per bucket). The container automatically increases the number of buckets if the load factor exceeds this threshold.** 默认值是 1,也就是每个 key 一个桶。

```
void print_map(unordered_map<string, string> data_map) // 遍历每个bucket
{
    int bkt_num = data_map.bucket_count();
    hash<string> hasher;
    for(int i=0; i < bkt_num ; i++)
    {
        cout << "bucket: " << setw(3) << setfill('0') << i
            << ", size=" << data_map.bucket_size(i) << ", data: [";
        for(auto iter = data_map.begin(i); iter != data_map.end(i); iter++) // 注意每个bucket的遍历方法
        {
            string key = (*iter).first;
            string value = (*iter).second;
            int bkt_index = hasher(key) % bkt_num; // 注意两个key的插号的计算方法!!!!!!
            cout << setw(3) << key << "-" << setw(3) << value << "|" << bkt_index << ",";
        }
        cout << "]" << endl;
    }
}

int main()
{
    int ele_num = 20;
    unordered_map<string, string> raw_profile_data;
    raw_profile_data.max_load_factor(2);
    for(int i = 0 ; i < ele_num / 3; i++)
    {
        raw_profile_data[to_string(i)] = to_string(i);
    }
    cout << "======" << endl;
    print_map(raw_profile_data);
    // 两轮insert之间存在重建hash表
    for(int i = ele_num / 3 ; i < ele_num; i++)
    {
        raw_profile_data[to_string(i)] = to_string(i);
    }
    cout << "======" << endl;
    print_map(raw_profile_data);
}
```

运行结果如下：可以看见很多元素在两轮 insert 之间被移动了(002 从桶 1 移动到桶 0 了)。

```
=====
bucket 000, size=0, data: []
bucket 001, size=1, data: [002->002|1,]
bucket 002, size=0, data: []
bucket 003, size=2, data: [005->005|3,001->001|3,]
bucket 004, size=2, data: [006->006|4,004->004|4,]
bucket 005, size=0, data: []
bucket 006, size=1, data: [003->003|6,]

=====
bucket 000, size=2, data: [002->002|0,001->001|0,]
bucket 001, size=1, data: [007->007|1,]
bucket 002, size=1, data: [000->000|2,]
bucket 003, size=0, data: []
bucket 004, size=2, data: [005->005|4,008->008|4,]
bucket 005, size=1, data: [016->016|5,]
bucket 006, size=3, data: [004->004|6,011->011|5,009->009|6,]
bucket 007, size=1, data: [015->015|7,]
bucket 008, size=1, data: [010->010|8,]
bucket 009, size=1, data: [019->019|9,]
bucket 010, size=2, data: [006->006|10,012->012|10,]
bucket 011, size=0, data: []
bucket 012, size=1, data: [014->014|12,]
bucket 013, size=3, data: [018->018|13,013->013|13,003->003|13,]
bucket 014, size=0, data: []
bucket 015, size=0, data: []
bucket 016, size=1, data: [017->017|16,]
```

- ◆ `void rehash( size_type count )`: Sets the number of buckets to count and rehashes the container, i.e. puts the elements into appropriate buckets considering that total number of buckets has changed.
  - 若参数 `count` 大于容器中的当前桶数，则将强制进行重建哈希表。新的桶数将等于或大于 `count`。
  - 若参数 `count` 小于容器中当前的桶数，则该函数可能对桶数没有影响，可能不会强制进行重建哈希表。

**当负载系数(load factor)大于阈值，容器便会自动执行重建哈希表(rehash)操作。rehash 操作时存在旧地址数据拷贝到新地址，及旧地址销毁、更新地址指向的过程。**

- ◆ `reserve`: Sets the number of buckets to the number needed to accomodate at least count elements without exceeding maximum load factor and rehashes the container, i.e. puts the elements into appropriate buckets considering that total number of buckets has changed. Effectively calls `rehash(std::ceil(count / max_load_factor()))`.

**在实际使用中如果需要存储有大量数据，频繁的 rehash 会非常影响性能。解决办法是在 `unordered_map` 建立时根据实际需要预先设定桶数和元素数避免后期可能的 rehash。**

```

2963 | int main()
2964 | {
2965 |     unordered_map<int, string> raw_profile_data;
2966 |     for(int i = 0 ; i < 10; i++)
2967 |     {
2968 |         cout << "-----" << endl;
2969 |         raw_profile_data[i] = to_string(i);
2970 |         cout << raw_profile_data.size() << "\t"
2971 |             << raw_profile_data.bucket_count() << "\t"
2972 |             << raw_profile_data.load_factor() << "\t"
2973 |             << raw_profile_data.max_load_factor() << "\t"
2974 |             << raw_profile_data.max_load_factor() * raw_profile_data.bucket_count()
2975 |             << endl;
2976 |     }
2977 | }

```

问题 输出 调试控制台 终端 JUPYTER  
[Running] cd "d:\wujianjun\code\my\zacpp\zacpp\src\" && g++ test.cpp -o test && "

```

=====
1 3 0.333333 1 3
=====
2 3 0.666667 1 3
=====
3 7 0.428571 1 7
=====
4 7 0.571429 1 7
=====
5 7 0.714286 1 7
=====
6 7 0.857143 1 7
7 17 0.411765 1 17
=====
8 17 0.470588 1 17
=====
9 17 0.529412 1 17
=====
10 17 0.588235 1 17
=====
```

可以看到：

- ◆ 随着插入，`bucket` 的数量会自动增加，增加不止一倍，比如:3->7->17->37->79，且增加时，可能会做内存拷贝，并且释放的内存不会还给 OS，而是被 STL 保留下来了。
- ◆ 冲突的概率很小：也就是多少个 `key` 基本就至少有多少个 `bucket`。

容器会自动增加桶数，以此将 `load_factor` 保持在 `max_load_factor` 以下。每次需要扩充桶数时都会 `rehash()`。

```

2963 int main()
2964 {
2965     unordered_map<int, string> raw_profile_data;
2966     raw_profile_data.rehash(20);
2967     for(int i = 0 ; i < 10; i++)
2968     {
2969         cout << "-----" << endl;
2970         raw_profile_data[i] = to_string(i);
2971         cout << raw_profile_data.size() << "\t"
2972             << raw_profile_data.bucket_count() << "\t"
2973             << raw_profile_data.load_factor() << "\t"
2974             << raw_profile_data.max_load_factor() << "\t"
2975             << raw_profile_data.max_load_factor() * raw_profile_data.bucket_count()
2976             << endl;
2977     }
2978 }

问题 ① 输出 调试控制台 终端 JUPYTER
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\src" && g++ test.cpp -o test &&
=====
1 23 0.0434783 1 23
2 23 0.0869565 1 23
3 23 0.130435 1 23
4 23 0.173913 1 23
5 23 0.217391 1 23
6 23 0.260871 23
7 23 0.304348 1 23
8 23 0.347826 1 23
9 23 0.391384 1 23
10 23 0.434783 1 23
=====

2963 int main()
2964 {
2965     unordered_map<int, string> raw_profile_data;
2966     raw_profile_data.reserve(10);
2967     for(int i = 0 ; i < 10; i++)
2968     {
2969         cout << "-----" << endl;
2970         raw_profile_data[i] = to_string(i);
2971         cout << raw_profile_data.size() << "\t"
2972             << raw_profile_data.bucket_count() << "\t"
2973             << raw_profile_data.load_factor() << "\t"
2974             << raw_profile_data.max_load_factor() << "\t"
2975             << raw_profile_data.max_load_factor() * raw_profile_data.bucket_count()
2976             << endl;
2977     }
2978 }

问题 ① 输出 调试控制台 终端 JUPYTER
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\src" && g++ test.cpp -o test &&
=====
1 23 0.0434783 23
2 23 0.0869565 1 23
3 23 0.130435 1 23
4 23 0.173913 1 23
5 23 0.217391 1 23
6 23 0.260871 23
7 23 0.304348 1 23
8 23 0.347826 1 23
9 23 0.391384 1 23
10 23 0.434783 1 23
=====
```

可以看到通过 `rehash` 和 `reserve` 都可以预先申请好 bucket，这样似乎可以免于重建 hash 表。

```

2963 int main()
2964 {
2965     unordered_map<int, string> raw_profile_data;
2966     raw_profile_data.max_load_factor(2);
2967     for(int i = 0 ; i < 10; i++)
2968     {
2969         cout << "-----" << endl;
2970         raw_profile_data[i] = to_string(i);
2971         cout << raw_profile_data.size() << "\t"
2972             << raw_profile_data.bucket_count() << "\t"
2973             << raw_profile_data.load_factor() << "\t"
2974             << raw_profile_data.max_load_factor() << "\t"
2975             << raw_profile_data.max_load_factor() * raw_profile_data.bucket_count()
2976             << endl;
2977     }
2978 }

问题 ① 嵌出 调试控制台 终端 JUPYTER
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\src" && g++ test.cpp -o test && "d:\w
=====
1 1 1 2 2
2 3 0.666667 2 6
3 3 1 2 6
4 3 1.33333 2 6
5 3 1.66667 2 6
6 7 0.857143 2 14
7 7 1 2 14
8 7 1.14286 2 14
9 7 1.28571 2 14
10 7 1.42857 2 14
=====
```

可以看到，也可以增大 `max_load_factor` 来减少 bucket 的使用量。

我们现在来看看内存增长策略：

首先看看如何不加干预时的增长过程：

```

int main()
{
    int pre_vm_size = get_vm_size();
    int pre_pm_size = get_pm_size();
    cout << "初始虚拟内存占用:" << pre_vm_size << "KB" << ",初始物理内存占用" << pre_pm_size << "KB" << endl;
    int num = 5000000;
    unordered_map<int, string> raw_profile_data;
    raw_profile_data.reserve(num);
    for(int i = 0 ; i < num; i++)
    {
        raw_profile_data[i] = to_string(i);
        int cur_vm_size = get_vm_size();
        int cur_pm_size = get_pm_size();
        if(cur_vm_size > pre_vm_size*1.1)
        {
            cout << fixed << setprecision(4)
                << "虚拟内存:" << setw(8) << pre_vm_size << "KB" ->> setw(8) << cur_vm_size << "KB,\t\t"
                << "物理内存:" << setw(8) << pre_pm_size << "KB" ->> setw(8) << cur_pm_size << "KB,\t\t"
                << raw_profile_data.size() << "\t\t"
                << raw_profile_data.bucket_count() << "\t\t"
                << raw_profile_data.load_factor() << "\t\t"
                << raw_profile_data.max_load_factor() << "\t\t"
                << raw_profile_data.max_load_factor() * raw_profile_data.bucket_count()
                << endl;

            pre_vm_size = cur_vm_size;
            pre_pm_size = cur_pm_size;
        }
    }
    cout << "最终虚拟内存占用:" << get_vm_size() << "KB" << ",最终物理内存占用" << get_pm_size() << "KB" << endl;
    cout << "insert finished!!!!!!!!!!!!!!" << endl;
    std::this_thread::sleep_for(std::chrono::seconds(60));
}
```

加入 reserve 前后的运行日志如下：

```
(base) [root@dm-devel wujianjun]# cat t.log
初始虚拟内存占用:12700KB,初始物理内存占用1176KB
虚拟内存: 12700KB -> 1398KB, 物理内存: 1176KB -> 2560KB, 14910 15173 0.9827 1.0000 15173.0000
虚拟内存: 1398KB -> 1549KB, 物理内存: 2560KB -> 4208KB, 30727 62233 0.4937 1.0000 62233.0000
虚拟内存: 1549KB -> 1708KB, 物理内存: 4208KB -> 5692KB, 49758 62233 0.7995 1.0000 62233.0000
虚拟内存: 1708KB -> 1890KB, 物理内存: 5692KB -> 7512KB, 66654 126271 0.5279 1.0000 126271.0000
虚拟内存: 1890KB -> 2088KB, 物理内存: 7512KB -> 9492KB, 91998 126271 0.7286 1.0000 126271.0000
虚拟内存: 2088KB -> 22992KB, 物理内存: 9492KB -> 11696KB, 119932 126271 0.9427 1.0000 126271.0000
虚拟内存: 22992KB -> 25329KB, 物理内存: 11696KB -> 13944KB, 135928 256279 0.5304 1.0000 256279.0000
虚拟内存: 25329KB -> 27969KB, 物理内存: 13944KB -> 16536KB, 16977 256279 0.6652 1.0000 256279.0000
虚拟内存: 27969KB -> 30404KB, 物理内存: 16536KB -> 19408KB, 206901 256279 0.0073 1.0000 256279.0000
虚拟内存: 30404KB -> 32404KB, 物理内存: 19408KB -> 22656KB, 247442 256279 0.9855 1.0000 256279.0000
虚拟内存: 32404KB -> 37556KB, 物理内存: 22656KB -> 26172KB, 266027 520241 0.5114 1.0000 520241.0000
虚拟内存: 37556KB -> 41384KB, 物理内存: 26172KB -> 30000KB, 315026 520241 0.6955 1.0000 520241.0000
虚拟内存: 41384KB -> 45609KB, 物理内存: 30000KB -> 34224KB, 369093 520241 0.7095 1.0000 520241.0000
虚拟内存: 45609KB -> 50229KB, 物理内存: 34224KB -> 38844KB, 428229 520241 0.8231 1.0000 520241.0000
虚拟内存: 50229KB -> 55376KB, 物理内存: 38844KB -> 43992KB, 494123 520241 0.9498 1.0000 520241.0000
虚拟内存: 55376KB -> 61544KB, 物理内存: 43992KB -> 50229KB, 520241 1056323 0.4925 1.0000 1056323.0000
虚拟内存: 61544KB -> 67749KB, 物理内存: 50229KB -> 56360KB, 598078 1056323 0.5669 1.0000 1056323.0000
虚拟内存: 67749KB -> 74612KB, 物理内存: 56360KB -> 63228KB, 686738 1056323 0.6501 1.0000 1056323.0000
虚拟内存: 74612KB -> 82136KB, 物理内存: 63228KB -> 70752KB, 783045 1056323 0.7413 1.0000 1056323.0000
虚拟内存: 82136KB -> 90452KB, 物理内存: 70752KB -> 79068KB, 889490 1056323 0.8421 1.0000 1056323.0000
虚拟内存: 90452KB -> 99569KB, 物理内存: 79068KB -> 88176KB, 1086972 1056323 0.9524 1.0000 1056323.0000
虚拟内存: 99569KB -> 11696KB, 物理内存: 88176KB -> 106940KB, 1056323 2144977 0.4925 1.0000 2144977.0000
虚拟内存: 11696KB -> 123112KB, 物理内存: 106940KB -> 1176KB, 110894 2144977 0.5598 1.0000 2144977.0000
虚拟内存: 123112KB -> 135202KB, 物理内存: 1176KB -> 124136KB, 1367599 2144977 0.6329 1.0000 2144977.0000
虚拟内存: 135202KB -> 149116KB, 物理内存: 124136KB -> 137732KB, 1531538 2144977 0.7140 1.0000 2144977.0000
虚拟内存: 149116KB -> 164032KB, 物理内存: 137732KB -> 152644KB, 1722462 2144977 0.8030 1.0000 2144977.0000
虚拟内存: 164032KB -> 180532KB, 物理内存: 152644KB -> 169144KB, 1933662 2144977 0.9015 1.0000 2144977.0000
虚拟内存: 180532KB -> 214304KB, 物理内存: 169144KB -> 202929KB, 2144977 4355707 0.4925 1.0000 4355707.0000
虚拟内存: 214304KB -> 235820KB, 物理内存: 202929KB -> 224436KB, 2420267 4355707 0.5557 1.0000 4355707.0000
虚拟内存: 235820KB -> 259449KB, 物理内存: 224436KB -> 248064KB, 2722706 4355707 0.6251 1.0000 4355707.0000
虚拟内存: 259449KB -> 285452KB, 物理内存: 248064KB -> 274066KB, 3055557 4355707 0.7015 1.0000 4355707.0000
虚拟内存: 285452KB -> 314096KB, 物理内存: 274066KB -> 327012KB, 3422200 4355707 0.7857 1.0000 4355707.0000
虚拟内存: 314096KB -> 345512KB, 物理内存: 327012KB -> 334128KB, 3824325 4355707 0.8780 1.0000 4355707.0000
虚拟内存: 345512KB -> 380096KB, 物理内存: 334128KB -> 368712KB, 4267000 4355707 0.9796 1.0000 4355707.0000
虚拟内存: 380096KB -> 422032KB, 物理内存: 368712KB -> 410712KB, 4355707 8844859 0.4925 1.0000 8844859.0000
虚拟内存: 422032KB -> 442742KB, 物理内存: 410712KB -> 452888KB, 4895531 8844859 0.5535 1.0000 8844859.0000
最终虚拟内存占用:472324KB,物理内存占用46104KB
insert finished!!!!!!!!!!!!!!
```

```
(base) [root@dm-devel wujianjun]# cat t.log
初始虚拟内存占用:12704KB,初始物理内存占用1172KB
虚拟内存: 12704KB -> 52540KB, 物理内存: 1172KB -> 41104KB, 1 5098259 0.0000 1.0000 5098259.0000
虚拟内存: 52540KB -> 57920KB, 物理内存: 41104KB -> 46532KB, 68857 5098259 0.0135 1.0000 5098259.0000
虚拟内存: 57920KB -> 63808KB, 物理内存: 46532KB -> 52416KB, 144223 5098259 0.0283 1.0000 5098259.0000
虚拟内存: 63808KB -> 70208KB, 物理内存: 52416KB -> 58816KB, 226143 5098259 0.0444 1.0000 5098259.0000
虚拟内存: 70208KB -> 77248KB, 物理内存: 58816KB -> 65856KB, 316255 5098259 0.0620 1.0000 5098259.0000
虚拟内存: 77248KB -> 85656KB, 物理内存: 65856KB -> 73668KB, 416198 5098259 0.0816 1.0000 5098259.0000
虚拟内存: 85656KB -> 93632KB, 物理内存: 73668KB -> 82244KB, 525971 5098259 0.1032 1.0000 5098259.0000
虚拟内存: 93632KB -> 103104KB, 物理内存: 82244KB -> 91716KB, 647212 5098259 0.1269 1.0000 5098259.0000
虚拟内存: 103104KB -> 113472KB, 物理内存: 91716KB -> 162084KB, 779923 5098259 0.1530 1.0000 5098259.0000
虚拟内存: 113472KB -> 124864KB, 物理内存: 162084KB -> 113476KB, 925740 5098259 0.1816 1.0000 5098259.0000
虚拟内存: 124864KB -> 137408KB, 物理内存: 113476KB -> 126016KB, 1086303 5098259 0.2131 1.0000 5098259.0000
虚拟内存: 137408KB -> 151232KB, 物理内存: 126016KB -> 139844KB, 1263251 5098259 0.2478 1.0000 5098259.0000
虚拟内存: 151232KB -> 166464KB, 物理内存: 139844KB -> 155076KB, 1458220 5098259 0.2860 1.0000 5098259.0000
虚拟内存: 166464KB -> 183232KB, 物理内存: 155076KB -> 171844KB, 1672851 5098259 0.3281 1.0000 5098259.0000
虚拟内存: 183232KB -> 201664KB, 物理内存: 171844KB -> 190276KB, 1908780 5098259 0.3744 1.0000 5098259.0000
虚拟内存: 201664KB -> 211888KB, 物理内存: 190276KB -> 210496KB, 2167547 5098259 0.4252 1.0000 5098259.0000
虚拟内存: 211888KB -> 244160KB, 物理内存: 210496KB -> 232772KB, 2452729 5098259 0.4811 1.0000 5098259.0000
虚拟内存: 244160KB -> 268608KB, 物理内存: 232772KB -> 257216KB, 2756563 5098259 0.5425 1.0000 5098259.0000
虚拟内存: 268608KB -> 295488KB, 物理内存: 257216KB -> 284096KB, 3109772 5098259 0.6100 1.0000 5098259.0000
虚拟内存: 295488KB -> 325056KB, 物理内存: 284096KB -> 313668KB, 3488198 5098259 0.6842 1.0000 5098259.0000
虚拟内存: 325056KB -> 357568KB, 物理内存: 313668KB -> 346176KB, 3940351 5098259 0.7658 1.0000 5098259.0000
虚拟内存: 357568KB -> 393408KB, 物理内存: 346176KB -> 382016KB, 4363163 5098259 0.8558 1.0000 5098259.0000
虚拟内存: 393408KB -> 432832KB, 物理内存: 382016KB -> 421444KB, 4867731 5098259 0.9548 1.0000 5098259.0000
最终虚拟内存占用:443068KB,物理内存占用431776KB
insert finished!!!!!!!!!!!!!!
```

可以看见，内存扩张的次数减少了约 40%，但是最终占用的内存当然不会减少。

我们现在来研究 clear 之后能不能重用以前的内存空间：

```
int pre_vm_size = get_vm_size();
int pre_pm_size = get_pm_size();
cout << "初始虚拟内存占用:" << pre_vm_size << "KB" << ",初始物理内存占用:" << pre_pm_size << "KB" << endl;
int num = 5000000;
unordered_map<string, string> raw_profile_data;
raw_profile_data.reserve(num);
for(int i = 0 ; i < num; i++)
{
    raw_profile_data[to_string(i)] = to_string(i);
    int cur_vm_size = get_vm_size();
    int cur_pm_size = get_pm_size();
    if(cur_vm_size > pre_vm_size*1.1)
    {
        cout << fixed << setprecision(4)
        << "虚拟内存:" << setw(8) << pre_vm_size << "KB" -> << setw(8) << cur_vm_size << "KB" \t \t
        << "物理内存:" << setw(8) << pre_pm_size << "KB" -> << setw(8) << cur_pm_size << "KB" \t \t
        << raw_profile_data.size() << "\t\t"
        << raw_profile_data.bucket_count() << "\t\t"
        << raw_profile_data.load_factor() << "\t\t" << raw_profile_data.max_load_factor() << "\t\t"
        << endl;
        pre_vm_size = cur_vm_size;
        pre_pm_size = cur_pm_size;
    }
}
cout << "insert finished!!!!!!!!!!!!!" << endl;
cout << "insert后虚拟内存占用:" << get_vm_size() << "KB" << ",物理内存占用"<< get_pm_size() << "KB" << endl;

// 再次插入一些新的值
raw_profile_data.clear();
for(int i = num ; i < num + 5000000; i++)
{
    raw_profile_data[to_string(i)] = to_string(i);
}
cout << "add finished!!!!!!!!!!!!!" << endl;
cout << "add后虚拟内存占用:" << get_vm_size() << "KB" << ",物理内存占用"<< get_pm_size() << "KB" << endl;
```

运行日志为：

```
insert finished!!!!!!!!!!!!!
insert后虚拟内存占用:755644KB,物理内存占用744284KB
add finished!!!!!!!!!!!!!
add后虚拟内存占用:755644KB,物理内存占用744284KB
add
```

可以看出，内存并没有增长。

我们使用新的 string 的 hash 函数，并增大冲突因子：

```
struct StringHasher {
    size_t operator()(const std::string& t) const {
        return stol(t) % 1000000;
    }
};

int main()
{
    int pre_vm_size = get_vm_size();
    int pre_pm_size = get_pm_size();
    cout << "初始虚拟内存占用:" << pre_vm_size << "KB" << ", 初始物理内存占用" << pre_pm_size << "KB" << endl;
    int num = 5000000;
    unordered_map<string, string, StringHasher> raw_profile_data;
    raw_profile_data.max_load_factor(10);
    raw_profile_data.reserve(num);
    for(int i = 0 ; i < num; i++)
    {
        raw_profile_data[to_string(i)] = to_string(i);
        int cur_vm_size = get_vm_size();
        int cur_pm_size = get_pm_size();
        if(cur_vm_size > pre_vm_size*1.1)
        {
            cout << fixed << setprecision(4)
            << "虚拟内存:" << setw(8) << pre_vm_size << "KB" ->> setw(8) << cur_vm_size << "KB" \t\ t
            << "物理内存:" << setw(8) << pre_pm_size << "KB" ->> setw(8) << cur_pm_size << "KB" \t\ t
            << raw_profile_data.size() << "\t\t"
            << raw_profile_data.bucket_count() << "\t\t"
            << raw_profile_data.load_factor() << "\t\t" << raw_profile_data.max_load_factor() << "\t\t"
            << endl;
        }
    }
}
```

运行日志如下：

```
虚拟内存: 368524KB -> 405388KB,          物理内存: 357140KB -> 394004KB,           2763459  520241   5.3119  10.0000
虚拟内存: 405388KB -> 445964KB,          物理内存: 394004KB -> 434584KB,           3052000  520241   5.0665  10.0000
虚拟内存: 445964KB -> 490639KB,          物理内存: 434584KB -> 479252KB,           3396667  520241   6.4771  10.0000
虚拟内存: 490639KB -> 539788KB,          物理内存: 479252KB -> 528408KB,           3719193  520241   7.1490  10.0000
虚拟内存: 539788KB -> 597044KB,          物理内存: 528408KB -> 582424KB,           4103807  520241   7.8873  10.0000
虚拟内存: 597044KB -> 653196KB,          物理内存: 582424KB -> 641816KB,           4523650  520241   8.6991  10.0000
虚拟内存: 653196KB -> 718604KB,          物理内存: 641816KB -> 707224KB,           4990775  520241   9.5952  10.0000
insert后虚拟内存占用:719889KB,物理内存占用708520KB
add finished!!!!!!!!!!!!!!
add后虚拟内存占用:719880KB,物理内存占用708520KB
```

内存使用量减少了一点。

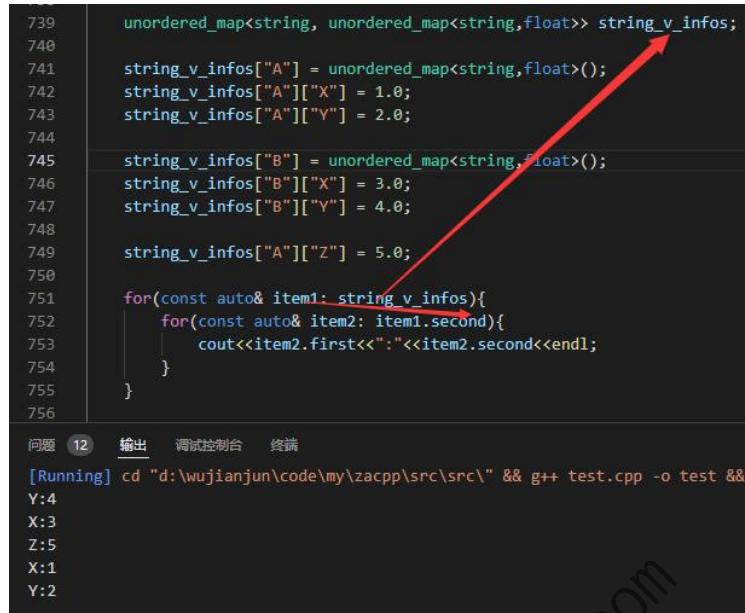
- ◆ `emplace( Args&&... args )`: Inserts a new element into the container constructed in-place with the given args if there is no element with the key in the container. **Careful use of emplace allows avoiding unnecessary copy or move operations.**
- ◆ `count( const K& x )`: Returns the number of elements with key that compares equal to the specified argument key, which is either 1 or 0 since this container does not allow duplicates.

```
494     class Info {
495     public:
496         std::string name;
497         Info(){}
498         Info(std::string name_):name(name_) { }
499         Info(const Info& info){[this->name=info.name];}
500         ~Info(){}
501     };
502
503     int main(){
504         std::unordered_map<std::string, Info> dict;
505
506         // 几种不同插入方式
507         dict["A"] = Info("AAA");
508         dict.emplace("B",Info("BBB"));
509         dict.insert(std::make_pair("C",Info("CCC")));
510
511         // 遍历
512         for(const auto &item: dict){
513             std::cout<<item.first<<"."<<item.second.name<<std::endl;
514         }
515
516         // 查找:
517         std::cout<< dict["A"].name<<std::endl;
518         std::cout<< dict.count("C")<<","<< dict.count("D")<<std::endl;
519
520         // 排序
521         std::cout<<=====<<std::endl;
522         std::vector<std::pair<std::string, Info>> elems(dict.begin(), dict.end());
523         std::sort(elems.begin(), elems.end(), [] (const auto &x, const auto &y) {return x.first < y.first;});
524         for(const auto &item: elems){
525             std::cout<<item.first<<"."<<item.second.name<<std::endl;
526         }
527     }
528 }
```

问题 12 编辑 调试控制台 终端  
Running: cd d:\wujianjun\code\my\zapp\src\src\ -&& g++ test.cpp -o test && d:\wujianjun\code\my\zapp\src\src\ test  
C:CCC  
A:AAA  
B:BBB  
AAA  
1,0  
=====  
A:AAA  
B:BBB  
C:CCC

**map** 本身不支持 **sort**, 应将 **map** 中的元素 **pair** 保存在 **vector**, 然后排序。

多层 **map** 的操作示例如下:



```
739     unordered_map<string, unordered_map<string, float>> string_v_infos;
740
741     string_v_infos["A"] = unordered_map<string, float>();
742     string_v_infos["A"]["X"] = 1.0;
743     string_v_infos["A"]["Y"] = 2.0;
744
745     string_v_infos["B"] = unordered_map<string, float>();
746     string_v_infos["B"]["X"] = 3.0;
747     string_v_infos["B"]["Y"] = 4.0;
748
749     string_v_infos["A"]["Z"] = 5.0;
750
751     for(const auto& item1: string_v_infos){
752         for(const auto& item2: item1.second){
753             cout<<item2.first<<":"<<item2.second<<endl;
754         }
755     }
756 
```

问题 12 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ test.cpp -o test &&  
Y:4  
X:3  
Z:5  
X:1  
Y:2

注意, 看下面的例子:



```
2183     unordered_map<string, string> map_t;
2184     cout<< " | " << map_t.count("A") << " | " << endl;
2185     cout<< " | " << map_t["A"] << " | " << endl;
2186     cout<< " | " << map_t.count("A") << " | " << endl;
```

问题 3 调试控制台 终端 JUPYTER  
[Running] cd "d:\wujianjun\code\my\zancode\zacpp\zacpp\src\"  
|0|  
||  
|1|

也就是[]访问时如果不存在对应的 key, 就会创建一个, value 采用默认的。

`std::unordered_map<Key,T,Hash,KeyEqual,Allocator>::operator[]`

T& operator[]( const Key& key ); (1) (since C++11)  
T& operator[]( Key&& key ); (2) (since C++11)

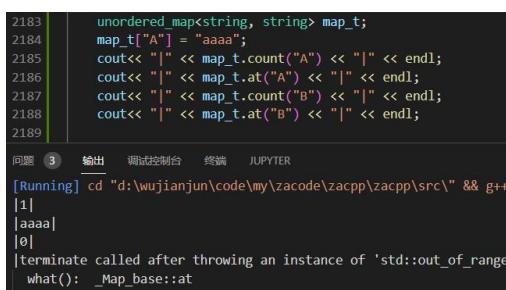
Returns a reference to the value that is mapped to a key equivalent to key, performing an insertion if such key does not already exist.

应该用 **at** 访问, 就不会创建了 :

`std::unordered_map<Key,T,Hash,KeyEqual,Allocator>::at`

T& at( const Key& key ); (1) (since C++11)  
const T& at( const Key& key ) const; (2) (since C++11)

Returns a reference to the mapped value of the element with key equivalent to key. If no such element exists, an exception of type `std::out_of_range` is thrown.



```
2183     unordered_map<string, string> map_t;
2184     map_t["A"] = "aaaa";
2185     cout<< " | " << map_t.count("A") << " | " << endl;
2186     cout<< " | " << map_t.at("A") << " | " << endl;
2187     cout<< " | " << map_t.count("B") << " | " << endl;
2188     cout<< " | " << map_t.at("B") << " | " << endl;
2189 
```

问题 3 调试控制台 终端 JUPYTER  
[Running] cd "d:\wujianjun\code\my\zancode\zacpp\zacpp\src\" && g++  
|1|  
|aaaa|  
|0|  
|terminate called after throwing an instance of 'std::out\_of\_range'  
what(): \_Map\_base::at

注意, []和 at 返回的都是引用!!!!

读时应该一律用 at!!!!只有赋值时才可以用[]

## What is the difference between `unordered_map::emplace` and `unordered_map::insert` in C++?

Chris Drew:

- ◆ `unordered_map::insert` copies or moves a key-value pair into the container.
- ◆ `unordered_map::emplace` allows you to avoid unnecessary copies or moves by constructing the element in place.

use of `emplace` doesn't guarantee you will avoid copies or moves.

*In principle, emplacement functions should sometimes be more efficient than their insertion counterparts, and they should never be less efficient.*

( Howard Hinnant ran some experiments that showed sometimes `insert` is faster than `emplace`)

Most implementations of `unordered_map::emplace` will cause memory to be dynamically allocated for the new pair even if the map contains an item with that key already and the `emplace` will fail. This means that if there is a good chance that an `emplace` will fail you may get better performance using `insert` to avoid unnecessary dynamic memory allocations.

```
#include <unordered_map>
#include <iostream>

struct Employee {
    std::string firstname;
    std::string lastname;
    Employee(const std::string& firstname, const std::string& lastname)
        : firstname(firstname), lastname(lastname){}
};

int main() {
    auto employees = std::unordered_map<int, Employee>{};
    auto employee1 = std::pair<int, Employee>{1, Employee{"John", "Smith"}};

    employees.insert(employee1); // copy insertion
    employees.insert(std::make_pair(2, Employee{"Mary", "Jones"})); // move insertion
    employees.emplace(3, Employee("Sam", "Thomas")); // emplace with pre-constructed Employee
    employees.emplace(std::piecewise_construct,
                      std::forward_as_tuple(4),
                      std::forward_as_tuple("James", "Brown")); // construct in-place
}
```

## Why is `std::unordered_map` slow, and can I use it more effectively to alleviate that?

einpoklum:

The standard library's maps are, indeed, inherently slow (`std::map` especially but `std::unordered_map` as well). Google's Chandler Carruth explains this in his CppCon 2014 talk; in a nutshell: *`std::unordered_map` is cache-unfriendly because it uses linked lists as buckets.*

This SO question mentioned some efficient hash map implementations - use one of those instead

### Super high performance C/C++ hash map (table, dictionary) [closed]

Scharron:

I would recommend you to try [Google SparseHash](#) (or the C11 version Google SparseHash-c11) and see if it suits your needs. They have a memory efficient implementation as well as one optimized for speed. I did a benchmark a long time ago, it was the best hashtable implementation available in terms of speed (however with drawbacks).

## Google Sparse Hash 的使用

clone 源代码: <https://github.com/sparsehash/sparsehash>

解压后, 进入目录, ./configure && make && make install

编写示例代码, 如下:

```
#include <iostream>
#include <google/sparse_hash_map>

struct eqstr
{
    bool operator()(const char* s1, const char* s2) const
    {
        return (s1 == s2) || (s1 && s2 && strcmp(s1, s2) == 0);
    }
};

int main()
{
    google::sparse_hash_map<const char*, int, std::hash<const char*>, eqstr> months;
    months.set_deleted_key(NULL);
    months["april"] = 30;
    months["june"] = 30;
    months["september"] = 30;
    months["november"] = 30;

    std::cout << "september -> " << months["september"] << std::endl;
    std::cout << "april -> " << months["april"] << std::endl;
    std::cout << "june -> " << months["june"] << std::endl;
    std::cout << "november -> " << months["november"] << std::endl;
}
```

## std::unordered\_map 复制耗时

发现 insert 很慢。

```
2989 class TCleaner
2990 {
2991     public:
2992     TCleaner() { gender_id2str_dict["0"] = "男"; gender_id2str_dict["1"] = "女"; }
2993     inline string id2tr(string& gender) { return gender_id2str_dict[gender]; }
2994     public:
2995         unordered_map<string, string> gender_id2str_dict;
2996     };
2997
2998 void test(vector<unordered_map<string, string>>& raw_datas, TCleaner& cleaner, vector<unordered_map<string, string>>& res)
2999 {
3000     for(int i=0; i<1000; i++)
3001     {
3002         unordered_map<string, string> cur_r;
3003         cur_r.reserve(400);
3004         for(int j=0; j<400; j++)
3005         {
3006             string c = cleaner.id2tr(raw_datas[i][to_string(j)]);
3007             //cur_r[to_string(j)] = c; // 这句话很耗时!!!!注释掉后,耗时能从309减少到117
3008         }
3009         res.emplace_back(cur_r);
3010     }
3011 }
3012
3013 int main()
3014 {
3015     vector<unordered_map<string, string>> datas;
3016     for(int i=0; i<1000; i++)
3017     {
3018         datas.emplace_back(unordered_map<string, string>());
3019         for(int j=0; j<400; j++)
3020         {
3021             datas[i][to_string(j)] = "0";
3022         }
3023     }
3024     TCleaner cleaner = TCleaner();
3025     auto start1 = chrono::system_clock::now(); // 结束时间
3026     vector<unordered_map<string, string>> res;
3027     res.reserve(1000);
3028     test(datas, cleaner, res);
3029     auto end1 = chrono::system_clock::now(); // 结束时间
3030     auto duration1 = chrono::duration_cast<chrono::milliseconds>(end1 - start1);
3031     cout << "总耗时:" << duration1.count() << " 毫秒" << endl;
3032 }
3033
3034 [问题 1] 输出 调试控制台 终端 JUPITER
3035 [Running] cd "d:\wujianjun\code\my\zacode\zacpp\src" && g++ test.cpp -o test && "d:\wujianjun\code\my\zacode\zacpp\test"
3036 总耗时1:306 毫秒
3037
3038 [Done] exited with code=0 in 1.532 seconds
3039
3040 [Running] cd "d:\wujianjun\code\my\zacode\zacpp\src" && g++ test.cpp -o test && "d:\wujianjun\code\my\zacode\zacpp\test"
3041 总耗时1:116 毫秒
```

## Is std::unordered\_set contiguous (like std::vector)?

<https://stackoverflow.com/questions/14384668/is-stdunordered-set-contiguous-like-stdvector>

Matthieu M.

The exact implementation of containers is not detailed by the standard... however the standard does prescribes a number of behaviors which constrains the actual representation.

For example, `std::unordered_set` is required to be memory stable: a reference to/address of an element is valid even when adding/removing other elements. The only way to achieve this is by allocating elements more or less independently. It cannot be achieved with a contiguous memory allocation as such an allocation would necessarily be bounded, and thus could be overgrown with no possibility of re-allocating the elements in a bigger chunk.

## std::algorithm

transform 的例子如下：

```
716 int main(int argc, char * argv[]){
717
718     vector<pair<string,int>> vs;
719     vs.push_back(make_pair("A",1));
720     vs.push_back(make_pair("B",2));
721     vs.push_back(make_pair("C",3));
722     for(const auto& item:vs){
723         cout<<item.first<<"|<<item.second<<endl;
724     }
725     cout<<"-----"<<endl;
726     vector<string> nvs;
727     std::transform(
728         vs.begin(),vs.end(),
729         // back_inserter用于在末尾插入元素。可以使用back_inserter的容器是有push_back成员函数的容器
730         std::back_inserter(nvs),
731         [] (const pair<string,int> item) -> string { return item.first; }
732     );
733     for(const auto& item:nvs){
734         cout<<item<<endl;
735     }
736 }
```

问题 12 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ test.cpp -o test && "d:\wujianjun\code\my\z  
A|1  
B|2  
C|3  
-----  
A  
B  
C

- ◆ **find\_if:** 根据指定的查找规则，在指定区域内查找第一个符合该函数要求的元素。

*InputIterator find\_if (InputIterator first, InputIterator last, UnaryPredicate pred);*

```
//以函数对象的形式定义一个查找规则
class mycomp2 {
public:
    bool operator()(const int& i) {
        return ((i % 2) == 1);
    }
};

int main() {
    vector<int> myvector{ 4,2,3,1,5 };
    vector<int>::iterator it = find_if(myvector.begin(), myvector.end(), mycomp2());
    cout << "*it = " << *it;
    return 0;
}
```

其底层实现的参考代码如下：

```
template<class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred)
{
    while (first!=last) {
        if (pred(*first)) return first;
        ++first;
    }
    return last;
}
```

- ◆ **find\_if\_not:** 用于查找第一个不符合谓词函数规则的元素。

*InputIterator find\_if\_not (InputIterator first, InputIterator last, UnaryPredicate pred);*

**int isspace( int ch );**

Checks if the given character is whitespace character.the whitespace characters are the following:

- ◆ space (0x20, ' ')
- ◆ form feed (0x0c, '\f')
- ◆ line feed (0x0a, '\n')
- ◆ carriage return (0x0d, '\r')
- ◆ horizontal tab (0x09, '\t')
- ◆ vertical tab (0x0b, '\v')

The behavior is undefined if the value of ch is not representable as unsigned char and is not equal to EOF.

Return value non-zero value if the character is a whitespace character, zero otherwise.

## ◆ copy

```
copy (InputIterator first, InputIterator last, OutputIterator result);
```

## std::array

std::array 是在 C++11 标准中增加的 STL 容器，它的设计目的是提供与原生数组类似的功能与性能。也正因此，使得 std::array 有很多与其他容器不同的特殊之处，**比如：std::array 的元素是直接存放在实例内部，而不是在堆上分配空间；std::array 的大小必须在编译期确定；当定义一个 array 时，除了指定元素类型，还要指定容器大小。array 不能被动态地扩展或压缩。不能添加或删除元素。**

一个默认构造的 array 是非空的：它包含了与其大小一样多的元素。这些元素都被默认初始化，就像一个内置数组中的元素那样。

## random

<https://cplusplus.com/reference/random/>

先看个例子：

```
2831 #include <iostream>
2832 #include <random>
2833 using namespace std;
2834
2835 int main()
2836 {
2837     std::random_device dev;
2838     std::mt19937 rng(dev());
2839     std::uniform_int_distribution<std::mt19937::result_type> dist6(1,6); // distribution in range [1, 6]
2840     for(int i =0; i<5 ;i++)
2841     {
2842         std::cout << dist6(rng) << std::endl;
2843     }
2844 }
```

输出

```
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\src\" && g++ test.cpp -o test && "d:\wujianjun\code\my\zacode\zacpp\src\test"
4
5
3
6
1
```

我们再做下性能对比测试：

```
(base) [root@VM-40-14-centos wujianjun]# vi t.cpp
(base) [root@VM-40-14-centos wujianjun]# g++ t.cpp -O3
(base) [root@VM-40-14-centos wujianjun]# ./t
stdmap vs tslmap耗时:4470 vs3879 微秒
stdmap vs tslmap耗时:3672vs 3441 微秒
(base) [root@VM-40-14-centos wujianjun]# ./t
stdmap vs tslmap耗时:83560 vs 64247 微秒
stdmap vs tslmap耗时:677 vs 296 微秒
```

结论：

- 1.key 很多时，性能差别才明显，比如上百万个 key。
- 2.随机查询时，大概快 20%，顺序查询时大概快一倍。

## C++和STL中有哪些副作用或者稍不注意会产生性能开销的地方？

作者：陈硕

range-based for loop，最好用 auto，否则容易有额外的拷贝。

```
map<string, int> word_count;
for (const auto& kv : word_count) {
    // kv.first, kv.second
}
```

如果你想把 auto 的类型写出来，一定不要忘了 key 是 const。

```
for (const std::pair<const std::string, int>& kv : word_count) {
```

如果你写成了 pair<string, int>，会造成额外的拷贝：

```
// 低效的写法,
for (const std::pair<std::string, int>& kv : word_count) {
```

作者：果冻虾仁

- ◆ STL 容器的 clear 的时间复杂度不是 O(1)。不管是序列容器(比如 vector)还是关联容器(比如 unordered\_map)，执行 clear() 就需要对其存储的元素调用析构函数，这个析构操作显然是逐个析构的，因而时间复杂度是 O(n)。但是，也有例外，比如当 vector 存储基本数据类型或 POD 类型(比如基本数据类型构成的 struct)的时候，由于其元素类型没有析构函数(也不需要析构函数)，加之 vector 内部连续存储的特性，编译器可能可以在常量时间完成 clear() 的。
- ◆ auto 替代手写类型。这个其他答主也讲到了。
- ◆ 减少隐性的重复操作。从 map 中查找 value。我们一般怎么写呢？比如：

```
// dict_data是一个unordered_map<string, double>
// vec是一个vector<string>
double sum = 0;
for (auto& key : vec) {
    if (dict_data.count(key)) { // 或 dict_data[key] > 0
        sum += dict_data[key]; // 或 sum += dict_data.at(key);
    }
}
```

或者：

```
for (auto& key : vec) {
    if (dict_data.find(key) != dict_data.end()) {
        sum += dict_data[key]; // 或 sum += dict_data.at(key);
    }
}
```

其实 map 或 unordered\_map 的[] 或者 at() 函数内部也会进行查找。既然我们已经查找过一次 key 是否存在了，那么就吧结果存储下来就好了。

```
for (auto& key : vec) {
    auto it = dict_data.find(key);
    if (it != dict_data.end()) {
        sum += it->second;
    }
}
```

- ◆ 我不鼓励在生产环境中使用会抛异常的函数。因为 C++ 不同于 java，java 如果有未捕获或 throw 的异常，编译都过不了。而 C++ 则不管，所以如果你的代码不小心抛出了异常，而没被 catch，那么就可能让程序 core dump

```
try {
    ...
    auto& e = v.at[index];
    // do sth for e
    ...
} catch (...) {
}
```

- ◆ sort 给定义对象排序可能存在对象拷贝的开销。比如对于一个 vector 进行排序。

```
std::sort(elems.begin(), elems.end(),
          [] (const auto &x, const auto &y) {return x.first < y.first;});
```

**当你的自定义类型没有移动构造函数的时候，调用的是拷贝构造函数！最好给你的自定义对象添加一个移动构造函数。**当然如果你不想这么麻烦的话，那么用 vector 存储该类型的指针，然后传入一个该类型指针进行比较大小的 lambda 表达式。如果你想拥有 N 个元素的 vector 排序，然后取出 K 个元素。STL 的算法中还有一个 **partial\_sort**，只帮助你找到最大(或最小)的 K 个元素，而不需要把整个 vector 变得有序。

- ◆ shared\_ptr 修改指向时有时是有开销的。shared\_ptr 用起来和普通指针类似，还不要手工管理释放内存。**我们应该大量使用 shared\_ptr 替代普通指针，以规避各种各样的内存问题。**当一个已经存在的 shared\_ptr 修改指向的时候：

```
shared_ptr<T> p4 = p2;
...
p4 = p1;
```

如果 p4 是最后一个持有旧对象的 shared\_ptr 的话，那么当 p4 = p1 的时候，不止是触发 shared\_ptr 成员变量复制操作。还会触发旧对象的析构操作！**我们知道 shared\_ptr 在所管理的数据对应的引用计数清零的时候，会触发析构操作。**

- ◆ clear()不会清空 vector 的内存。尽管 clear()会调用 vector 中元素的析构函数，但是并不会释放掉元素所占用的内存。如果你想在 vector 生命周期结束之前及时释放掉 vector 的内存，请：`vector<int>().swap(v);`
- ◆ 因为 size() 返回是无符号整型，当 vector 是空的时候，size()返回 0，无符号的 0 减去 1，得到的是一个极大的正数。

```
for (int i = 0; i <= v.size() - 1; ++i) {
    auto& e = v[i];
    // do sth for e
    ...
}
```

- ◆ 一写多读 STL 容器也不是线程安全的。大家都知道并发多个线程去写 STL 容器(写指的是插入新元素)不是线程安全的，可能会触发 core dump。然而一写多读，也不是线程安全的。比如 vector，尽管只有一个线程来写入，但是如果他触发了扩容了。那么其他线程不管是只读这个 vector 的，其中的迭代器也会失效。对于 unordered\_map 也是类似，单线程不停插入元素的话，可能触发 rehash，导致其他线程中在 map 中 find 的过程中 core dump.

# c++内存管理

## 内存管理入门

### 基本概念

C/C++程序地址空间由低到高依次为：

- ◆ **Code Segment**: 存放着程序的机器码和只读数据(如字符串常量, `const` 常量)。这个段在内存中一般被标记为只读, 任何对该区的写操作都会导致段错误(Segmentation Fault)。
- ◆ **Data Segment**: 存放已初始化的全局或静态变量。
- ◆ **BSS** 中存放未初始化的全局或静态变量。
- ◆ **Heap**: 堆的大小并不固定, 可动态扩张或缩减。其分配由 `malloc()`、`new()` 等这类实时内存分配函数来实现。
- ◆ **Stack**: 用来存储函数调用时的临时信息, 如函数调用所传递的参数、函数的返回地址、函数的局部变量等。在程序运行时由编译器分配和清除。

内存分配方式有三种：

- ◆ **静态存储**。内存在程序编译的时候就已经分配好, 这块内存在程序的整个运行期间都存在。例如全局变量, `static` 变量。
- ◆ **栈**创建。在执行函数时, 函数内局部变量的存储单元都可以在栈上创建, 函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中, 效率很高, 但是分配的内存容量有限。
- ◆ **堆**分配。程序在运行的时候用 `malloc` 或 `new` 申请任意多少的内存, 程序自己负责在何时用 `free` 或 `delete` 释放内存。但如果在堆上分配了空间, 就有责任回收它, 否则运行的程序会出现内存泄漏, 频繁地分配和释放不同大小的堆空间将会产生**堆内碎块**。

**内存泄漏**就是 `new` 了一段堆空间, 不再需要时候没有 `delete` 它, 或者在没有 `delete` 之前, 指针又指向了另外一个地址, 这样先前的空间就丢失了。

- ◆ `delete p` 意思是释放了 `p` 所指的目标所占的堆空间, 而不删除 `p` 本身(该指针所占内存空间并未释放), 所以 `delete p` 之后要让 `p=NULL`。
- ◆ 对于指向数组指针, 如 `int[] p`, `delete p` 只释放第一个元素的内存, `delete [] p` 才释放全部内存。
- ◆ `delete` 只能释放堆空间。如果 `new` 返回的指针值丢失, 则所分配的堆空间无法回收, 称内存泄漏。同一空间重复释放也是危险的, 因为该空间可能已另分配。

**内存越界**是指向系统申请一块内存后, 使用时却超出申请范围。如果在之前你的程序运行一切正常, 但因为你新增了几个类的成员变量或者修改了一部分代码而导致程序发生错误, 则因考虑是否是内存被破坏的原因了, 重点排查内存是否越界。

**内存屏障**其实质是 CPU 对 cache 的使用导致对内存的操作不能够及时的反映出来。CPU 一般情况下是不直接读写内存的, 而是:

**读取内存数据到 cache -> CPU 读取 cache-> CPU 的计算->将结果写入 cache->写回数据到内存**  
这有可能导致一个问题, cache 中的数据和内存实际的数据不一致, 当多线程情况下, 有可能会读“脏数据”。这就是所谓的“内存屏障”。

**不要在栈上定义大数组/大对象**, 系统栈空间很小。应该改为堆从堆中分配的, 不过记得 `delete` 你申请的堆空间。通常, 栈的缺省大小为 8M。`alloca(size_t size)`**在栈上申请空间**, The `alloca()` function allocates `size` bytes of space in the stack frame of the caller. **This temporary space is automatically freed when the function that called alloca() returns to its caller.**

## 作用域与内存回收

局部变量在离开作用范围后，分配的内存会被系统自动回收。

`new` 出来的内存空间存放在堆中，不受作用域管理，不会被系统自动回收，只有在使用 `delete` 删除或者整个程序结束后才会释放内存。

- ◆ 智能指针的出现则是在指针被销毁的情况下，也会销毁指针指向的内存，避免内存泄漏发生。

析构函数在下边 3 种情况时被调用：

- ◆ 对象生命周期结束时。
- ◆ `delete` 指向对象的指针时，或者 `delete` 指向对象的基类类型的指针。
- ◆ 对象 A 是对象 B 的成员，B 的析构函数被调用时，对象 A 的析构函数也会被调用。

注意：

- ◆ 当对象的引用超出作用域(或者删除对象的引用)时，不会运行析构函数。

## STL 内存管理

### vector 内存要点

vector 的 **具体数据存放在堆区**, 变量名存放在栈区。容器均调用了 `allocator` 来创建。

**vector 的内存空间只会增长不会减少。** vector 有两个函数, 一个是 `capacity()`, 返回对象缓冲区(vector 维护的内存空间)实际申请的空间大小; 另一个 `size()`, 返回当前对象缓冲区存储数据的个数。`capacity` 是永远大于等于 `size` 的。

在调用 `push_back` 时, 若当前容量已经不能够放入新元素, 那么 `vector` 会重新申请一块内存, 把之前的内存里的元素拷贝到新的内存当中, 然后把 `push_back` 的元素拷贝到新的内存中, 最后要析构原有的 `vector` 并释放原有的内存。

**clear() 和 erase(), 只是使得容器内部的对象通通析构, 减少了 size(), 清除了数据, 并不会减少 capacity, 内存空间没有减少, 释放内存应该使用 swap() 操作:**

```
vector<Point>().swap(pointVec);
```

使用 `vector` 的默认构造函数建立临时对象, 再调用 `swap`, `swap` 使 `vector` 离开其自身的作用域, 从而强制释放 `vector` 所占的内存空间。

**如果 `vector` 中存放的是指针, 那么当 `vector` 销毁时, 这些指针指向的对象不会被自动释放。**

### STL 是否会产生内存碎片?

`vector` 是顺序存储的, 不会生产内存碎片; 但是 `list` 和 `map` 是非连续存储的, 会产生内存碎片。只要有 `new`, 就会有内存碎片, 就看内部是怎么管理的。

**STL 的内存分配器 allocator, 用了内存池。相对来说, 内存管理还是不错的。**

### Vector 内存分配与释放

<https://zhuanlan.zhihu.com/p/338390842>

相关函数:

- ◆ `reserve( size_type new_cap )`: 预留空间。`reserve` 函数用来给 `vector` 预分配存储区大小, 即 `capacity` 的值, 但是没有给这段内存进行初始化。参数 `n` 是推荐预分配内存的大小, 实际分配的可能等于或大于这个值。`reserve` 函数分配出来的内存空间, 只是表示 `vector` 可以利用这部分内存, 但 `vector` 不能有效地访问这些内存空间, 会导致程序崩溃。
- ◆ `void resize( size_type count, T value = T() )`: 如果当前的容器长度大于新的长度值, 则该容器后部的元素会被删除(`remove and destroy`); 如果当前的容器长度小于新的长度值, 则系统会在该容器后部添加新元素, 并对这些元素进行初始化。

`vector` 为了支持快速的随机访问, `vector` 容器的元素以连续方式存放。

**所有内存空间是在 `vector` 析构时候才能被系统回收。**

### 关于 c++ 中 map 的内存占用问题

很多人说 `map` 中的 `erase` 以及 `clear` 不能释放内存, 这个说法是不确切的。**应该是, `map` 中的 `erase` 以及 `clear` 不能“马上”释放内存, `map` 有自己的机制回收内存, `map` 不会马上释放删掉内容的内存, 而是会对内存进行预留, 如果很长时间用不到预留的内存, 才会释放。**

### STL 容器的内存消耗问题

我们的进程会吃掉很多内存, 根据 `gperftools` (google 出品的一个性能分析工具) 观察, 发现:

问题 1: `map/set unordered_map/unordered_set` 占用内存高, 有效载荷低。

- ◆ `vector`、`deque` 这两个数据结构的内存额外开销可以忽略。
- ◆ **set/map 的内存开销实际上是非常大的。以一个 4B 的有效载荷为例, 一个 item 占用了 40B, 整整有 9 倍的额外消耗。**

问题 2 的分析: `unordered_map/unordered_set` 插入后, 再擦除, 内存不会回收。

可以调用 `rehash` 来重新设置桶数并重哈希容器。也就是重新放置新元素。其中满足 `size() < count * max_load_factor()`。在调用 `rehash` 之后, 内存从 143M 下降到 88B。

## 谨慎使用 STL

<https://www.jianshu.com/p/5539ad303d28>

- ◆ STL 的内存分配器分多级，对于申请较大的内存，它会在堆上去申请。
- ◆ **STL 的内存释放，有时候并没有直接返还给 os，只是返还给了分配器。**
- ◆ 针对大量数据，谨慎大量使用 STL 局部变量，虽然栈上分配的，但它维护的队列是分配在 heap 上的，它操作的内存不一定能够即使释放，可能产生碎片。
- ◆ malloc()/free()作为 C 标准，ANSI C 并没有指定它们具体应该如何实现。各个平台上 (windows, mac, linux 等等)，调用这两个函数时，实现不一样。
- ◆ 在 linux 下，malloc()/free()的实现是由 glibc 库负责的。

## How to determine CPU and memory consumption from inside a process

<https://stackoverflow.com/questions/63166/how-to-determine-cpu-and-memory-consumption-from-inside-a-process>

## map 内存彻底释放方法

```
#include <malloc.h>
int malloc_trim(size_t pad);
```

glibc 采用的是 dlmalloc。为了避免频繁调用系统调用，它内部维护了一个内存池，方便 reuse，又称为 free-list。所有调用 delete 释放的内存，并不是立即归还给操作系统，而是先将这个内存块挂在 free-list 里面，然后进行内存归并，并检查是否达到 malloc\_trim 的 threshold，如果达到了，则调用 malloc\_trim 归还部分可用内存给操作系统。glibc 中，设置了默认进行 malloc\_trim 的 threshold 为 128K，也就是说当 dlmalloc 管理的内存池中最大可用内存>128K 时，就会执行 malloc\_trim 操作，归还部分内存给操作系统；而在可用内存<=128K 时，及时程序中 delete 了这部分内存，这些内存也是不会归还给操作系统的。

malloc\_trim(0)可以立即执行 trim 操作，将内存还给操作系统。

## Does std::vector use stack or heap memory and why?

Jerry Coffin:

The *vector* itself is allocated wherever you allocate it.

*vector* doesn't allocate the space directly. Rather, it uses an *Allocator* object to allocate the space. If you want, you can specify an allocator, which allows you to control where the space is allocated (doing so is a fairly common optimization). The default *Allocator*, *std::allocator<T>*, allocates space from the free store using *operator new*.

## STL 在多线程环境下的内存管理

我们写个小程序来揭示一个有意思的现象，程序如下：

```
class ProfileSrv{
public:
    ProfileSrv(int interval_):interval(interval_){
        paths = {"./data/dm/ZaMainpageTa/pz/20220914/data",
                 "./data/dm/ZaMainpageTa/pz/20220915/data",
                 "./data/dm/ZaMainpageTa/pz/20220916/data"};
        cur_pi = 0;
        read_profile_files("./data/dm/ZaMainpageTa/pz/20220916/data"); // 这句话注释前后内存使用大不一样
    }
    void start_timer(){
        std::thread t([=](){
            while (true) {
                cout << "Timer:" << ", start running ....." << endl;
                read_profile_files(paths[cur_pi % 3]);
                cur_pi += 1;
                cout << "Timer:" << ", finised running ....." << endl;
                std::this_thread::sleep_for(std::chrono::seconds(this->interval));
            }
        });
        t.detach();
    }
    void read_profile_files(string zipped_pdata_path);
    int cur_raw_profile_data;
    inline unordered_map<string, string>& writable_raw_profile_data(){
        if(cur_raw_profile_data == 1){ return raw_profile_data_0;}
        else{ return raw_profile_data_1;}
    }
public:
    int interval; int cur_pi;
    vector<string> paths;
    unordered_map<string, string> raw_profile_data_0; unordered_map<string, string> raw_profile_data_1;
};
void ProfileSrv::read_profile_files(string zipped_pdata_path){
    vector<string> filenames = {zipped_pdata_path + "/part-00000"};
    istringstream liss;
    string line;
    string id;
    string feature;
    unordered_map<string, string>& raw_profile_data = writable_raw_profile_data();
    raw_profile_data.clear(); // 注意首先删除!!!!!!
    for (const auto& filename : filenames){
        cout << "reading raw profile :" << filename << endl;
        ifstream infile(filename);
        while (getline(infile, line)){
            liss.clear();
            liss.str(line);
            liss >> id >> feature;
            raw_profile_data[id] = feature;
        }
    }
    cur_raw_profile_data = (cur_raw_profile_data + 1) % 2;
}
int main(int argc, char * argv[]){
    ProfileSrv profileSrv(3);
    profileSrv.start_timer();
    this_thread::sleep_for(std::chrono::seconds(3000));
}
```

注释前的运行截图为

```
(base) [root@dm-dev01 wujianjun]# g++ t.cpp -lpthread -o stl_thrad_mtest
(base) [root@dm-dev01 wujianjun]# ./stl_thrad_mtest
reading raw profile :./data/dm/ZaMainpageTa/pz/20220916/data/part-00000
Timer: start running .....
Timer: finished running .....
reading raw profile :./data/dm/ZaMainpageTa/pz/20220914/data/part-00000
Timer: start running .....
Timer: finished running .....
reading raw profile :./data/dm/ZaMainpageTa/pz/20220915/data/part-00000
Timer: start running .....
Timer: finished running .....
[1]
```

top - 14:52:32 up 55 days, 2:31, 7 users, load average: 0.00, 0.01, 0.05								
Tasks:	1	total.	0	running,	1	sleeping,	0	stopped,
%Cpu(s):	0.1	us.	0.1	sy.	0.0	ni.	99.8	id.
0.0	wa.	0.0	hi.	0.0	si.	0.0	st	
KiB Mem:	65807304	total.	14386916	free,	17869824	used,	33551364	buff/cache
KiB Swap:	0	total.	0	free,	0	used.	47370468	avail Mem
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU %MEM
26885	root	20	0	517164	468208	1156	S	0.0 0.7
TIME+ COMMAND								
0:02.22	stl_thrad_mtest							

注释后的运行截图为

```
(base) [root@dm-dev01 wujianjun]# g++ t.cpp -lpthread -o stl_thrad_mtest
(base) [root@dm-dev01 wujianjun]# ./stl_thrad_mtest
Timer: start running .....
Timer: finished running .....
reading raw profile :./data/dm/ZaMainpageTa/pz/20220914/data/part-00000
Timer: start running .....
Timer: finished running .....
reading raw profile :./data/dm/ZaMainpageTa/pz/20220915/data/part-00000
Timer: start running .....
Timer: finished running .....
reading raw profile :./data/dm/ZaMainpageTa/pz/20220916/data/part-00000
Timer: start running .....
Timer: finished running .....
[1]
```

top - 14:54:20 up 55 days, 2:33, 7 users, load average: 0.00, 0.01, 0.05								
Tasks:	1	total.	0	running,	1	sleeping,	0	stopped,
%Cpu(s):	0.0	us.	0.1	sy.	0.0	ni.	99.9	id.
0.0	wa.	0.0	hi.	0.0	si.	0.0	st	
KiB Mem:	65807304	total.	14616866	free,	17639864	used,	33551380	buff/cache
KiB Swap:	0	total.	0	free,	0	used.	47599628	avail Mem
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU %MEM
26964	root	20	0	289328	246408	1064	S	0.0 0.4
TIME+ COMMAND								
0:02.28	stl_thrad_mtest							

也就是说：如果初始化 ProfileSrv 的时候读入 20220916 这个文件夹，然后再定时线程先后读入 20220914 和 20220915 两个文件夹，此时进程占用的物理内存为 468208。但是如果三个文件夹全部用定时线程读入，此时进程占用的物理内存为 240408。同样都是读入三份数据，且调用同一个读入函数，但是注释前后的进程占用的内存有明显的差别。

## C++ STL 容器如何解决线程安全的问题？

作者：果冻虾仁

众所周知，STL 容器不是线程安全的。

对于 `vector`，即使写方是单线程写入，但是并发读的时候，由于潜在的内存重新申请和对象复制问题，会导致读方的迭代器失效。实际表现也就是招致了 `core dump`。如果是多个写方，并发的 `push_back()`，也会导致 `core dump`。

解法一：加锁是一种解决方案，但是加 `std::mutex` 互斥锁确实性能较差。对于多读少写的场景可以用读写锁来缓解。`C++17` 引入了 `std::shared_mutex`。

解法二：更多的时候，其实可以通过固定 `vector` 的大小，避免动态扩容(无 `push_back`)来做到 lock-free！即在开始并发读写之前(比如初始化)的时候，给 `vector` 设置好大小。

```
struct Data {  
    ...  
};  
vector<Data> v;  
v.resize(1000);
```

注意是 `resize`，不是 `reserve`！可能大家平时用 `reserve()` 比较多，顾名思义，`reserve` 就是预留内存。为的是避免内存重新申请以及容器内对象的拷贝。说白了，`reserve()` 是给 `push_back()` 准备的！而 `resize` 除了预留内存以外，还会调用容器元素的构造函数，不仅分配了 N 个对象的内存，还会构造 N 个对象。从这个层面上来说，`resize()` 在时间效率上是比 `reserve()` 低的。但是在多线程的场景下，用 `resize` 再合适不过。所谓的「写操作」在这里不是插入新元素，而是修改旧元素。

如果 N 没办法预期就再把 `vector` 搞成 ring buffer(环形队列)来缓解压力。可以给元素类加上成员变量标记当前的读写状态、是否被消费等等。你可以把队列头的下标定义成原子变量(`std::atomic`)，尽管原子变量也需要做线程同步，但是比一般的锁开销要小很多啦。

STL 中还有一类关联容器其线程安全问题也不容小觑。比如 `map`、`unordered_map`。当有多个写线程对情况下，并发地插入 `map/unordered_map` 都会引发 `core dump`。如果全量的 key 有办法在并发之前就能拿到的，那么就对这个 `map`，提前做一下 `insert`。并发环境中如果只是修改 value，而不是插入新 key 就不会 `core dump`！不过如果你没办法保证多个写线程不会同时修改同一个 key 的 value，那么可能存在 value 的覆盖。

`unordered_map` 在单写多读的多线程场景下，会不会有问题呢？`gcc` 的 `unordered_map` 实现曾被爆出有这个问题。原因的新插入的元素，触发了 `rehash`，让其他线程在 `unordered_map` 中查找的过程之中出现了 `core dump`。

## STL 容器的线程安全性

基本上 STL 容器提供的线程安全性只有以下两点：

- ◆ 多个线程读取是安全的。
- ◆ 多个线程可以读同一个容器内的数据，读时不允许写操作。
- ◆ 多个线程对不同的容器写入是安全的。

## Can vector cause false sharing

I'm working with C++11 on a project and here is a function:

```
void task1(int* res) {  
    *res = 1;  
}  
  
void task2(int* res) {  
    *res = 2;  
}  
  
void func() {  
    std::vector<int> res(2, 0); // {0, 0}  
    std::thread t1(task1, &res[0]);  
    std::thread t2(task2, &res[1]);  
    t1.join();  
    t2.join();  
    return res[0] + res[1];  
}
```

The function is just like that. You see there is a std::vector, which store all of the results of the threads. My question is: can std::vector cause false sharing? If it can, is there any method to avoid false sharing while using std::vector to store the results of threads?  
eerorika:

1.can std::vector cause false sharing?

writing in one thread to an object that is in the same "cache line" as another object that is accessed in another thread causes false sharing.

Elements of an array are adjacent in memory and hence adjacent small elements of an array are very likely in the same cache line. Vector is an array based data structure. The pattern of accessing the elements of the vector in your example are a good example of false sharing.

2.is there any method to avoid false sharing while using std::vector to store the results of threads?

Don't write into adjacent small elements of an array (or a vector) from multiple threads. Ways to avoid it are:

Divide the array into contiguous segments and only access any individual segment from a separate thread. The size of the partition must be at least the size of the cache line on the target system.

Or, write into separate containers, and merge them after the threads have finished.

### **Are STL containers thread safe? If not, how can we achieve thread safety manually?**

John R. Grout:

The standard discusses which operations are thread safe and which require a mutex to maintain thread safety.

In a map-like container (either tree or hash based), any action that could trigger the creation of a new node requires a mutex, including operator [] (which will create a new node if one is not already present). The mutex has to lock out any would-be searchers, but searchers who have already received an iterator have thread-safe access to that node (through the iterator) even through an insert or a delete of another node. How to keep the value in the node thread safe is up to you.

## 返回值的内存管理

### 基本要点

执行某个函数时，如果有参数，则在栈上为参数分配空间(引用类型的参数例外)，进入到函数体内部，如果遇到变量则在栈上分配空间(static 类型变量和字符串常量例外)，执行完后，如果存在返回值，则先将返回值进行拷贝传回(**函数的返回值用于初始化在调用函数时创建的临时对象，如果返回类型不是引用，则将函数返回值复制给临时对象**)，再返回执行点，接着进行退栈操作，将刚才函数内部在栈上申请的内存空间释放掉。

```
void f()
{
    int a = 1; // a在栈区
    char s[] = "123"; // s在栈区, "123"在栈区, 其值可以被修改
    char *s = "123"; // s在栈区, "123"在常量区, 其值不能被修改
    int *p = (int *)malloc(sizeof(int)); // p在栈区, p指向的空间在堆区
    static int b = 0; // b在静态区
}

int test(void)
{
    int a = 1;
    return a;
}

int main(void)
{
    int b;
    b = test();
    cout << b << endl;
    return 0;
}
```

输出结果为 1。在 test 函数执行完后，存放 a 值的单元是可能会被重写，但是在函数执行 return 时，会创建一个 int 型的零时变量，将 a 的值复制拷贝给该零时变量，因此即使存放 a 值的单元被重写数据，但是不会受到影响。

```
char* test(void)
{
    char str[] = "hello world!";
    return str;
}

int main(void)
{
    char *p;
    p = test();
    cout << p << endl;
    return 0;
}
```

输出结果可能是 hello world!，也可能是乱麻。因为 str 以及"hello world" 是在栈上保存的，当用 return 将 str 的值返回时，将 str 的值拷贝一份传回，然后释放栈上的空间，即存放 hello world 的单元可能被重新写入数据，因此虽然 main 函数中的指针 p 是指向存放 hello world 的单元，但是无法保证 test 函数执行完后该存储单元里面存放的还是 hello world。

```
char* test(void)
{
    char *p = (char *)malloc(sizeof(char) * 100);
    strcpy(p, "hello world");
    return p;
}

int main(void)
{
    char *str;
    str = test();
    cout << str << endl;
    return 0;
}
```

运行结果 hello world。这种情况下同样可以输出正确的结果，是因为是用 malloc 在堆上申请的空间，这部分空间是由程序员自己管理的，如果程序员没有手动释放堆区的空间，那么存储单元里的内容是不会被重写的。

局部对象在离开函数作用域后会被自动析构(自动调用其析构函数)

- ◆ 如果返回局部变量的指针会造成了野指针，因为函数只是把指针复制后返回了，但是指针指向的内容已经被释放了。
- ◆ 局部变量数组是不能作为函数的返回值的。因为返回一个数组，实际上是返回指向这个数组首地址的指针。
- ◆ 函数返回指向存储在堆上的变量的指针是可以的。但是，程序员要自己负责在函数外释放分配在堆上的内存。
- ◆ 局部对象的引用是不能作为函数返回值的。函数返回后，局部变量的引用不再指向一个有效对象，结果无法预测。

### 引用返回

C++ 函数可以返回一个引用，用引用作函数的返回值的最大好处是在内存中不产生返回值的副本。引用作为返回值，须遵守以下规则：

- ◆ 不能返回局部变量的引用。主要原因是局部变量会在函数返回后被销毁。
- ◆ 不能返回函数内部 new 分配的内存的引用。如，函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间就无法释放。

### 返回引用的场景：

- ◆ 返回引用最常见的是返回类的成员的引用：

```
class A
{
public:
    std::string const& GetStr() const
    {
        return _s; // 返回的是引用
    }
private:
    std::string _s;
};
```

- ◆ 普通函数也可以返回 static 变量的引用。
- ◆ 比如两个参数 a,b，如果 a.B() 为空则返回 b，否则返回 \*a.B()，这种返回输入参数的引用很方便啊。再比如，写一个函数让字符串中的每个英文小写字母变成大写：

```
std::string& ToUpper(std::string& s)
{
    for (auto& c : s)
    {
        if (c >= 'a' && c <= 'z')
            c -= ('a' - 'A');
    }

    return s;
}
```

- ◆ 有时返回某个参数的引用可以方便链式调用，比如 cout<< a << b << c，就是靠不停返回 stream 的引用，即 builder.methodA().methodB().done()。

## 对象返回

当一个函数返回一个对象的时，要生成一个临时对象，写了下面一段代码进行测试：

```
struct Test{
    Test() {cout<<"Constructor"<<endl;}
    Test(const Test &){cout<<"copy Constructor"<<endl;}
    ~Test() {cout<<"destroy"<<endl;}
};

Test fun()
{
    Test t1;
    cout<<"&t1 is "<<&t1<<endl;
    return t1;
}

int main()
{
    Test t2 = fun();
    cout<<"&t2 is "<<&t2<<endl;
    cout<<"This is a test!"<<endl;
    return 0;
}
```

运行结果如下：

```
Administrator@ZA19030001 MINGW64 /d/wujianjun/code/my/zacpp/src/src
$ g++ *.cpp -fno-elide-constructors -o main -I/d/wujianjun/code/my/zacpp/src/include
Administrator@ZA19030001 MINGW64 /d/wujianjun/code/my/zacpp/src/src
$ ./main.exe
Constructor
&t1 is 0x61fdbf
copy Constructor
destroy
copy Constructor
destroy
&t2 is 0x61fe0e
This is a test!
destroy
```

可以看到，`return` 语句首先把 `t1` 赋值给临时对象，然后销毁 `t1`，接着把临时对象赋值给 `t2`，然后销毁临时对象。注意：`-fno-elide-constructors` 选项关闭了返回值优化的机制(**Return Value Optimization(RVO)**)，RVO 会略过临时对象赋值和销毁这一步，且直接拿到局部对象)。

```
Administrator@ZA19030001 MINGW64 /d/wujianjun/code/my/zacpp/src/src
$ g++ *.cpp -o main -I/d/wujianjun/code/my/zacpp/src/include
Administrator@ZA19030001 MINGW64 /d/wujianjun/code/my/zacpp/src/src
$ ./main.exe
Constructor
&t1 is 0x61fe0f
&t2 is 0x61fe0f
This is a test!
destroy
```

要将 `vector` 作为返回值的话，

```
vector<int> fun1(int num);
```

更好的处理方法：

```
bool fun1(int num, vector<int> &vec);
```

## 大对象返回拷贝问题

每次调用 `Get()` 会把 `temp` 里的元素都复制一遍，这样效率很低。不知道各位大大是怎么处理这种情况的？

```
vector<int> Get()
{
    vector<int> temp;
    //fill temp with some values
    return temp;
}

int main()
{
    vector<int> vec = Get();
}
```

作者：蓝色

**这不就是 C++11 右值引用和 Move 语意要解决的问题吗？**

C++11 中引进 `std::move()` 函数，使得我们可以“移动”对象内存所有权，免去多余复制操作。移动构造函数是 C++11 中新增加的一种构造函数，其作用是提高程序性能。

## 引用与指针

下面我们实现一个类，其中包含两个 map，一个用于读，一个用于写：

```
class Buffer
{
public:
    Buffer():cur_buffer_index(0){
        raw_profile_data_0 = unordered_map<string, string>();
        raw_profile_data_1 = unordered_map<string, string>();
    }

    unordered_map<string, string>& cur_read_buff(){
        if(cur_buffer_index == 0){ return raw_profile_data_0; }
        else{ return raw_profile_data_1; }
    }

    unordered_map<string, string>& cur_write_buff(){
        if(cur_buffer_index == 1){ return raw_profile_data_0; }
        else{ return raw_profile_data_1; }
    }

    void print0(){
        for(auto const& item : raw_profile_data_0){
            cout << item.first << "|" << item.second << endl;
        }
    }

    void print1(){
        for(auto const& item : raw_profile_data_1){
            cout << item.first << "|" << item.second << endl;
        }
    }

    int cur_buffer_index;
};

public:
    unordered_map<string, string> raw_profile_data_0;
    unordered_map<string, string> raw_profile_data_1;
};
```

接着我们操作两个 map：

```
2157     Buffer buffer = Buffer();
2158
2159     auto map_1 = buffer.cur_write_buff();
2160     map_1["A"] = "a";
2161     map_1["B"] = "b";
2162     map_1["C"] = "c";
2163     cout << "======"<< endl;
2164     buffer.print0();
2165     unordered_map<string, string> map_2 = buffer.cur_write_buff();
2166     map_2["A"] = "a";
2167     map_2["B"] = "b";
2168     map_2["C"] = "c";
2169     cout << "======"<< endl;
2170     buffer.print0();
2171     unordered_map<string, string>& map_3 = buffer.cur_write_buff();
2172     map_3["A"] = "a";
2173     map_3["B"] = "b";
2174     map_3["C"] = "c";
2175     cout << "======"<< endl;
2176     buffer.print0();
2177     buffer.cur_write_buff()["A"] = "a";
2178     buffer.cur_write_buff()["B"] = "b";
2179     buffer.cur_write_buff()["C"] = "c";
2180     cout << "======"<< endl;
2181     buffer.print0();
```

问题 3 输出 调试控制台 终端 JUPYTER

```
=====
=====
=====
C|c
A|a
B|b
=====
=====
C|c
A|a
B|b
```

可以看出：

- ◆ **auto 推断的类型不一定准确。**
- ◆ **函数返回值为引用时，必须用引用类型去接。**

# 现代内存与指针管理技术

## 智能指针与内存管理

智能指针使用了引用计数，每当增加一次对同一个对象的引用，那么引用计数就会加一，每删除一次引用，引用计数就会减一，当一个引用计数减为零时，就自动删除指向的堆内存。

### std::shared\_ptr

这是一种智能指针，它能够记录多少个 shared\_ptr 共同指向一个对象，当引用计数变为零的时候就会将对象自动 delete。**std::make\_shared 会自动 new(创建)传入参数中的对象，并返回这个对象类型的 std::shared\_ptr 指针**。std::shared\_ptr 可以通过 get() 来获取原始指针，通过 reset() 减少一个引用计数，并通过 use\_count() 来查看一个对象的引用计数。例如：

```
3882 #include <iostream>
3883 #include <memory>
3884 using namespace std;
3885
3886 class A
3887 {
3888     public:
3889         A(int _data):data(_data){ cout << "A is construct" << ",data=" << data << endl;}
3890         void print(){ cout << data << endl;}
3891         ~A(){ cout << "A is deconstruct" << ",data=" << data << endl;}
3892     public:
3893         int data;
3894     };
3895
3896 int main()
3897 {
3898     // 智能指针用法跟普通指针几乎是一样的，但是就是两种指针之间无法相互赋值
3899     cout << "======" << endl;
3900     A* pointer1 = new A(123); // 普通指针
3901     auto pointer2 = make_shared<A>(456); // 用法跟new高度类似
3902     pointer1->print();
3903     pointer2->print();
3904
3905     auto pointer1 = pointer2, pointerj = pointer2, pointerk = pointer2;
3906     cout << pointer2->use_count() << endl; // 4
3907     cout << "======" << endl;
3908     pointer1 = nullptr, pointerj = nullptr, pointerk = nullptr, pointer2 = nullptr; // 引用计数为0立即自动delete
3909     cout << pointer2->use_count() << endl; // 0
3910     cout << "======" << endl;
3911     // 手动delete
3912     delete pointer1;
3913     pointer1 = nullptr;
3914     return 0;
3915 }
```

输出结果：

```
A is construct,data=123
A is construct,data=456
123
456
4
=====
A is deconstruct,data=456
0
=====
A is deconstruct,data=123
```

直到 c++17 前 std::shared\_ptr 都有一个严重的限制，那就是它并不支持动态数组。

注意不能直接通过同一个 raw pointer 指针来构造多个 shared\_ptr：

```
int *p = new int{10};
shared_ptr<int> ptr1{ p };
shared_ptr<int> ptr2{ p };
```

此时 ptr1 和 ptr2 指向了同一块内存，但是却是不同的 shared\_ptr，他们的 use\_count 都是 1，当其中一个 use\_count=0 的时候，内存就会被释放，引起另外一个指针错误。再看个例子：每次调用 addToGroup() 都会创建一个新的 shared\_ptr，但是指向却是同一块内存，应该将 std::enable\_shared\_from\_this<Student> 作为 Student 的基类：

```
class Student
{
public:
    Student( const string &name ) : name_( name ) { }
    void addToGroup( vector<shared_ptr<Student>> &group ) {
        group.push_back( shared_ptr<Student>(this)); // ERROR
    }
private:
    string name_;
};

class Student : public std::enable_shared_from_this<Student>
{
public:
    Student( const string &name ) : name_( name ) { }
    void addToGroup( vector<shared_ptr<Student>> &group ) {
        group.push_back( shared_from_this());
    }
private:
    string name_;
};
```

另外不要保存 get() 的返回值，无论是保存为裸指针还是 shared\_ptr 都是错误的，保存为裸指针不知什么时候就会变成空悬指针，保存为 shared\_ptr 则产生了独立指针。不要 delete get() 的返回值，会导致对一块内存 delete 两次的错误。

没有指向引用的指针!!!

```
int x = 1;
int& y = x;
int *p1 = &y; // p1其实是指向x
int& *p2 = &y; // 错误
```

[Running]  
test.cpp: In function 'int main()':  
test.cpp:4322:8: error: cannot declare pointer to 'int&'  
int& \*p2 = &y; // 错误  
^~

shared\_ptr 包裹的是指定类型的指针，所以指定类型不能是引用，包括左值引用和右值引用。

### std::unique\_ptr

这是一种独占的智能指针，它禁止其他智能指针与其共享同一个对象，从而保证代码的安全：

```
std::unique_ptr<int> pointerx = std::make_unique<int>(10); // make_unique 从 C++14 引入
std::unique_ptr<int> pointery = pointerx; // 非法
```

但是，我们可以利用 std::move 将其转移给其他的 unique\_ptr，例如：

```
3916 std::unique_ptr<int> pointerx = std::make_unique<int>(10); // make_unique 从 C++14 引入
3917 std::unique_ptr<int> pointery = std::move(pointerx); // 非法
3918 // cout << *pointerx << endl; // 报错,此指针已经失效
3919 cout << "pointery" << endl;
3920
```

[Running]  
10

### std::weak\_ptr

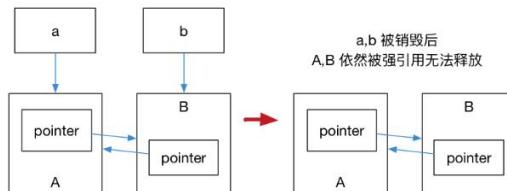
std::shared\_ptr 就会发现依然存在着资源无法释放的问题。看下面这个例子：

```
struct A;
struct B;

struct A {
    std::shared_ptr<B> pointer;
    ~A() {
        std::cout << "A 被销毁" << std::endl;
    }
};
struct B {
    std::shared_ptr<A> pointer;
    ~B() {
        std::cout << "B 被销毁" << std::endl;
    }
};

int main() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->pointer = b;
    b->pointer = a;
}
```

运行结果是 A, B 都不会被销毁，这是因为 a,b 内部的 pointer 同时又引用了 b,a，构成了循环引用，而离开作用域时，a,b 智能指针被析构，却只能造成这块区域的引用计数减一，引用计数不为零，造成了内存泄露，如下图：



为避免循环引用导致的内存泄露，就需要使用 weak\_ptr。

## std::move 与右值引用

### 概述与例子

先看段代码:

The screenshot shows a code editor with a dark theme. The code is as follows:

```
341 std::vector<std::string> fun(){
342
343     std::vector<std::string> vs;
344     vs.push_back("111"); vs.push_back("222"); vs.push_back("333");
345
346     return std::move(vs);
347 }
348
349 int main(int argc, char *argv[])
350 {
351     std::vector<std::string> vs1;
352     vs1.push_back("A"); vs1.push_back("B"); vs1.push_back("C");
353     std::cout<<"======"<<std::endl;
354     for(auto& item: vs1){
355         std::cout<<"vs1:<<item<<" ";
356     }
357     std::cout<<std::endl;
358
359     std::vector<std::string> vs2;
360     vs2 = std::move(vs1); // 把 vs1 管理的内存的管理权移交给 vs2
361     std::cout<<"======"<<std::endl;
362     for(auto& item: vs1){
363         std::cout<<"vs1:<<item<<" ";
364     }
365     std::cout<<std::endl;
366     for(auto& item: vs2){
367         std::cout<<"vs2:<<item<<" ";
368     }
369     std::cout<<std::endl;
370
371     std::cout<<"======"<<std::endl;
372     std::vector<std::string> vs3 = fun();
373     for(auto item: vs3){
374         std::cout<<"vs3:<<item<<" ";
375     }
376     std::cout<<std::endl;
377 }
378 }
```

问题 ② 编辑 调试控制台 终端

```
[Running] cd "d:\wujianjun\code\my\za\pp\src\src\" && g++ main.cpp -o main
=====
vs1:A vs1:B vs1:C
=====
vs2:A vs2:B vs2:C
=====
vs3:111 vs3:222 vs3:333
```

总结为:

- ◆ `T a = std::move(b)` 表示把对象 b 管理的内存的管理权移交给对象 a。
- ◆ `return std::move(a)` 会避免大型返回值的拷贝。

C++右值引用 std::move <https://zhuanlan.zhihu.com/p/94588204>

一个值要么是右值，要么是左值，左值是指表达式结束后依然存在的持久化对象，右值是指表达式结束时就不再存在的临时对象。所有的具名对象都是左值，而右值不具名。

左值可以取地址、位于等号左边；而右值没法取地址，位于等号右边。`int a = 5;` 这里 a 可以通过 `&` 取地址，位于等号左边，所以 a 是左值。5 位于等号右边，5 没法通过 `&` 取地址，所以 5 是个右值。“abc”,123 这些常量就是右值。一般的引用(左值引用)不能指向右值：

```
3924 int a = 5;
3925 int & ref1 = a;
3926 int & ref2 = 5;
3927 return 0;
问题 ③ 编辑 调试控制台 终端 JUPYTER
[Running]
test.cpp: In function 'int main()':
test.cpp:3926:15: error: cannot bind non-const lvalue reference of type 'int&' to an rvalue of type 'int'
    int & ref2 = 5;
```

但是，`const` 左值引用是可以指向右值的，如：`const int & ref2 = 5;`。

这也是为什么要使用 `const &` 作为函数参数的原因之一，如 `std::vector` 的 `push_back`:

```
void push_back (const value_type& val);
```

如果没有 `const`，`vec.push_back(5)` 这样的代码就无法编译通过了。

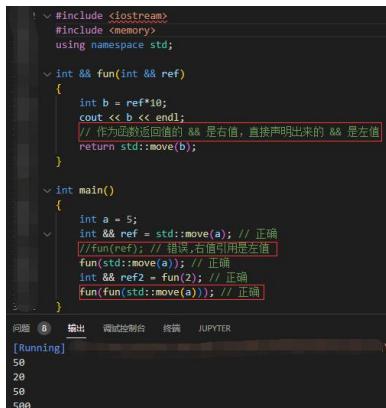
```
int a = 5;
int & ref1 = a;
int && ref2 = 5;
```

右值引用的标志是`&&`，可以指向右值，不能指向左值：

右值引用有办法指向左值吗？有办法，`std::move`：

```
int a = 5;
// int && ref3 = a; // 错误
int && ref3 = std::move(a); // 正确
```

`std::move` 唯一的功能是把左值强制转化为右值，其实现等同于一个类型转换：  
`static_cast<T&&>(value)`。被声明出来的左、右值引用都是左值。因为被声明出的左右值引用是有地址的，也位于等号左边。仔细看下边代码：



```
! ~ #include <iostream>
# include <memory>
using namespace std;

~ int && fun(int && ref)
{
    int b = ref*10;
    cout << b << endl;
    // 作为函数返回值的 && 是右值，直接声明出来的 && 是左值
    return std::move(b);
}

~ int main()
{
    int a = 5;
    int && ref = std::move(a); // 正确
    // fun(ref); // 错误：右值引用是左值
    fun(std::move(a)); // 正确
    int && ref2 = fun(2); // 正确
    fun(fun(std::move(a))); // 正确
}
```

其实右值引用既可以是左值也可以是右值，如果有名称则为左值，否则是右值。或者说：  
作为函数返回值的 `&&` 是右值，直接声明出来的 `&&` 是左值!!!!!!

下面的代码显示了移动构造函数&移动赋值函数的写法和用法：

```
class MemoryBlock
{
public:
    MemoryBlock(size_t length): _length(length), _data(new int[_length]) // 构造函数
    {
        cout << "normal constructor " << endl;
        for(size_t i=0 ;i < _length; i++){ _data[i] = length; } // 每个元素被赋值为length
    }
    ~MemoryBlock(){ // 析构函数
        cout << "de constructor " << endl;
        if(_data != nullptr){
            delete[] _data;
        }
    }
    MemoryBlock(const MemoryBlock& other): _length(other._length), _data(new int[_length]) { // 拷贝构造函数
        cout << "copy constructor " << endl;
        std::copy(other._data, other._data + _length, _data);
    }
    MemoryBlock& operator=(const MemoryBlock& other){ // 赋值运算符
        cout << "normal assignment " << endl;
        if (this != &other){
            delete[] _data; // 注意，赋值运算后一定要释放掉本对象的内存
            _length = other._length;
            _data = new int[_length];
            std::copy(other._data, other._data + _length, _data);
        }
        return *this;
    }
    MemoryBlock(MemoryBlock&& other): _data(nullptr), _length(0) { // 移动构造函数,参数是右值引用!!!!!
        cout << "move constructor " << endl;
        _data = other._data; // 将本对象指针指向对方
        _length = other._length;
        other._data = nullptr; // 置空对方状态
        other._length = 0;
    }
    MemoryBlock& operator=(MemoryBlock&& other){ // 移动赋值运算符,参数是右值引用!!!!!
        cout << "move operator " << endl;
        if (this != &other){
            delete[] _data;
            _data = other._data; // 将本对象指针指向对方
            _length = other._length;
            other._data = nullptr; // 置空对方状态
            other._length = 0;
        }
        return *this;
    }
    void print() // 打印每个元素
    {
        for(size_t i =0 ;i < _length; i++){ cout << _data[i] << ","; }
        cout << endl;
    }
private:
    size_t _length; // The length of the resource.
    int* _data; // The resource.
};
```

下面是运行结果：

```
int main() {
    MemoryBlock mb1(1); // normal constructor
    MemoryBlock mb2(mb1); // 调用拷贝构造函数
    MemoryBlock mb3(3); //
    mb3 = mb1; // 调用赋值函数
    MemoryBlock mb4(move(mb3)); // 调用移动构造函数
    MemoryBlock mb5(5); //
    mb5 = move(mb4); // 调用移动赋值函数
    mb5.print();
    mb5 = static_cast<MemoryBlock &&>(mb4); // 调用移动赋值函数, 注意mb4此时已经为空
    mb5.print(); // mb5是空的!!!!!!
    cout << "-----" << endl;
}
```

[Running]

```
normal constructor
copy constructor
normal constructor
normal assignment
move constructor
normal constructor
move operator
1,
move operator
```

g++ test.cpp -o test

再看一份运行结果：

```
void fun1(MemoryBlock mb) { cout << "fun1" << endl;}
void fun2(MemoryBlock&& mb) { cout << "fun2" << endl;}
```

```
int main() {
    MemoryBlock mb(2); // normal constructor
    fun1(mb); // 先用mb构造一个临时对象, 函数返回后析构这个临时对象
    fun2(move(mb)); // 不会新建对象
    cout << "-----" << endl;
    MemoryBlock mb1(2); // normal constructor
    MemoryBlock mb2(3); // normal constructor
    // 右值引用可以赋值给左值(需要调用移动构造函数), 也可以赋值给右值(不会调用移动构造函数)
    MemoryBlock mbx = std::move(mb1); // move constructor
    MemoryBlock&& mby = std::move(mb2); // 这一步根本没生成新对象!!!!!
}
```

问題 2 執出 調試控制台 終端 JUPYTER

```
normal constructor
copy constructor
fun1
de constructor
fun2
-----
normal constructor
normal constructor
move constructor
-----
de constructor
de constructor
de constructor
de constructor
```

g++ test.cpp -o test &&

我们可以得出一下重要结论：

- ◆ 当遇到右值引用构造或赋值时，则会调用对应的移动函数构造函数或移动赋值函数。
- ◆ 一个对象第一次被 move 后，其以及为空了，虽然还没被析构。!!!!!!
- ◆ 右值引用可以赋值给左值，此时需要调用移动构造函数，会生成新对象；右值引用也可以赋值给右值，此时不会调用移动构造函数，不会生成新对象。!!!!!!

STL 的很多容器都实现了移动构造函数和移动赋值重载函数。**参数为左值引用意味着拷贝，为右值引用意味着移动。**vector::push\_back 使用 std::move 可以提高性能：

```
std::string str1 = "aacasxs";
std::vector<std::string> vec;
// 传统方法, copy
vec.push_back(str1);
// 调用移动语义的push_back方法, 避免拷贝, str1 会失去原有值, 变成空字符串
vec.push_back(std::move(str1));
```

在<需要拷贝且被拷贝者之后不再被需要>的场景，建议使用 std::move，提升性能。

## C++ 函数返回局部变量的 std::move() 问题？

```
User create_user(const std::string &username, const std::string &password) {
    User user(username, password);
    validate_and_save_to_db(user);
    return user;
}
void signup(const std::string &username, const std::string &password) {
    auto new_user = create_user(username, password);
    login(user);
}
```

一些 C++ 初学者可能会觉得这个代码不够优化，因为 `create_user` 创建先创建了一个 `user`，然后返回时又把 `user` 赋值给 `new_user`，这会 copy `user` 里面的内容，如果 `user` 很大的话，这样太慢。会想用 `move` 来优化，因为 `user` 在 `return` 了之后就没用了，我可以把 `user move` 到 `new_user`，这样不就省掉了 copy 时很大的开销了么。但事实上，在这种简单的情况下编译器比你更聪明，编译器可以直接把 `user` 创建在 `new_user` 里，`user` 只被创建一次，没有任何 copy 开销，`user` 和 `new_user` 经过编译器优化之后其实是同一个 variable！这种优化叫做 copy elision。但是如果用户想自己用 `move` 优化的话，编译器就不用做 copy elision 了，它先创建一个 `user`，然后在调用 `User` 的 `move constructor` 来创建 `new_user`。这样开销更大。

有没有可能我写的函数逻辑特别复杂，编译器没法优化呢？此时我如果写 `return move(a)` 不就会比 copy 更快了吗？这个逻辑是正确的，那到底什么时候应该 `move`，什么时候应该依靠 copy elision 呢？通常主流的编译器都会 100% copy elision 以下两种情况：

1. URVO(Unnamed Return Value Optimization): 函数的所有执行路径都返回同一个类型的匿名变量，比如：

```
User create_user(const std::string &username, const std::string &password) {
    if (find(username)) return get_user(username);
    else if (validate(username) == false) return create_invalid_user();
    else User{username, password};
}
```

这里所有的 `return` 都返回一个 `User` 类型，且每个返回的都是一个匿名变量。那编译器 100% 会执行 copy elision。

2. NRVO(Named Return Value Optimization): 函数的所有路径都返回同一个非匿名变量，比如

```
User create_user(const std::string &username, const std::string &password) {
    User user{username, password};
    if (find(username)) {
        user = get_user(username);
        return user;
    } else if (user.is_valid() == false) {
        user = create_invalid_user();
        return user;
    } else {
        return user;
    }
}
```

这里因为所有路径都返回同一个变量 `user`。

其他的情况编译器可能都不会使用 copy elision 的优化。

作者：暮无井见铃

补充两点：

- ◆ URVO 在 C++17 是强制的。不过 NRVO 不是强制，意味着有时不这么优化也是允许的。
- ◆ 若 `return` 的表达式是符合返回类型的左值，且编译器没有进行复制省略，那么标准（C++11 开始）也要求编译器先试图把表达式当右值，优先匹配移动构造函数（再匹配通常的复制构造函数），若失败的话则再将其当左值，匹配接受非 `const` 引用的复制构造函数。所以按照标准，上面的 `std::move(a)` 是不必要的（除非你希望强制调用移动构造函数），编译器在必要时会做同样的处理。

## 如何评价 C++11 的右值引用(Rvalue reference)特性?

作者: Tinro

类的右值是一个临时对象, 如果没有被绑定到引用, 在表达式结束时就会被废弃。我们用右值来表示对象的资源可以移动。对于左值, 如果我们明确放弃对其资源的所有权, 则可以通过 `std::move()` 来将其转为右值引用。`std::move()` 实际上是 `static_cast<T&&>()` 的封装。

右值引用至少可以解决以下场景中的移动语义缺失问题:

- ◆ 按值传入参数: 如果传入参数是为了将资源交给函数接受者, 就应该按值传参:

```
class People {
public:
    People(string name) // 按值传入字符串, 可接收左值、右值。接收左值时为复制, 接收右值时为移动
    : name_(move(name)) // 显式移动构造, 将传入的字符串移入成员变量
    {
    }
    string name_;
};

People a("Alice"); // 移动构造name
```

构造 `a` 时, 调用了一次字符串的构造函数和一次字符串的移动构造函数。如果使用 `const string& name` 接收参数会有一次构造和一次拷贝构造, 以及一次 `non-trivial` 的析构。

- ◆ 按值返回: 曾经为了返回容器而这样写: `void str_split(const string& s, vector<string>* vec);` 这样要求 `vec` 在外部被事先构造。有了移动语义, 就可以写成这样:

```
vector<string> str_split(const string& s) {
    vector<string> v;
    // ...
    return v; // v是左值, 但优先移动, 不支持移动时仍可复制
}
```

如果函数按值返回, 标准要求优先调用移动构造函数, 如果不符再调用拷贝构造函数。

下面展示了注释移动构造函数前后的不同

```
class MOVEC
{
public:
    MOVEC() { cout << "normal constructor " << endl; } // 构造函数
    ~MOVEC() { cout << "de constructor " << endl; } // 析构函数
    MOVEC(const MOVEC& other) { cout << "copy constructor " << endl; } // 拷贝构造函数
    MOVEC(MOVEC&& other) { cout << "move constructor " << endl; } // 移动构造函数, 参数是右值引用
};

MOVEC fun()
{
    MOVEC a();
    MOVEC b();
    return a;
}

int main()
{
    MOVEC c = fun();
}

(base) g++ t.cpp -o t -fno-elide-constructors
(base) ./t
normal constructor
move constructor
de constructor
copy constructor
de constructor
copy constructor
de constructor
de constructor
```

- ◆ 容器管理: 参见 `std::vector` 的 `push_back` 函数。

```
void push_back( const T& value ); // (1)
void push_back( T&& value ); // (2)
```

如果你要往容器内放入超大对象, 那么版本 2 自然是最好选择。

```
vector<vector<string>> vv;

vector<string> v = {"123", "456"};
v.push_back("789"); // 临时构造的string类型右值被移动进容器
vv.push_back(move(v)); // 显式将v移动进vv
```

当 `vector` 的存储容量需要增长时, 通常会申请新内存, 复制内容, 删除对象。改用移动就高效多了了。曾经由于 `vector` 增长时会复制对象, 像 `std::unique_ptr` 这样不可复制的对象是无法放入容器的。所以随着移动语义的引入, `std::unique_ptr` 放入 `std::vector` 成为理所当然的事情。使用 `vector<unique_ptr<T>>`, 完全无需显式析构, `unique_ptr` 自会打理一切。如果需要共享所有权, 那么基于引用计数的 `shared_ptr` 是一个好的选择。同样 `std::thread`, `std::future` `std::promise` `std::packaged_task` 等等这一票多线程类都是不可复制的, 也都可以用移动的方式传递。

## 引用折叠和完美转发

<https://zhuanlan.zhihu.com/p/50816420>

下面的代码是错误的：

```
int a = 0;
int &ra = a;
int & &rra = ra; // 编译器报错：不允许使用引用的引用!
```

先看看万能引用，它并不是 C++ 的语法特性，而是我们利用现有的 C++ 语法，自己实现的一个功能。这个功能既能接受左值类型的参数，也能接受右值类型的参数。

```
template<typename T>
ReturnType Function(T&& param) // 万能引用的形式
{ ... }
```

看个例子：

The screenshot shows a code editor with several highlighted sections in red boxes:

- A red box highlights the line `int & &rra = ra;` with the note: "编译器报错：不允许使用引用的引用!" (Compiler error: it's not allowed to use a reference to a reference!).
- A red box highlights the code `template<typename T> void print(T && t){ std::cout << "右值" << std::endl; }` with the note: "可以接受 T& 也可以接受T&&,但是就是不可以接受T" (Can accept T& or T&&, but not T).
- A red box highlights the code `template<typename T> void fun1(T && v){ std::cout << "=====fun1======" << std::endl; print(v); print(std::forward<T>(v)); print(std::move(v)); }` with the note: "这种参数必须是引用类型,可以是左值引用,也可以是右值引用,但必须是引用" (This parameter must be a reference type, can be l-value reference or r-value reference, but must be a reference).
- A red box highlights the code `template<typename T> void fun2(T v){ // 虽然参数都是右值引用 std::cout << "=====fun2======" << std::endl; }` with the note: "可以接受 T& 也可以接受T&&,可以接受T" (Can accept T& or T&&, can accept T).
- A red box highlights the line `fun1<int&>(x);` with the note: "明明是int,怎么就变成了int &&呢???" (It's int, why did it become int &&?).
- A red box highlights the line `fun1<int&&>(move(x));` with the note: "注意T&的形参可以接受T的实参的" (Note that T& parameter can accept T's actual parameter).
- A red box highlights the line `fun2<int>(x);`.
- A red box highlights the line `fun2<int&&>(move(x));`.
- A red box highlights the line `// fun1<int>(x); // 报错!!!!!!"!!!` with the note: "fun1<int>(x); // 报错!!!!!!"!!!

The output window shows the results of the program execution:

```
=====fun1=====
左值
左值
右值
=====fun1=====
左值
右值
右值
=====fun2=====
=====fun2=====
=====fun2=====
```

**int & && 是什么呢？它是引用折叠，int& && 等价于 int &，int&& && 等价于 int &&。std::move 的作用是无论你传给它的是左值还是右值，通过 std::move 之后都变成了右值。std::forward 的作用是保持原来的值属性不变。如果原来的值是左值，经 std::forward 处理后该值还是左值；如果原来的值是右值，经 std::forward 处理后它还是右值。看下 forward 代码：**

```
template <typename T>
T&& forward(typename std::remove_reference<T>::type& param)
{
    return static_cast<T&&>(param);
}

template <typename T>
T&& forward(typename std::remove_reference<T>::type&& param)
{
    return static_cast<T&&>(param);
}
```

forward 实现了两个模板函数，一个接收左值，另一个接收右值。在上面有代码中：  
`typename std::remove_reference<T>::type` 的含义是获得去掉引用的参数类型。

## new /delete 运算符

下面是使用 new 运算符来为任意的数据类型动态分配内存的通用语法:

```
new data-type;
```

**data-type** 可以是包括数组在内的任意内置的数据类型，也可以是包括类或结构在内的用户自定义的任何数据类型。

```
class Box
{
public:
    Box() { cout << "constructing" << endl; }
    ~Box() { cout << "destroying" << endl; }
};

int main(){
    // new/delete 基本类型
    double* pvalue = new double; // 为变量请求内存
    cout << *pvalue << endl; // 随机值
    *pvalue = 100.0; // 在分配的地址存储值
    cout << *pvalue << endl;
    delete pvalue;

    // new/delete 数组类型
    int *array=new int [5];
    for (int i = 0; i < 5;i++){
        array[i] = i;
    }
    delete[] array;

    // new/delete 对象
    Box* myBox = new Box();
    delete myBox;

    // new/delete 对象数组
    Box* myBoxArray = new Box[4];
    delete [] myBoxArray;
}
```

**new** 用于动态分配内存、并用构造函数初始化分配的内存。注意，**new** 会调用对象的构造函数，**malloc** 则不会。每个 **new** 获取的对象，必须用 **delete** 析构并释放内存，以免内存泄漏。

**new** 内部实现分为两步：

- ◆ 调用相应的 **operator new()** 函数，动态分配内存。
- ◆ 调用 **placement new**，在分配到的内存块上初始化相应类型的对象并返回其首地址。

**new** 运算符表达式是 C++ 的一种语言结构，不可重载。但用户可重载 **operator new()** 函数。

默认的 **operator new** 实现如下：

```
// 全局 operator new
void * operator new(std::size_t size) throw(std::bad_alloc) {
    if (size == 0)
        size = 1;
    void* p;
    while ((p = ::malloc(size)) == 0) { // 采用 malloc 分配空间
        std::new_handler nh = std::get_new_handler();
        if (nh)
            nh();
        else
            throw std::bad_alloc();
    }
    return p;
}
// 对应的全局 operator delete 采用 free 释放空间
void operator delete(void* ptr) {
    if (ptr)
        ::free(ptr); // 采用 free 释放空间。
}
```

可以看到全局 **operator new** 分配空间，简单的调用了 **malloc()** 函数来分配空间。并没有

做任何初始化工作。现在问题来了：已经有了一段分配好的空间，如何在这个空间上调用这个类的构造函数，从而真正的创建一个对象呢？解决方案是：placement new。

```
class A {...} // 声明一个类 A  
void *buf = malloc(sizeof(A)); // 简单地分配空间。  
A *obj = new (buf)A(); // 在分配的空间上调用构造函数。
```

placement new 的语法是：

```
new (expression-list) new-type-id (optional-initializer-expression-list);
```

C++98 标准规定，new 创建的对象数组不能被显式初始化，数组所有元素被缺省初始化。

如果数组基类型没有缺省初始化，则编译报错。但 C++11 已经允许显式初始化，例如：

```
int *p_int = new int[3] {1, 2, 3};
```

## malloc 类函数行为研究

`void* malloc (size_t size);`

Allocates a block of size bytes of memory, returning a pointer to the beginning of the block. **The content of the newly allocated block of memory is not initialized, remaining with indeterminate values.**

```
51     int *p = (int*)malloc(sizeof(int)*5);  
52     for (int i = 0; i < 5;i++){  
53         std::cout << p[i] << std::endl;  
54     }  
55 }
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2

```
16784064  
0  
16777552  
0  
0
```

但是在 linux 上被初始化了：

```
jjwu@priv02:~/cholmodtest$ cat t.cpp  
#include<iostream>  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <vector>  
int main(){  
    int *p = (int*)malloc(sizeof(int)*5);  
    for (int i = 0; i < 5;i++){  
        std::cout << p[i] << std::endl;  
    }  
    return 0;  
}  
jjwu@priv02:~/cholmodtest$  
jjwu@priv02:~/cholmodtest$  
jjwu@priv02:~/cholmodtest$  
jjwu@priv02:~/cholmodtest$ ./t  
0  
0  
0  
0  
0
```

但是官方文档中仍然明说没有被初始化。

The `malloc()` function allocates `size` bytes and returns a pointer to the allocated memory. **The memory is not initialized.** If `size`

## c++内存管理之性能优化

<https://zhuanlan.zhihu.com/p/344377490>

### 内存对齐

尽管内存是以字节为单位，但是大部分处理器并不是按字节块来存取内存的。它一般会以 2 字节，4 字节，8 字节，16 字节甚至 32 字节为单位来存取内存，我们将上述这些存取单位称为**内存存取粒度**。现在考虑 4 字节存取粒度的处理器取 int 类型变量，该处理器只能从地址为 4 的倍数的内存开始读取数据。假如没有内存对齐机制，数据可以任意存放，现在一个 int 变量存放在从地址 1 开始的联系四个字节地址中，该处理器去取数据时，要先从 0 地址开始读取第一个 4 字节块，剔除不要的数据(0 地址)，然后从地址 4 开始读取下一个 4 字节块，同样剔除不要的数据(5, 6, 7 地址)，最后留下的两块数据合并放入寄存器。现在有了内存对齐的，int 类型数据只能存放在按照对齐规则的内存中，比如说 0 地址开始的内存。那么现在该处理器在取数据时一次性就能将数据读出来了，提高了效率。

C++空类的内存大小为 1 字节，而非空类的大小与类中非静态成员变量和虚函数表的多少有关。`struct/class` 内存对齐原则：

- ◆ 数据成员对齐规则：**第一个数据成员放内存 offset(32 位机器为 32bit, 64 位机器为 64bit) 为 0 地方，以后每个基本类型的数据成员的起始位置要从其大小与#`pragma pack(n)` 中 n 两者的最小值的整数倍开始。**如果成员是节后提，则要从其内部“最宽基本类型成员”的整数倍地址开始存储(如 struct a 里存有 struct b,b 里有 char,int,double 等元素，那 b 应该从 8 的整数倍开始存储。)。
- ◆ 在成员变量内存对齐之后，`struct/class` 本身也要进行内存对齐，对齐按照#`pragma pack(n)` 指定的数值和类中最大成员变量类型所占字节数中，比较小的那个进行，不足的要补齐。`sizeof` 的结果。

#`pragma pack(n)`作为一个预编译指令用来设置内存对齐的字节数。需要注意的是，n 的缺省数值是编译器设置的，一般为 8，合法的数值分别是 1、2、4、8、16。

看下例子：

```
64 //#pragma pack(1)
65
66 struct Test1
67 {
68     char a;
69     int b;
70     short c;
71 };
72
73 class Test2
74 {
75     int b;
76     short c;
77     char a;
78 };
79
80 int main(){
81     cout << alignof(Test1) << endl;
82     cout << sizeof(Test1) << endl;
83     cout << alignof(Test2) << endl;
84     cout << sizeof(Test2) << endl;
85 }
```

```
64 #pragma pack(1)
65
66 struct Test1
67 {
68     char a;
69     int b;
70     short c;
71 };
72
73 class Test2
74 {
75     int b;
76     short c;
77     char a;
78 };
79
80 int main(){
81     cout << alignof(Test1) << endl;
82     cout << sizeof(Test1) << endl;
83     cout << alignof(Test2) << endl;
84     cout << sizeof(Test2) << endl;
85 }
```

OUTPUT	TERMINAL	DEBUG CONSOLE	PROBLEMS
4 12 4 8	xe' } 1 7 1 7		

## 内存分配原理

`malloc` 其采用内存池的方式，先申请大块内存作为堆区，然后将堆区分为多个内存块。

- ◆ 当有申请请求时，`malloc` 会扫描空闲链表，直到找到一个足够大的块为止(首次适应)。因此每次调用 `malloc` 时并不是花费了完全相同的时间。如果该块恰好与请求的大小相符，则将其从链表中移走并返回给用户。如果该块太大，则将其分为两部分，尾部的部分分给用户，剩下的部分留在空闲链表中。**因此 `malloc` 分配的是一块连续的内存。**
- ◆ 释放时，首先搜索空闲链表，找到可以插入被释放块的合适位置。如果与被释放块相邻的任一边是一个空闲块，则将这两个块合为一个更大的块，以减少内存碎片。

`malloc` 会预先向操作系统申请一块内存供用户使用，当我们申请和释放内存的时候，`malloc` 会将这些内存管理起来，并通过一些策略来判断是否将其回收给操作系统。

几个内存分配函数的区别：

- ◆ `void* malloc(unsigned size)`: 在堆内存中分配一块长度为 `size` 字节的连续区域，参数 `size` 为需要内存空间的长度。**函数 `malloc` 不能初始化所分配的内存空间。通过 `malloc` 函数得到的堆内存必须使用 `memset` 函数来初始化。**
- ◆ `void* calloc(size_t numElements, size_t sizeOfElement)`: `sizeOfElement` 为单位元素长度，`numElements` 为元素个数，即在内存中申请 `numElements * sizeOfElement` 字节大小的连续内存空间。**该函数与 `malloc` 函数的一个显著不同时是，`calloc()` 会将所分配的内存空间中的每一位都初始化为零。**
- ◆ `void* realloc(void* ptr, unsigned newsize)`: 使用 `realloc` 函数为 `ptr` 重新分配大小为 `size` 的一块内存空间。**realloc 则是对给定的指针所指向的内存空间进行扩大或者缩小。** 下面是这个函数的工作流程：
  - 如果 `ptr` 为 `NULL`，则函数相当于 `malloc(new_size)`。
  - 如果 `ptr` 不为 `NULL`，先查看 `ptr` 是不是在堆中，如果不在则抛 `invalid pointer` 异常。  
**如果 `ptr` 在堆中，则查看 `new_size` 大小，如果 `new_size` 大小为 0，则相当于 `free(ptr)`，将 `ptr` 指向的内存空间释放掉，返回 `NULL`。如果 `new_size` 小于原大小，则 `ptr` 中的数据可能会丢失，只有 `new_size` 大小的数据会保存；如果 `size` 等于原大小，等于什么都没有做；如果 `size` 大于原大小，则查看 `ptr` 指向的位置还有没有足够的连续内存空间，如果有的话，分配更多的空间，返回的地址和 `ptr` 相同，如果没有的话，在更大的空间内查找，如果找到 `size` 大小的空间，将旧的内容拷贝到新的内存中，把旧的内存释放掉，则返回新地址，否则返回 `NULL`。**

`malloc` 或 `realloc` 返回的内存块地址都是 8 的倍数，

- ◆ `void *aligned_alloc (size_t size, size_t alignment)`: Allocate a block of `size` bytes, **starting on an address that is a multiple of alignment**. The alignment must be a power of two and `size` must be a multiple of alignment. As one example, SSE2 (SIMD) instructions need their data aligned on 16-byte boundaries.

从堆上获得的内存空间在程序结束以后，系统不会将其自动释放，需要程序员来自己管理。一个程序结束时，必须保证所有从堆上获得的内存空间已被安全释放，否则，会导致内存泄露。`void free (void * p)` 用来实现 `malloc` 分配的内存。**但是，`free` 函数只是释放指针指向的内容，而该指针仍然指向原来指向的地方，此时，指针为野指针。** 安全做法是：在使用 `free` 函数释放指针指向的空间之后，将指针的值置为 `NULL`。

注意，`memset` 是有执行代价的。我们来看实测数据：

内存大小	重复次数	时间
1K	100000	14ms
1M	1000	60ms
10M	1000	725ms

所以如果你在代码里循环调用 `memset`, 或者每次流程处理都要调用 `memset`, 而且每次都要 `memset` 1M 以上, 那么你就要小心了, 1000 次的 `memset` 1M 就能够消耗你 100 多 ms 的时间, 对于高并发高性能的系统来说, 这个时间是非常可观的。**memset 1k 以下, 次数也不多, 则可以用; 否则除非你确定一定要 memset, 否则就不要用, 你可以用其它替代方法。另外数组初始化时, 可能会隐式地调用 memset:**

```
char buffer[1024 * 1024] = {0}; // 这时可以调用memset(0xbfe9afec, '/000', 1048576)
```

在 gcc 编译时加上 O2 或者 O3 优化参数, 就不会隐含调用 `memset` 了。

## C++新特征

### C++新特性

<https://zhuanlan.zhihu.com/p/139515439>

#### auto & decltype

<https://zhuanlan.zhihu.com/p/137662774>

auto 让编译器在编译时推导出变量的类型，如：

```
auto f=3.14;      //double
auto s("hello"); //const char*
auto z = new auto(9); // int*
auto x1 = 5, x2 = 5.0, x3='r'; //错误，必须是初始化为同一类型
```

但是，这么简单的变量声明类型不建议用 auto，**它更适用于类型冗长复杂的场景**：

```
4829 // void fun(auto& data) {} // 错误!!! auto不能用作函数参数
4830
4831 int main()
4832 {
4833     std::map<std::string, int> m{{"CPU", 10}, {"GPU", 15}, {"RAM", 20}};
4834     for(std::map<std::string, int>::iterator iter = m.begin(); iter != m.end(); iter++)
4835     {
4836         cout << iter->first << "-" << iter->second << endl;
4837     }
4838     cout << "======" << endl;
4839     for(auto& iter : m)
4840     {
4841         cout << iter.first << "-" << iter.second << endl;
4842     }
4843
4844     int i = 10;
4845     auto a = i, &b = i, *c = &i; // a是int, b是i的引用, c是i的指针, auto就相当于int
4846     // auto e; // 错误!!! 使用auto必须马上初始化, 否则无法推导类型
4847     // auto d = 0, f = 1.0; // 错误!!! auto在一行定义多个变量时, 各个变量的类型必须一样, 这里d是int, f是float
4848
4849     int x[3] = {1,2,3};
4850     // auto y[3] = {1,2,3}; // 错误!!! auto不能定义数组
4851 }
```

问题 ③ 输出 调试控制台 终端 JUPYTER

CPU->10  
GPU->15  
RAM->20  
=====  
CPU->10  
GPU->15  
RAM->20

◆ auto 的使用必须马上初始化，否则无法推导出类型。

auto 在一行定义多个变量时，各个变量的类型应该一样。

◆ auto 不能用作函数参数。auto 不能定义数组，可以定义指针。

◆ **在类中 auto 不能用作非静态成员变量，auto 无法推导出模板参数**

decltype 则用于在编译时推导表达式类型(表达式实际不会进行运算)：

```
class A{
public:
    A(){}
    A(const A& obj){}
    A& operator=(const A& obj){return *this;}
    void fun(){ cout << "fun" << endl;}
};

class B{
public:
    B(){}
    B(const B& obj){}
    B& operator=(const B& obj){ return *this;}
    int operator[](int i) { return 1; }
    A operator[](const string i){ return A(); }
};

int main()
{
    int a = 1;
    double b = 4.1;
    decltype(a+b) c; // 
    c = 0.001;
    // c = string("1"); // 错误, 因为 decltype(a+b) 等于 double, 所以 c 是double, 不能被转换为string
    B obj_a;
    decltype(obj_a[1]) i; //
    // i.fun(); // 错误!!! decltype(obj_a[1]) 等于 int
    decltype(obj_a["1"]) j; // 正确 decltype(obj_a[1]) 等于 A
    j.fun();
}
```

问题 ③ 输出 调试控制台 终端 JUPYTER

fun

对于 decltype(exp)有

- ◆ exp 是表达式， decltype(exp) 和 exp 类型相同。
- ◆ exp 是函数调用， decltype(exp) 和函数返回值类型相同。

decltype 与 auto 一起主要对编写模板库的开发人员有用。

```
struct A{};
struct B{};

class T1{
public:
    T1(){}
    T1(const T1& obj){}
    B operator+(const B& obj) { cout<<"T1+B"<<endl; return B(); }
    A operator+(const A& obj) { cout<<"T1+A"<<endl; return A(); }
};

class T2{
public:
    T2(){}
    T2(const T2& obj){}
    A operator+(const B& obj) { cout<<"T2+B"<<endl; return A(); }
    B operator+(const A& obj) { cout<<"T2+A"<<endl; return B(); }
};

template<typename T, typename U>
auto add1(T obj1, U obj2) -> decltype(obj1 + obj2)
{ return obj1 + obj2; }

// 返回值类型是未知输入类型的对象经过复杂运算后的类型,此时decltype很有用!!!!
template<typename T, typename U>
decltype(auto) add2(T obj1, U obj2) // 也可以这样写,显然这样写更好!!!!!!
{
    A a; T1 t1;
    auto c = add1(t1, a);
    return c;
}

int main()
{
    A a; B b;
    T1 t1; T2 t2;
    auto c1 = add1(t1, a);
    auto c2 = add1(t2, a);
    auto c3 = add2(t1, b);
    auto c4 = add2(t2, b);
}
```

问题 3 输出 调试控制台 终端 JUPYTER

T1+A  
T2+A  
T2+A  
T1+A

## final & override

<https://zhuanlan.zhihu.com/p/258383836>

对比下面两个例子

```
class Base {
public:
    virtual void doSomething(int i) const
    { cout << "This is from Base with " << i << endl; }
};

class Derived : public Base {
public:
    virtual void doSomething(int i) const
    { cout << "This is from Derived with " << i << endl; }
};

void letDoSomething(Base& base)
{
    base.doSomething(419);
}

int main()
{
    Derived d;
    letDoSomething(d);
}
```

问题 3 输出 调试控制台 终端 JUPYTER

[Running]  
This is from Derived with 419

```
class Base {
public:
    virtual void doSomething(int i) const
    { cout << "This is from Base with " << i << endl; }
};

class Derived : public Base {
public:
    virtual void doSomething(int i)
    { cout << "This is from Derived with " << i << endl; }
};

void letDoSomething(Base& base)
{
    base.doSomething(419);
}

int main()
{
    Derived d;
    letDoSomething(d);
}
```

问题 3 输出 调试控制台 终端 JUPYTER

[Running]  
This is from Base with 419

就是你本来想重写基类的某个虚函数，但是你却稍微搞错了基类中该函数的原型(这里仅仅忘记写 const 了)，结果完全不同了。如果此时编译器能告诉我们搞错了就好了。

看下面的例子：

```

class Base {
public:
    virtual void doSomething(int i) const
    { cout << "This is from Base with " << i << endl; }
};

class Derived : public Base {
public:
    virtual void doSomething(int i) override
    { cout << "This is from Derived with " << i << endl; }
};

void letDoSomething(Base& base)
{
    base.doSomething(419);
}

int main()
{
    Derived d;
    letDoSomething(d);
}

```

^—————  
test.cpp:4928:16: error: 'virtual void Derived::doSomething(int)' marked 'override', but does not override  
virtual void doSomething(int i) override

也就是 `override` 表示本函数确实是重写基类中某个函数，如果函数原型与基类中同名函数不一致，就报。有了 `override` 之后，子类的虚函数可不加 `virtual` 关键字。在最顶层的虚函数上加上 `virtual` 后，其余的子类覆写后就不再加 `virtual` 了，但是要统一加上 `override`。如下，我们去掉了子类函数前部的 `virtual`，但是在尾部加了 `override`，一样实现了多态，

```

class Base {
public:
    virtual void doSomething(int i) const
    { cout << "This is from Base with " << i << endl; }
};

class Derived : public Base {
public:
    void doSomething(int i) const override
    { cout << "This is from Derived with " << i << endl; }
};

void letDoSomething(Base& base)
{
    base.doSomething(419);
}

int main()
{
    Derived d;
    letDoSomething(d);
}

```

^—————  
[Running] cd "d:\wujianjun\code\myz\zcode\torchcpp\torchcpp\src\" && g++ test.cpp -o test  
This is from Derived with 419

如果父类的某些方法被不想再被改变(无论父类方法是不是 `virtual`)，怎么办?在 C++11 出现前，很难阻止它的子类覆写这个方法。**final** 作用在函数表示后代类必定不能重写此函数:

```

class Base {
public:
    virtual void doSomething(int i) final
    { cout << "This is from Base with " << i << endl; }
};

class Derived : public Base {
public:
    void doSomething(int i) override // 错误!!! 对应函数在基类是 final, 无法被重写
    { cout << "This is from Derived with " << i << endl; }
};

void letDoSomething(Base& base)
{
    base.doSomething(419);
}

int main()
{
    Derived d;
    letDoSomething(d);
}

```

^—————  
test.cpp:4928:8: error: virtual function 'virtual void Derived::doSomething(int)' overriding final function  
void doSomething(int i) override

`final` 直接用在类上，表示这个类禁止任何其他类继承它。

## lambda 与可调用对象

<https://junqianghan.github.io/2018/05/16/cpp-callable-object/>

满足以下条件之一就可称为可调用对象：

- ◆ 一个函数指针，
- ◆ 一个具有 `operator()` 成员函数的类对象(即仿函数)，
- ◆ `lambda` 表达式：利用 `lambda` 可以编写内嵌的匿名函数，`lambda` 表达式一般都是`[]`开始，结束于花括号`{}`。如下：

```
// 暂时返回类型
auto add = [](int a, int b) -> int { return a + b; };
// 自动推断返回类型
auto multiply = [](int a, int b) { return a * b; };
int sum = add(2, 5); // 输出: 7
int product = multiply(2, 5); // 输出: 10
```

`[]`的意义何在？定义一个 `lambda` 表达式后编译器会生成一个匿名类(重载了`()`运算符)，是闭包类型。运行时会返回一个匿名的闭包实例。**闭包的一个强大之处是其可以通过传值或者引用的方式捕捉其封装作用域内的变量，`[]`就是用来定义捕捉模式以及变量。**看下面的例子：



```
int main()
{
    int x = 10;
    auto add_x = [x](int a) { return a + x; }; // 复制捕捉x
    auto multiply_x = [&x](int a) { return a * x; }; // 引用捕捉x
    cout << add_x(10) << " " << multiply_x(10) << endl;
}
```

当`[]`为空时，表示没有捕捉任何变量。但是上面的 `add_x` 是以复制的形式捕捉变量 `x`，而 `multiply` 是以引用的方式捕捉 `x`。闭包类会相应添加对应的数据成员，运行时会初始化。**还有一点要注意：lambda 表达式是不能被赋值的。**



```
auto a = [] { cout << "A" << endl; };
auto b = [] { cout << "B" << endl; };

a = b; // 非法, Lambda无法赋值

auto c = a; // 合法, 生成一个副本
```

**a 与 b 对应的函数类型是一致的，为什么不能相互赋值呢？因为禁用了赋值操作符：**

```
ClosureType& operator=(const ClosureType&) = delete;
```

但是没有禁用拷贝构造函数。**lambda 表达式也可以赋值给相对应的函数指针。**

- `[]`: 不捕获任何变量；
- `[=]`: 以值捕获所有变量；
- `[&]`: 以引用捕获所有变量；
- `[x]`: 以值捕获 `x`，其它变量不捕获；
- `[&x]`: 以引用捕获 `x`，其它变量不捕获；
- `[=, &x]`: 默认以值捕获所有变量，但是 `x` 是例外，通过引用捕获；
- `[&, x]`: 默认以引用捕获所有变量，但是 `x` 是例外，通过值捕获；
- `[this]`: 通过引用捕获当前对象(其实是复制指针)；
- `[*this]`: 通过传值方式捕获当前对象；

**注意最好不要使用`[=]`和`[&]`默认捕获所有变量!!!**

从 C++14 开始，`lambda` 表达式支持泛型：其参数可以使用自动推断类型的功能。

```
auto add = [](auto x, auto y) { return x + y; };

int x = add(2, 3); // 5
double y = add(2.5, 3.5); // 6.0
```

再看看下的例子：

```

void print(int x)
{
    cout << "正常函数or函数指针:" << x << endl;
}

class Fun // 仿函数
{
public:
    void operator()(int x) { cout << "仿函数:" << x << endl; }
};

template<class CallableType, class T>
void testf(CallableType fun, T arg)
{
    fun(arg);
}

int main()
{
    int a = 10;
    testf(print, a); // 函数指针
    testf(Fun(), a); // 仿函数
    testf([](int x){ cout<<"lambda表达式:"<< x <<endl; }, a); // lambda表达式
}

```

正常函数or函数指针:10  
仿函数:10  
lambda表达式:10

- ◆ `std::bind`: `bind` 用于生成偏函数，可以将函数对象的参数绑定至特定的值。对于没有绑定的参数可以使用 `std::placeholders::_1`, `std::placeholders::_2` 等标记。还可以嵌套从而实现函数组合。**注意：无论嵌套多深的位置 `std::placeholders::_n` 都表示调用时参数列表的第 n 个实参，调用参数可以多余所需的个数。另外，绑定器默认是以传值方式绑定参数，如果需要引用绑定值，那么要使用 `std::ref` 和 `std:: cref` 函数，分别代表普通引用和 `const` 引用绑定参数。**

```

double complete_sum(double x, double y, double z)
{
    return x + y + z;
}
double complete_prod(double x, double y, double z)
{
    return x * y * z;
}
int write_fun(int& run_num, int y)
{
    run_num++; return run_num * y;
}

int main()
{
    // std::bind 常用于生产偏函数
    // std::placeholders::_2 表示用调用时参数列表的第二个实参
    cout << "======" << endl;
    double x = 2.0, y = 3.0, z = 4.0;
    auto partialFun1 = std::bind(&complete_sum, std::placeholders::_1, y, z);
    auto partialFun2 = std::bind(&complete_sum, std::placeholders::_1, std::placeholders::_2, z);
    cout << complete_sum(x, y, z) << endl;
    cout << partialFun1(x) << endl;
    cout << partialFun2(x, y) << endl;
    // std::bind不可嵌套 (x + 1 + 0) * (0 + y + 1) * (0 + 1 + z)
    // 无论嵌套多深的位置, std::placeholders::_n 表示用调用时参数列表的第n个实参
    cout << "======" << endl;
    auto combined_fun = std::bind(&complete_prod,
        std::bind(&complete_sum, std::placeholders::_1, 1, 0),
        std::bind(&complete_sum, 0, std::placeholders::_2, 1),
        std::bind(&complete_sum, 0, 1, std::placeholders::_3));
    cout << combined_fun(x, y, z) << endl;
    cout << combined_fun(x, y, z, 70, 80) << endl; // 调用时可以传入很多参数, bind如果没引用则不起作用!!!!!
    // 绑定器默认是以传值方式绑定参数，如果需要引用绑定值，那么要使用std::ref
    cout << "======" << endl;
    int run_num = 1;
    auto ref_fun = std::bind(&write_fun, std::ref(run_num), std::placeholders::_1);
    int s1 = ref_fun(10);
    int s2 = ref_fun(10);
    cout << "run_num=" << run_num << ", s1=" << s1 << ", s2=" << s2 << endl;
}

=====  
9  
9  
9  
=====  
60  
60  
=====  
run_num=3 ,s1=20, s2=30

```

- ◆ `std::function` 是一个函数包装器模板，能包装任何类型的可调用对象。`std::function` 对象可被拷贝和转移。因为它可以延迟函数的执行，特别适合作为回调函数使用。

```

    } double complete_sum(double x, double y, double z)
} { return x + y + z; }

} double complete_prod(double x, double y, double z)
} { return x * y * z; }

} int main()
} {
    double x = 2.0, y = 3.0, z = 4.0;
    cout << "===== " << endl;
    std::function<double(double, double, double)> aggregator;
    aggregator = complete_sum;
    cout << aggregator(x, y, z) << endl;
    aggregator = complete_prod;
    cout << aggregator(x, y, z) << endl;
}

```

=====  
9  
24

### 委托构造函数

**委托构造函数允许在同一个类中一个构造函数调用另外一个构造函数。可以在变量初始化时简化操作。**不使用委托构造函数 vs 使用委托构造函数如下：

#### default

如果类中有自定义的构造函数，编译器就不会生成默认构造函数，如下代码编译出错将：

```

struct A {
    int a;
    A(int i) { a = i; }
};

int main() {
    A a; // 编译出错
    return 0;
}

```

而通过 `default`，程序员只需在函数声明后加上“`=default;`”，就可将该函数声明为 `defaulted` 函数，编译器将为显式声明的 `defaulted` 函数自动生成函数体，如下：

```

struct A {
    A() = default;
    int a;
    A(int i) { a = i; }
};

int main() {
    A a;
    return 0;
}

```

#### delete

如果开发人员没有定义特殊成员函数，编译器在需要特殊成员函数时候会隐式自动生成一个默认的特殊成员函数，例如拷贝构造函数或者拷贝赋值操作符，如下代码：

```

struct A {
    A() = default;
    int a;
    A(int i) { a = i; }
};

int main() {
    A a1;
    A a2 = a1; // 正确，调用编译器隐式生成的默认拷贝构造函数
    A a3;
    a3 = a1; // 正确，调用编译器隐式生成的默认拷贝赋值操作符
}

```

我们若禁止对象的拷贝与赋值，可以使用 `delete` 修饰，如下：

```
struct A {
    A() = default;
    A(const A&) = delete;
    A& operator=(const A&) = delete;
    int a;
    A(int i) { a = i; }
};

int main() {
    A a1;
    A a2 = a1; // 错误，拷贝构造函数被禁用
    A a3;
    a3 = a1; // 错误，拷贝赋值操作符被禁用
}
```

### constexpr

<https://www.zhihu.com/question/35614219>

`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。而如下：



```
int main()
{
    int a = 10;
    const int &v = a;
    a++; // v引用的对象是可以变的,只是v只能引用a而已
    cout << v << endl; // 11

    int b = 1;
    const int* w = &b;
    b++; // w指向的对象是可以变的,只是w只能指向a而已
    cout << * w << endl; // 2
}
```

被 `constexpr` 修饰的变量的初始值必须是常量表达式，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。`*` `&` 都不属于常量表达式。

```
constexpr int m = 10; // 10是常量表达式
constexpr int n = m + 1; // m+1是一个常量表达式

/* & 都不属于常量表达式
// constexpr const int *p = &m; // 错误
// constexpr const int &r = m; // 错误
```

`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。**也就是说：`constexpr` 修饰的函数，如果其传入的参数可以在编译时期计算出来，那么这个函数就会产生编译时期的值。但是，传入的参数如果不能在编译时期计算出来，那么 `constexpr` 修饰的函数就和普通函数一样了。**不过，我们不必因此而写两个版本，所以如果函数体适用于 `constexpr` 函数的条件，可以尽量加上 `constexpr`。

```
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
}
```

语义上：

- ◆ `constexpr`: 告诉编译器我可以是编译期间可知的，尽情的优化我吧。
- ◆ `const`: 告诉程序员没人动得了我，放心的把我传出去；或者放心的把变量交给我，我啥也不动就瞅瞅。

在 C++11 以后，建议凡是「常量」语义的场景都使用 `constexpr`，只对「只读」语义使用 `const`。

## const& , & and && specifiers for member functions in C++

<https://stackoverflow.com/questions/28066777/const-and-specifiers-for-member-functions-in-c>

Recently I was reading through the API of boost::optional and came across the lines:

```
T const& operator *() const&;
T& operator *() &;
T&& operator *() &&;
```

I know what it means to declare a member function const, but can anyone explain what it means to declare it const&, & and &&.

Alper:

it is one of the features added in C++11. It is possible to overload non-static member functions based on whether the implicit this object parameter is an lvalue or an rvalue.

```
#include <iostream>
struct myStruct {
    void fun(){std::cout << "all ok \n"; } // 左值对象和右值对象均可调用
    void func() & { std::cout << "lvalue\n"; } // 只能被左值对象调用
    void func() && { std::cout << "rvalue\n"; } // 只能被右值对象调用
};

int main()
{
    myStruct s;
    s.fun();           // prints "lvalue"
    std::move(s).func(); // prints "rvalue"
    myStruct().func(); // prints "rvalue"

    myStruct s2;
    s2.fun();
    std::move(s2).fun();
}
```

输出

```
lvalue
rvalue
rvalue
all ok
all ok
```

```
struct A {
    A(){}
    A(int a) { a_ = a; }

    A(int a, int b) { // 好麻烦
        a_ = a;
        b_ = b;
    }

    A(int a, int b, int c) { // 好麻烦
        a_ = a;
        b_ = b;
        c_ = c;
    }

    int a_;
    int b_;
    int c_;
};
```

```
struct A {
    A(){}
    A(int a) { a_ = a; }

    A(int a, int b) : A(a) { b_ = b; }

    A(int a, int b, int c) : A(a, b) { c_ = c; }

    int a_;
    int b_;
    int c_;
};
```

## 继承构造函数

继承构造函数可以让派生类直接使用基类的构造函数。如果有一个派生类，我们希望派生类采用和基类一样的构造方式，可以直接使用基类的构造函数，而不是再重新写一遍构造函数。  
不使用继承构造函数 VS 使用继承构造函数如下：

```
struct Base {
    Base() {}
    Base(int a) { a_ = a; }
    Base(int a, int b) : Base(a) { b_ = b; }

    Base(int a, int b, int c) : Base(a, b) { c_ = c; }

    int a_;
    int b_;
    int c_;
};

struct Derived : Base {
    Derived() {}
    Derived(int a) : Base(a) {} // 好麻烦
    Derived(int a, int b) : Base(a, b) {} // 好麻烦
    Derived(int a, int b, int c) : Base(a, b, c) {} // 好麻烦
};

int main() {
    Derived a(1, 2, 3);
    return 0;
}

struct Base {
    Base() {}
    Base(int a) { a_ = a; }
    Base(int a, int b) : Base(a) { b_ = b; }

    Base(int a, int b, int c) : Base(a, b) { c_ = c; }

    int a_;
    int b_;
    int c_;
};

struct Derived : Base {
    using Base::Base;
};

int main() {
    Derived a(1, 2, 3);
    return 0;
}
```

只需要使用 `using Base::Base` 继承构造函数，就免去了很多重写代码的麻烦。

再来一个例子：

```
class Base
{
public:
    Base(){ cout << "Base()" << endl; }
    Base(int i){ cout << "Base(int i)" << endl; }
    Base(const Base& other){ cout << "Base(const Base& other)" << endl; }
    Base(Base&& other){ cout << "Base(Base&& other)" << endl; }
};

class Children : public Base
{
public:
    using Base::Base; // 使用父类全部构造函数
};

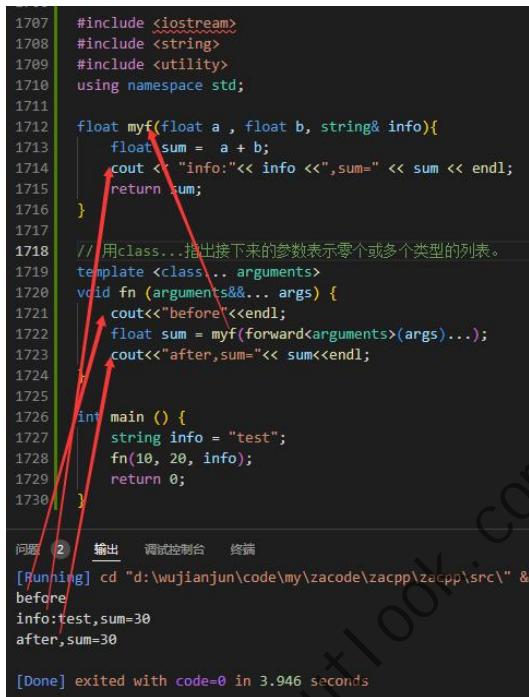
int main()
{
    Children c1;
    Children c2(2);
    Children c3(c1);
    Children c4(move(c2));
}

Base()
Base(int i)
Base(const Base& other)
Base(Base&& other)
```

## std::forward 与完美转发

### std::forward

定义在<utility>头文件中，通常是用于完美转发的，它会将输入的参数原封不动地传递到下一个函数中。示例如下：

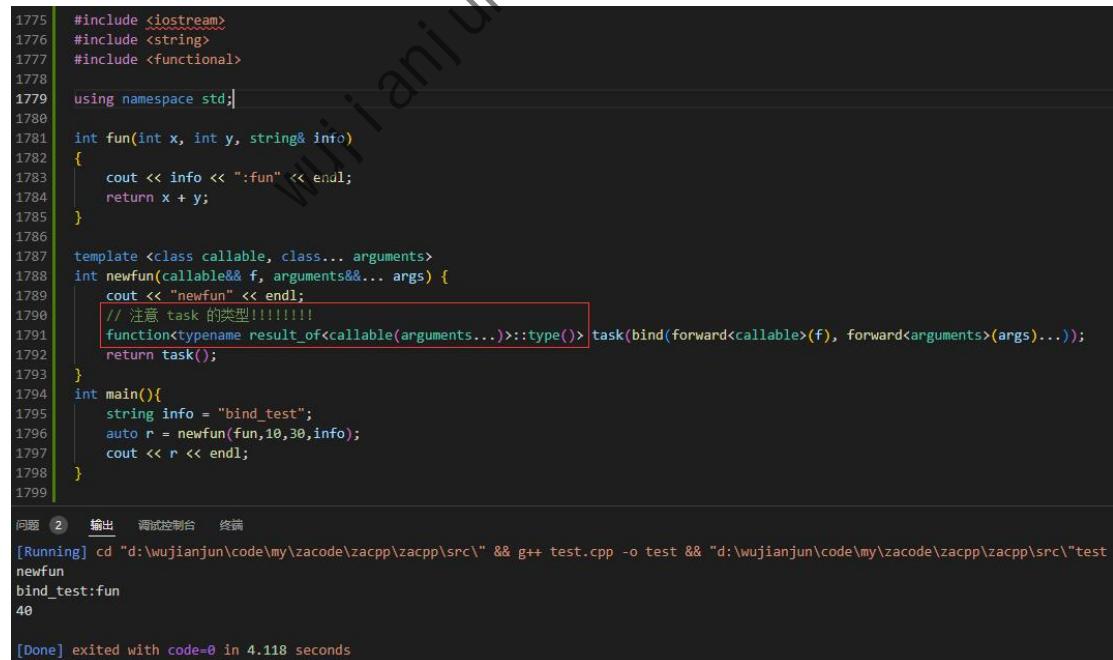


```
1707 #include <iostream>
1708 #include <string>
1709 #include <utility>
1710 using namespace std;
1711
1712 float myf(float a , float b, string& info){
1713     float sum = a + b;
1714     cout << "info:<< info <<,sum=" << sum << endl;
1715     return sum;
1716 }
1717
1718 // 用class...指出接下来的参数表示零个或多个类型的列表。
1719 template <class... arguments>
1720 void fn (arguments&&... args) {
1721     cout<<"before"<<endl;
1722     float sum = myf(forward<arguments>(args)...);
1723     cout<<"after,sum="<< sum<<endl;
1724 }
1725
1726 int main () {
1727     string info = "test";
1728     fn(10, 20, info);
1729     return 0;
1730 }
```

问题 ② 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\src\" && before  
info:test,sum=30  
after,sum=30  
[Done] exited with code=0 in 3.946 seconds

### 综合例子

下面例子显示了，在一个函数原型提前并不知道情况下，如何绑定：



```
1775 #include <iostream>
1776 #include <string>
1777 #include <functional>
1778
1779 using namespace std;
1780
1781 int fun(int x, int y, string& info)
1782 {
1783     cout << info << ":fun" << endl;
1784     return x + y;
1785 }
1786
1787 template <class callable, class... arguments>
1788 int newfun(callable&& f, arguments&&... args) {
1789     cout << "newfun" << endl;
1790     // 注意 task 的类型!!!!!!!
1791     function<typename result_of<callable(arguments...)::type()> task(bind(forward<callable>(f), forward<arguments>(args)...));
1792     return task();
1793 }
1794
1795 int main(){
1796     string info = "bind_test";
1797     auto r = newfun(fun,10,20,info);
1798     cout << r << endl;
1799 }
```

问题 ② 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\src\" && g++ test.cpp -o test && "d:\wujianjun\code\my\zacode\zacpp\src\"test  
newfun  
bind\_test:fun  
40  
[Done] exited with code=0 in 4.118 seconds

result\_of 主要用于目标函数定义的类型推导中。

用于在编译的时候推导出一个可调用对象的类型。

## { }-Initialization

### Brace initialization

<https://learn.microsoft.com/en-us/cpp/cpp/initializing-classes-and-structs-without-constructors-cpp?view=msvc-170>

太简单的类可以不需要定义构造函数，此时可以使用{}初始化实现 public 数据成员的初始化(必须按照声明的顺序)。并且不允许跳过任何数据成员。

```
class Data //没有构造函数
{
public:
    int a;
    double b;
    int c;
    string d;
};

int main() {
    Data d7; // 随机初始化
    Data d8{}; // 全零初始化
    cout << d7.b << endl << d8.b << endl;
    Data d9{1, 2.0, 3, "ffff"}; // 必须按照声明的顺序依次给值
    cout << d9.d << endl;
}
```

The terminal output shows:

```
1.18576e-322|0
ffff
```

If a class has non-default constructors, the order in which class members appear in the brace initializer is the order in which the corresponding parameters appear in the constructor.

如下例子：

```
class Demo{
public:
    Demo(int _a, int _b): a(_a),b(_b)
    { cout << "normal constructor" << endl; }

    Demo(const Demo& obj): a(obj.a),b(obj.b)
    { cout << "copy constructor" << endl; }
public:
    int a;
    int b;
};

class Data{
public:
    int a;
    int b;
};

void fun1(Demo d){ cout << "fun1" << endl; }
void fun2(Data d){ cout << "fun2" << endl; }

int main()
{
    Demo d1(10, 20); // ok
    Demo d2{10, 20}; // ok
    Demo d3 = {10, 20}; // ok

    Demo d4(d1); // ok
    Demo d5{d1}; // ok
    Demo d6 = d1; // ok

    fun1({3, 4}); // ok
    fun2({3, 4}); // ok
}
```

Output from the Jupyter cell:

```
normal constructor
normal constructor
normal constructor
copy constructor
copy constructor
copy constructor
normal constructor
fun1
fun2
```

**What are the advantages of list initialization (using curly braces)?**

Oleksiy:

列表初始化不允许窄话，如 `int -> char, double -> float`, 这些是不允许的。并且任何浮点数与任何整形之间相互不允许赋值。如下：

```
double val1 = 7.9;
int x1 = val1;      // if val1 == 7.9, x2 becomes 7 (bad)
int x2 {val1};     // error

int val2 = 10;
double x3 {val2}; // ok
double x4 {val2 }; // error
```

The only situation where `=` is preferred over `{}` is when using `auto` keyword to get the type determined by the initializer.

```
auto z1 {99};    // z1 is an int
auto z2 = {99}; // z2 is std::initializer_list<int>
auto z3 = 99;    // z3 is an int
```

*Conclusion*

**PREFER `{}` INITIALIZATION OVER ALTERNATIVES UNLESS YOU HAVE A STRONG REASON NOT TO.**

`{}`-Initialization

<https://www.modernescpp.com/index.phpinitialization>

# Using 的用法

## The 4 use of using in C++

<https://www.sandordargo.com/blog/2022/04/27/the-4-use-of-using-in-cpp>

### Aliasing

In old C++ we could use `typedef` to give an alias for our types to shorten long types:

```
typedef std::vector<std::string>::iterator Iterator;
```

Since C++11 we can use `using` instead of `typedef` to achieve the same results.

```
using Iterator = std::vector<std::string>::iterator;
```

Besides, `using` can be used for **template aliases** where `typedef` would lead to a compilation error.

```
template<typename T>
typedef std::map<int, T> MapT; // error

template<typename T>
using MapT = std::map<int, T>; // OK
```

### Using-directive in namespace and block scope

例子如下：

```
namespace Level_1 {
    namespace Level_A{
        void fun1(){ cout << "Level_1::Level_A" << endl; }
    }
    namespace Level_B{
        class Demo{
            public:
                Demo(){ cout << "Demo" << endl;}
                void print(){ cout << "Demo::print" << endl; }
        };
    }

    void test1(){
        Level_1::Level_A::fun1(); // 完整引用namespace
    }

    void test2()
    {
        using namespace Level_1; // 可以临时using一部分namespace
        Level_B::Demo d1;
    }

    using LBT = Level_1::Level_B::Demo; // 把很深namespace中的某个类型拿出来

    void test3(LBT& obj)
    {
        obj.print();
    }
}
```

### Importing base class members with using-declaration

With **using-declaration**, you can introduce base class members - including constructors - into derived classes. It's an easy way of exposing protected base class members as public in the derived class. It can be used both for functions and variables.

```
▼ class Base {
protected:
    ▼ void foo() {
        std::cout << "Base::foo()\n";
    }
    int m_i = 42;
};

▼ class Derived : public Base {
public:
    using Base::foo;
    using Base::m_i;
};

▼ int main() {
    Derived d;
    d.foo();
    std::cout << d.m_i << '\n';
}
```

问题 ② 编辑 高级控制台 终端 JUPYTER  
[Running]  
Base::foo()  
42

If you try to remove any of the two `using-declarations`, you'll see the compilation failing.

## range-based for loop

### hello world

我们先写一个简单的例子，如下：



```
class Demo
{
public:
    Demo(initializer_list<int> _data)
        : len(_data.size()), data(new int[len])
    {
        int i = 0;
        for (auto d : _data)
        {
            data[i++] = d;
        }
    }

    int* begin()
    { return data; }

    int* end()
    { return data + len; }

private:
    int len;
    int* data;
};

int main()
{
    Demo datas{1, 2, 3, 4, 5};
    for (auto const& e : datas)
    {
        cout << e << endl;
    }
}
```

### 自定义类型使用 range-based for loops

<https://blog.csdn.net/dashuniuniu/article/details/79119094>

三种 for 形式的区别如下：

- ◆ for (auto item : Vec): item 采用值传递
- ◆ for (auto &item : Vec): item 采用引用传递
- ◆ for(auto &&item : Vec): item 采用右值引用传递

要使自定义类型应用 range-based for loops, 关键在于 begin() 和 end(), 有两种方法实现他们：

- ◆ 作为自定义类型的成员方法
- ◆ 作为普通函数定义在该类型的定义所在的 namespace 中

<https://blog.csdn.net/zhouguoqionghai/article/details/51804397>

自定义类型怎样才能进行基于范围的 for 循环操作呢？只要满足下边几个条件即可：

- ◆ 定义该容器相关的迭代器，这里的迭代器是广义的迭代器，指针也属于该范畴。
- ◆ 该类型拥有 begin() 和 end() 成员方法，返回值为迭代器。
- ◆ 迭代器支持 != 比较操作，支持 ++，支持 \* 解引用操作。

<https://en.cppreference.com/w/cpp/language/range-for>

for ( range-declaration : range-expression ) loop-statement

等价于

```
auto && __range = range-expression ;
for (auto __begin = begin Expr, __end = end Expr; __begin != __end; ++__begin)
{
    range-declaration = * __begin;
    loop-statement
}
```

一个例子如下：

```
template <typename T>
class MyIterator
{
public:
    MyIterator(T* raw_data, unsigned int init_pos, unsigned int _step_size)
        : iter(raw_data + init_pos), step_size(_step_size) { }
    T& operator*()
    { return *iter; }
    bool operator!=(const MyIterator& that)
    { return this->iter != that.iter; }
    MyIterator& operator++(){ iter += step_size; return *this; }
private:
    T* iter;
    unsigned int step_size;
};

template<typename T>
class Mycollection
{
public:
    Mycollection(initializer_list<T> _datas, unsigned int _step_size)
        : len(_datas.size()), data(new T[len]), step_size(_step_size){
            int i = 0;
            for (auto d : _datas){ data[i++] = d; }
    }
    // begin()返回的对象会做++,且for循环退出后会被析构,所以这里应该返回临时对象
    MyIterator<T> begin()
    { return MyIterator<T>(data, 0, step_size); }
    MyIterator<T> end()
    { return MyIterator<T>(data, len, step_size); }
};

int main()
{
    initializer_list<int> data = {1,2,3,4,5,6,7,8};
    Mycollection<int> collection1 = {data, 1}; // 步长为1遍历
    for(auto & item: collection1){ cout << item << ","; }
    cout << endl;
    Mycollection<int> collection2 = {data, 2}; // 步长为2遍历
    for(auto & item: collection2){ cout << item << ","; }
    cout << endl;
    Mycollection<int> collection3 = {data, 3}; // 步长为3遍历
    // 注意,这里会发生段错误,因为 __begin != __end 永远为真!!!!!!
    for(auto & item: collection3){ cout << item << ","; }
    cout << endl;
}
```

另外，注意下面的坑的：

```
initializer_list<int> data = {1,2,3,4,5,6,7,8};
Mycollection<int> collection3 = {data, 3}; // 步长为3遍历
// 注意,这里会发生段错误,因为 __begin != __end 永远为真!!!!!!
for(auto & item: collection3){ cout << item << ","; }
cout << endl;

[Running] cd "d:\wujianjun\code\my\zacode\torchcpp\torchcpp\src\" && g+
[Done] exited with code=3221225477 in 1.255 seconds
```

How to make my custom type to work with "range-based for loops"?

<https://stackoverflow.com/questions/8164567/how-to-make-my-custom-type-to-work-with-range-based-for-loops>

How to Make Your Classes Compatible with Range for Loop

<https://www.fluentcpp.com/2021/09/02/how-to-make-your-classes-compatible-with-range-for-loop/>

## 继承中的新特性

public virtual

In C++, what is a virtual base class?

<https://stackoverflow.com/questions/21558/in-c-what-is-a-virtual-base-class>

`static_pointer_cast` 与 `dynamic_pointer_cast`

`std::dynamic_pointer_cast`

`std::static_pointer_cast`

`std::reinterpret_pointer_cast`

CRTP

C++ 中 CRTP 的用途

<https://zhuanlan.zhihu.com/p/137879448>

C++ 惯用法 CRTP 简介

<https://liam.page/2016/11/26/Introduction-to-CRTP-in-Cpp/>

# c++指令编程

## 编译指令

### 介绍

语法: #pragma token\_string

#### #pragma once

指定该文件在编译源代码文件时仅由编译器包含一次。这和使用预处理宏定义来避免多次包含文件的内容的效果是一样的，但是需要键入的代码少。

```
使用#pragma once
#pragma once
// Code placed here is included only once per translation unit

使用宏定义方式
#ifndef HEADER_H_
#define HEADER_H_
// Code placed here is included only once per translation unit
#endif // HEADER_H_
```

#### #pragma message(messageString)

不中断编译的情况下，发送一个字符串文字量到标准输出。message 编译指示的典型运用是在编译时显示信息，例如：

```
#if _M_IX86 >= 500          //查看定义的宏有没有大于500，或者有时用作检查有没有定义宏
#pragma message("_M_IX86 >= 500")
#endif
```

#### #pragma pack([ show ] | [ push | pop ] [, identifier], n)

指定结构、联合和类成员的封装对齐。其实就是改变编译器的内存对齐方式。

#pragma pack(show)	以警告信息的形式显示当前字节对齐的值.
#pragma pack(n)	将当前字节对齐值设为 n .
#pragma pack()	将当前字节对齐值设为默认值(通常是8) .
#pragma pack(push)	将当前字节对齐值压入编译栈栈顶.
#pragma pack(pop)	将编译栈栈顶的字节对齐值弹出并设为当前值.

#### #pragma omp

OpenMP 的并行计算程序总是以#pragma omp 作为开始的。

#### #pragma omp parallel

如果使用上面这条简单的指令去运行并行计算的话，程序的线程数将由运行时系统决定，典型的情况下，系统将在每一个核上运行一个线程。

如果需要执行使用多少个线程来执行我们的并行程序，就得为 parallel 指令增加 num\_threads 子句，这样的话，就允许程序员指定执行后代码块的线程数。

#### #pragma omp parallel num\_threads(thread\_count)

## C/C++指令集

Intel 的 CPU 的指令集：

- ◆ MMX 指令集：多媒体扩展指令集。MMX 指令集支持算数、比较、移位等运算，MMX 指令集的向量寄存器是 64bit。
- ◆ SSE 指令集：Streaming SIMD Extensions，单指令多数据流扩展。SSE 向量寄存器由 MMX 的 64bit 拓展到 128bit；包括了 SIMD 的浮点和整型运算的指令以及整型和浮点数据之间的转换；
- ◆ AVX 指令集：Advanced Vector Extensions。AVX 是在之前的 SSE128 位扩展到和 256 位的单指令多数据流。增强了数据重排和灵活的不对齐地址访问；

SSE 的指令集是 X86 架构 CPU 特有的，对于 ARM 架构、MIPS 架构等 CPU 是不支持的。而且 Intel 和 AMD 对于同样的 128bit 向量的指令语法是不一样的，所以，在 Intel 之下所写的代码并不能一直到 AMD 的机器上进行指令集加速。也就是说，写的某一种指令加速代码，不具备完全的可移植性。

SSE 指令中 `intrinsics` 函数的数据类型为：

- ◆ `_m128`(单精度浮点数)，如果使用 `sizeof(_m128)` 计算该类型大小，结果为 16，即等于四个单精度浮点数长度。
- ◆ `_m128d`(双精度浮点数)

SIMD 即单指令多数据流。目前常见编译器对 X86-64 的 CPU 上 128bit 的 SIMD 计算支持比较好。SIMD 指令，可以一次性装载多个元素到寄存器。如果是 128 位宽度，则可以一次装载 4 个单精度浮点数。这 4 个 `float` 可以一次性地参与乘法计算，理论上可提速 4 倍。

我们再用 SIMD 写一个矢量相加，要使用 SIMD 技术需要 `immintrin.h` 头文件。

```
#include <iostream>
#include <immintrin.h>
#define N 20000000
using namespace std;
int main(){
    double *x,*y,*z,*px,*py,*pz;
    x=(double*)_mm_malloc(sizeof(double)*N,16); //申请内存并且按照2的4次方对齐地址
    y=(double*)_mm_malloc(sizeof(double)*N,16);
    z=(double*)_mm_malloc(sizeof(double)*N,16);
    px=x;
    py=y;
    pz=z;
    _m128d vx,vy,vz;// _m128d 是 SSE 指令集中操作双精度浮点数对应的数据类型
    for(int i=0;i<N/2;i++){
        vx=_mm_load_pd(px);//从 px 指向的内存中取出两个数，放入 vx
        vy=_mm_load_pd(py);//从 py 指向的内存中取出两个数，放入 vy
        vz=vx+vy;//计算 vx+vy 并将结果放入 vz
        _mm_store_pd(pz,vz);//将 vz 中的结果放入 pz 指向的内存
        px+=2;//由于前面取出了两个数据，所以指针后移两位
        py+=2;
        pz+=2;
    }
    _mm_free(x); //释放_mm_malloc申请的内存
    _mm_free(y);
    _mm_free(z);
    return 0;
}
```

SIMD 的官方解释文档在这里：

[https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=\\_mm\\_setr\\_epi32](https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=_mm_setr_epi32)

The screenshot shows the Intel® Intrinsics Guide website. On the left, there's a sidebar with a tree view of the instruction set categories: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, FMA, AVX\_VNNI, AVX-512, KNC, AMX, SVM, and Other. Below that is a 'Categories' section with an 'Application-Targeted' checkbox. The main content area is titled '\_mm\_setr\_epi32'. It includes a synopsis of the assembly code, a description stating it sets packed 32-bit integers in dst with supplied values in reverse order, and an operation section with assembly code: `dst[31:0] := e3  
dst[63:32] := e2  
dst[95:64] := e1  
dst[127:96] := e0`.

## c/c++中的条件编译

我们要写一个跨平台项目，要求项目既能在 Windows 下运行，也能在 Linux 下运行。此时我们可以使用条件编译。条件编译就是根据一定的条件进行选择性的编译。我们要达到的效果，就是在 Windows 环境下有的语句根本不会编译，这样生成的可执行文件中，也不会还有对应语句的机器码，这样既提高了编译效率，同时也减小了可执行文件的体积。

Windows 有专有的宏 `_WIN32`，Linux 有专有的宏 `_linux_`。我们使用 if-else，如下：

```
#ifdef(_WIN32)
    printf("Windows下执行的代码\n");
#elif (_linux_)
    printf("Linux下执行的代码\n");
#else
    printf("未知平台不能运行!\n");
#endif
```

在 gcc 中我们可以使用-D 选项，动态地定义程序所需要的宏。比如我们可以这样编译 `gcc test.c -o test -D _WIN32`，这样程序就可以在 Windows 下运行了（当然，实际情况是在 Windows 环境下 `_WIN32` 已经被定义）。gcc 中的-D 选项会默认将宏定义为 1，如果要定义为其他的值使用等于号如：`-D _WIN32=0`。

## C 语言之 gcc 中支持的内存对齐指令

`__attribute__((packed))`: 取消内存对齐，或者说是 1 字节对齐。

```
struct mystruct1
{
    int a;
    char b;
    short c;
}__attribute__((packed));
```

这样这个结构体类型就按 1 字节对齐，所以这个结构体类型占 7 字节。`__attribute__((packed))` 使用时直接放在要进行内存对齐的类型定义的后面，然后它起作用的范围只有加了这个东西的这一个类型。

`__attribute__((aligned(n)))`: 设定结构体类型整体按 n 字节对齐，注意是整体而不是这个结

构体变量内的元素按 n 字节对齐。

```
struct mystruct2
{
    int a;
    char b;
    short c;
}__attribute__((aligned(2))) mystr2;
```

这样 mystruct2 这个结构体类型就按 2 字节对齐。`__attribute__((aligned(n)))` 使用时直接放在要进行内存对齐的类型定义的后面，然后它起作用的范围只有加了这个东西的这一个类型。它的作用是让整个结构体变量整体进行 n 字节对齐(注意是结构体变量整体 n 字节对齐，而不是结构体内各元素也要 n 字节对齐)。

wuji anjunml@outlook.com

# 宏编程，泛型编程与元编程

## C/C++宏编程

- ◆ 宏定义执行的是直接替换。对象宏一般用来定义一些常数，比如：`#define PI 3.14159`。  
函数宏是用得最多的，例如：

```
#define MIN(A, B) A < B ? A : B
int a = 2 * MIN(3, 4); // 注意，展开后运算符优先级的问题，乘法先被运算

#define MIN(A, B) ((A) < (B) ? (A) : (B)) // 宏的没参数都要加括号，且整个表达式也要加个括号!!!!!
int a = MIN(3, 4 < 5 ? 4 : 5)
```

应该谨慎地将宏定义中的“参数”和整个宏都用**括弧括起来**。

- ◆ `#`会将宏参数转换为一个字符串：



```
51 #define LOG(msg) printf("LOG:%s\n",#msg)
52
53 int main(){
54     LOG("start.....");
55     LOG(10+10); // 注意，我们这里给的参数会被转为字符串
56 }
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2

LOG:"start....."  
LOG:10+10

- ◆ 而`##`被称为连接符，用来将两个 Token 连接为一个 Token。



```
51 #define NAME(name) string function_##name
52
53 NAME(out)(string info){ //宏替换后这里是 string function_out(string info)
54     return info;
55 }
56
57 int main(){
58     string r = function_out("==start=="); // 我们并没有显示地声明函数function_out!!!!, 但实际上这个函数是存在
59     cout << r << endl;
60 }
61
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2

==start==

这样可以管理函数名称，比如下面实现函数前缀的统一管理：

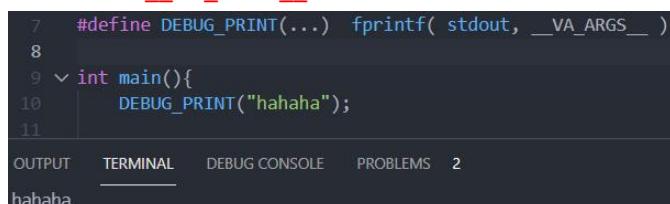


```
35 #define CHOLMOD(name) cholmod_ ## name
36
37 int CHOLMOD(add) // 编译器首先把这里替换为 cholmod_add, 所以下面我们调用函数cholmod_add就是正确的
38 (int a,int b){
39     return a + b;
40 }
41
42
43 int main(){
44     cout << cholmod_add(10, 20) << endl; // 注意 cholmod_add 函数在宏替换后是定义了
45     cout << CHOLMOD(add)(10, 20) << endl; // 也可以通过宏来
46 }
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2

30  
30

- ◆ 宏可以带有可变参数，比如：`#define LOG(format, ...) fprintf(stdout, format, __VA_ARGS__)`  
其中，...表示参数可变，`__VA_ARGS__`在预处理中为实际的参数集所替换。



```
7 #define DEBUG_PRINT(...) fprintf( stdout, __VA_ARGS__ )
8
9 int main(){
10     DEBUG_PRINT("hahaha");
11 }
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2

hahaha

- ◆ 我们可以在调试环境下定义 LOG 是一个变参输出宏；在发布环境下，LOG 是一个空宏：

```

#ifndef DEBUG
#define LOG(format, ...) fprintf(stdout, ">> "format"\n", ##__VA_ARGS__)
#else
#define LOG(format, ...)
#endif

```

- ◆ 使用宏还需要注意几点：
  - ◆ 如果某一个标识符被定义为宏名后，在取消该宏定义之前，不允许重新对它进行宏定义。应该及时取消宏定义：`#undef <标识符>`。**#undef 比较多的用途是在使用宏之前先进行 #undef 以保证这个宏不会与先前的定义冲突。**
  - ◆ 编译器对宏只进行简单的文本替换，而不会进行语法检查。因为宏不会检查语法，有时间这是个优点，比如：`#define MIN(x,y) ((x)<(y)?(x):(y))`。该宏适用于任何实现了operator<的类型，这点与 template 类似。
  - ◆ 防止宏的副作用，比如宏为定义：`#define MIN(A,B) ((A) <= (B) ? (A) : (B))`，所以 `MIN(*p++, b)` 的作用结果是：`((*p++) <= (b) ? (*p++) : (*p++))`。这个表达式会产生副作用，指针 p 会作三次++自增操作。
  - ◆ 宏不支持递归。为了防止无限制递归展开，语法规定，当一个宏遇到自己时，就停止展开。
  - ◆ ANSI 标准预定义的宏：
    - ◆ `_LINE_`: 当前源代码行号；
    - ◆ `_FILE_`: 当前源文件名；
    - ◆ `_DATE_`: 当前的编译日期；
    - ◆ `_TIME_`: 当前编译时间；
    - ◆ `_TIMESTAMP_` 当前源文件的最后修改时间。
  - ◆ gcc 中还定义了 `_func_` 标识当前的函数名，debug 编译时还定义了 `_DEBUG` 宏。
  - ◆ 我们可以通过执行命令 `gcc -P -E a.cpp -o a.cpp.i` 来让编译器对文件 `a.cpp` 只执行预处理并保存结果到 `a.cpp.i` 中。这样我们就可以看见预处理之后的结果了，包括宏展开的情况。
- ◆ do{...}while(0)技巧  
**这个技巧非常漂亮！**比如：

```
#define SAFE_DELETE(p) do{ delete p; p = NULL;} while(0)
```

假设这里去掉 `do...while(0)`，于是这样定义：

```
#define SAFE_DELETE(p) delete p; p = NULL; .
```

那么以下代码：

```
if(NULL != p) SAFE_DELETE(p)
else ...do sth...
```

就有两个问题，

- 因为 if 分支后有两个语句，else 分支没有对应的 if，编译失败
- 假设没有 else, `SAFE_DELETE` 中的第二个语句无论 if 测试是否通过会永远执行。

你可能发现，为了避免这两个问题，直接用 {} 括起来就可以了

```
#define SAFE_DELETE(p) { delete p; p = NULL; }
```

但是想对于 C++ 程序员来讲，在每个语句后面加分号是一种约定俗成的习惯，这样的话，以下代码：

```
if(NULL != p) SAFE_DELETE(p);
else ...do sth...
```

其 else 分支就无法通过编译了，所以采用 do...while(0) 是做好的选择了。

## C/C++泛型编程

### 模板

- ◆ 在定义类模板或者函数模板时，`typename` 和 `class` 关键字都可以用于指定参数类型。  
但是他们二者也有一些生僻的区别，见 <https://liam.page/2018/03/16/keywords-typename-and-class-in-Cxx/>

先看函数模板的例子：

```
4319 // 用class 声明类型参数
4320 template <class T>
4321 void Swap(T & x, T & y)
4322 {
4323     T tmp = x;
4324     x = y;
4325     y = tmp;
4326 }
4327 // 用 typename 声明类型参数
4328 template <typename T>
4329 T Max(T const & a, T const & b)
4330 {
4331     return a < b ? b : a ;
4332 }
4333 // 多个类型参数,且有默认类型,且可以表示常量的非类型参数
4334 template<class T1, class T2 = string, int size=2>
4335 void print(T1 arg1, T2 arg2)
4336 {
4337     cout << arg1 << " " << arg2 << "," << size << endl;
4338 }
4339
4340 int main()
4341 {
4342     int n = 1, m = 2;
4343     Swap(n, m); // 编译器自动生成 void Swap(int &, int &) 函数
4344     float a = 1.2, b = 2.3;
4345     Swap<float>(a, b); // 指定模板函数的类型参数
4346
4347     float x = 0.1, y = 0.2;
4348     Max(x, y);
4349
4350     int i = 100;
4351     string j = "abc";
4352     print<double>(i, j);
4353     print<double, string, 10>(i, j);
4354
4355     // int k = 30;
4356     // print<double, string, k>(i, j); // 报错,必须是常量
4357 }
```

问题 立即解决 调试控制台 终端 JUPYTER

```
100 abc,2
100 abc,10
```

再看类模板的例子：

```
template <class T1, class T2>
class Base
{
public:
    Base(T1 _v1, T2 _v2) : v1(_v1), v2(_v2) {}
public:
    T1 v1;
    T2 v2;
};

template <class T>
class Child : public Base<int, double>
{
public:
    Child(int _v1, double _v2, T _v) : Base(_v1, _v2), v(_v) {}
public:
    T v;
};

int main()
{
    Child<float> obj(1, 1.0, 3.0);
}
```

关于模板类型推导，最终是按照得到实参推导，至于调用时有无指定，指定了什么类型，默认是什么类型，只要相互不冲突，都可以，如下：

```
1 template<typename T = int, typename S>
2 void fun(T v, S s)
3 {
4     cout << "v=" << v << endl;
5     cout << "s=" << s << endl;
6 }
7
8 int main()
9 {
10     int a = 10;
11     float b = 10.15;
12     double c = 5.656;
13     fun<int, float>(a, b);
14     fun<double>(c, b); // T是double, 被显示指定; S是float,根据实参推导。
15     fun<float>(1, 2.0); // T是int,根据实参或者默认类型推导; S是float,根据实参推导。
16 }
```

```
v=10
s=10.15
v=5.656
s=10.15
v=1
s=2
```

### Alias Templates

<https://www.modernescpp.com/index.php/alias-templates-and-template-parameters>

例子如下：

```
template <typename T, int Line, int Col>
class Matrix{
public:
    Matrix(): data(new T[Line * Col]){}
    T * data;
};

template <typename T, int Line>
using Square = Matrix<T, Line, Line>;

template <typename T, int Line>
using Vector = Matrix<T, Line, 1>;

int main()
{
    Matrix<int, 5, 3> ma;
    Square<double, 4> sq;
    Vector<float, 5> vec;
}
```

```
问题    输出    调试控制台    终端    JUPYTER
[Running] cd "d:\wujianjun\code\my\zacode\torchcpp\tutorial\alias\src"
```

we define class template `Matrix`. For readability, we want to have two special matrices: a `Square` and a `Vector`. The keyword `using` declares a type alias.

## 可变参数与可变参数模板 Variadic template

### C++可变参数函数

[https://blog.csdn.net/qq\\_35280514/article/details/51637920](https://blog.csdn.net/qq_35280514/article/details/51637920)

C++11 引入了 `initializer_list`, 可以实现可变参数。参数必须放在一组“{}”(大括号)内。同一个 `initializer_list` 中的参数具有相同的类型。`initializer_list` 是一个编译器支持的容器类模板。

```
template<class T>
T sum(initializer_list<T> datas){
    T sum = 0;
    for(auto p = datas.begin(); p != datas.end(); p++) //使用迭代器访问参数
        sum += *p;
    return sum;
}

int main()
{
    float s1 = sum<float>({1.0,2.0,3.0,4.0});
    double s2 = sum<double>({1.0,2.0,3.0,4.0});
}
```

### Introduction to C++ Variadic Templates

<https://kevinushey.github.io/blog/2016/01/27/introduction-to-c++-variadic-templates/>

We'll create a variadic function `ignore` that does nothing.

```
template <typename... Ts> // (1)
void ignore(Ts... ts) {} // (2)

int main()
{
    ignore();
    ignore(1,"anc");
    ignore(1,"anc",true);
    ignore<int, double, bool>(1, 2.0, true); // 等同于 ignore(1, 2.0, true)
}
```

We use `typename... Ts` to declare `Ts` as a **template parameter pack**. Our function signature accepts a **function parameter pack**, a bag of parameters, whose types are given by the template parameter pack. It is declared as `Ts... ts`, with the ellipsis operator used to indicate that `Ts` does refer to a template parameter pack. Now, let's implement a variadic sum – it'll take a bunch of arguments, and just attempt to add them all up. We'll assume that the function returns a double for now. **you probably wished you could write was something like:**

```
template <typename... Ts>
double sum(Ts... ts) {
    double result = 0.0;
    for (auto el : ts)
        result += el;
    return result;
}
```

**Unfortunately, this won't do work. you can't think in the standard 'iterative' C++ style. You need to write such functions recursively – with a 'base' case, and a 'recursive' case that reduces, eventually, into a 'base' case. This implies a separate function for each case.**

let's start with a working example, and then break it down.

```
// The base case: we just have a single number.
template <typename T>
double sum(T t) {
    return t;
}
// The recursive case.
template <typename T, typename... Rest>
double sum(T t, Rest... rest) {
    return t + sum(rest...);
}

int main()
{
    double s = sum(1.0, 2.0, 3.0);
    cout << s << endl;
}
```

We have our ‘base’ case, accepting one argument T, and our ‘recursive’ case, accepting one or more arguments T and Rest. **Base case to overload your recursive case.**

How exactly does this work? for example, sum(1.0, 2.0, 3.0).

- ◆ **The compiler generates code for sum(1.0, 2.0, 3.0).** we’re passing in three arguments, the base case does not apply (it only accepts a single argument), so we select the recursive case.
- ◆ Type deduction is performed – the compiler deduces  $T = \text{double}$ , and puts the rest in our parameter pack, with Rest = <double, double>.
- ◆ The compiler generates code for  $t + \text{sum}(\text{rest...})$ . It sees the recursive call to  $\text{sum}(\text{rest...})$ .  
**Note the use of ... to ‘unpack’ the template argument – this has the effect of transforming  $\text{sum}(\text{rest...})$  to  $\text{sum}(2.0, 3.0)$ .**
- ◆ The compiler generates code for sum(2.0, 3.0). we select the recursive case again.
- ◆ Type deduction is performed – the compiler deduces T = double, and Rest = <double>. notice that we have now unpacked our original <double, double> pack to T = double and Rest = <double>.
- ◆ The compiler generates code for  $t + \text{sum}(\text{rest...})$ . It sees the recursive call to  $\text{sum}(\text{rest...})$  – this time, with  $\text{sum}(\text{rest...})$  expanding to simply sum(3.0).
- ◆ The compiler generates code for sum(3.0). We have now finally hit our base case. The compiler generates code for the base case, and we’re done with the recursion.

the expression expands in the following way

```
sum(1.0, 2.0, 3.0);
1.0 + sum(2.0, 3.0);
1.0 + (2.0 + sum(3.0));
1.0 + (2.0 + (3.0));
```

Let’s make our function a little bit more clever: we compute something like:

```
// A function that ‘squares’ a number.
template <typename T>
T square(T t) { return t * t; }
// Our base case just returns the value.
template <typename T>
double power_sum(T t) { return t; }
// Our new recursive case.
template <typename T, typename... Rest>
double power_sum(T t, Rest... rest) {
    // square(4.0, 6.0)... 将被展开为 square(4.0), square(6.0)
    // 注意,这里的square(rest)...
    return t + power_sum(square(rest)...);
}

int main()
{
    double ss1 = power_sum(2, 4, 6);
}
```

Notice the expression  $\text{square}(\text{rest})$ .... Recall that the ... operator will expand an entire expression, so for example, when it’s called with  $\text{square}(4.0, 6.0)$ ..., the compiler expands this as  $\text{square}(4.0)$ ,  $\text{square}(6.0)$ . Let’s trace the expansion of the resulting code:

```
power_sum(2, 4, 6);
2 + power_sum(square(4, 6))...; // return t + power_sum(square(rest)...);
2 + power_sum(square(4), square(6)); // square(rest)... 展开
2 + square(4) + power_sum(square(square(6))...); // return t + power_sum(square(rest)...);
2 + square(4) + power_sum(square(square(6))); // square(rest)... 展开
2 + square(4) + square(square(6)); // base case
2 + 4*4 + square(6*6); // square函数
2 + 4*4 + (6*6)*(6*6); // square函数
```

**... can be used to expand a whole expression containing a parameter pack – this is what makes it so powerful! On the downside, this expansion can only occur in certain contexts, e.g. within a function call. Clever use of ... expansion can allow you to avoid recursion in some cases.**

## Variadic templates in C++

<http://web.archive.org/web/20190608070158/https://eli.thegreenplace.net/2014/variadic-templates-in-c/>

wujianjunml@outlook.com

## C++ Template Specialization Using Enable If

<https://leimao.github.io/blog/CPP-Enable-If/>

In C++ metaprogramming, `std::enable_if` is an important function. Using types that are not enabled by `std::enable_if` for template specialization will result in compile-time error.

下面的是非类型模板例子 vs 类型模板的例子：

```
template <int N = 10> // 有默认值的常量
struct A
{
    int v{N};
};

template <int, int>
struct B
{
    int v{0};
};

int main()
{
    A<1> a{};
    A<> b{};
    B<10, 10> c{};
}
```

```
template <typename = int>
struct A
{
    int v{0};
};

template <typename, typename>
struct B
{
    int v{0};
};

int main()
{
    A<float> a{};
    A<> b{};
    B<int, int> c{};
}
```

`std::enable_if` 的定义如下：

`template< bool B, class T = void >`

`struct enable_if;`

If B is true, `std::enable_if` has a public member `typedef type, equal to T`; otherwise, there is no member `typedef`. `std::enable_if` could be implemented as follows:

```
template<bool B, class T = void>
struct enable_if {};
```

```
template<class T>
struct enable_if<true, T> : public T { typedef T type; };
```

This means, whenever the implementation tries to access `enable_if<B,T>::type` when `B = false`, the compiler will raise compilation error, as the object member type is not defined. 如下：

```
template<bool B, class T = void>
struct my_enable_if {};
```

```
template<class T>
struct my_enable_if<true, T> : public T { typedef T type; };
```

```
int main()
{
    my_enable_if<true, int>::type a = 0; // int
    cout << ++a << endl; // 1
    my_enable_if<false, int>::type b = 0; // 编译报错, 'type' is not a member of 'my_enable_if<false, int>'
```

Since C++14, there is an additional helper shortcut `std::enable_if_t`.

```
template< bool B, class T = void >
using enable_if_t = typename enable_if<B,T>::type;
```

```
std::enable_if_t<true, int> b = 0;
cout << ++b << endl; // 1
```

they make compiler to prevent any undesired types used for template specialization.

下面我们展示一下 `std::enable_if_t` 的实际使用例子，先看：

`template< class T >`

`struct is_integral;`

Checks whether T is an integral type. the member value is equal to true if T is the type bool, char, short, int, long, long long, any signed and unsigned variants. Otherwise, value is equal to false.



The screenshot shows a Jupyter notebook terminal window. It contains the following code and its output:

```
5
cout << std::is_integral<int>::value << endl;
cout << std::is_integral<long>::value << endl;
cout << std::is_integral<float>::value << endl;
cout << std::is_integral<string>::value << endl;
```

The output is:

```
1
1
0
0
```

类似的还有很多其他 `type_traits`，见 [https://en.cppreference.com/w/cpp/header/type\\_traits](https://en.cppreference.com/w/cpp/header/type_traits)

```

5429 template <typename T>
5430 void fun1()
5431 { std::cout << "T could be any one" << std::endl; }
5432
5433 // T如果是整形，则fun2是带一个类型模T的函数，否则将报错。
5434 // 这里 void 这个类型仅仅占位而已！！！也可以是其他任何类型。
5435 template <typename T, typename = std::enable_if_t<std::is_integral<T>::value, void>>
5436 void fun2()
5437 { std::cout << "T could only be integral" << std::endl; }
5438
5439 // T 如果是整形，那么 std::enable_if_t<std::is_integral<T>::value, bool> = true 等于 bool = true,
5440 // 则 bar 有两个模板参数，第一个是类型参数为T，第二个为非类型参数，类型是bool，默认值为 true
5441 // 否则将报错。
5442 template <typename T, std::enable_if_t<std::is_integral<T>::value, bool> = true>
5443 void bar()
5444 { std::cout << "T could only be integral" << std::endl; }
5445
5446 int main()
5447 {
5448     fun1<int>();
5449     // fun1<int, 20>(); // 报错，wrong number of template arguments
5450     fun1<float>();
5451     fun1<string>();
5452
5453     fun2<int, int>();
5454     // fun2<int, 20>(); // 报错，no matching function for call
5455     // fun2<float>(); // 报错,no matching function for call to
5456     // fun2<string>(); // 报错,no matching function for call
5457
5458     bar<int>();
5459     bar<int, 20>();
5460     // bar<int, 1.2>(); //报错，could not convert template argument '1.2e+0' from 'double' to 's
5461     // bar<int, float>(); //报错，no matching function for call
5462     // bar<float>(); // 报错，no type named 'type' in 'struct std::enable_if<false, bool>'
5463     // bar<string>(); // 报错，no type named 'type' in 'struct std::enable_if<false, bool>'
5464 }

```

问题 ③ 输出 调试控制台 终端 JUPYTER

```

T could be any one
T could be any one
T could be any one
T could only be integral
T could only be integral
T could only be integral

```

**Notice that we used the type template parameter compile-time checking for the function fun2 and used the non-type template parameter compile-time checking for the function bar.**

我们现在来看看如何约束函数的返回值，首先看看 `is_constructible`:

`template <class T, class... Args>`

`struct is_constructible;`

表示类型 `T` 是否可以从 `Args` 代表的多个类型构造出来，如果可以对应的 `value` 为 `true`，否则为 `false`。如下例子：

```

class A
{
public:
    A(int a, float b = 0.1){}
};

int main()
{
    cout << std::is_constructible<A, int>::value << endl; // true, 因为构造函数有默认参数
    cout << std::is_constructible<A, int, float>::value << endl; // true, 构造函数正是这样定义的
    cout << std::is_constructible<A, float, float>::value << endl; // true, 因为float可以转为 int, 虽然精度会损失

    cout << std::is_constructible<A, string>::value << endl; // false, string 无法自动转为 int
    cout << std::is_constructible<A, int, string>::value << endl; // false, string 无法自动转为 float
}

```

1  
1  
1  
0  
0

最后，我们给出返回值的类型限制例子：

```
  v typename std::enable_if<true, int>::type fun1()
  {
    cout << "std::enable_if<true, int>::type" << endl;
    int a = 100;
    return a;
  }

  v struct A
  {
    A(int i){}
  };

  v struct B
  {
    B(){ cout << "B is construct" << endl; }
  };

  template <typename T>
  typename std::enable_if<std::is_constructible<T, int>::value, std::unique_ptr<B>>::type // 这行的 typename 关键词必不可少!!!!
  v fun4(T t)
  {
    cout << "std::enable_if<std::is_constructible<T, int>::value, B>::type" << endl;
    return std::make_unique<B>();
  }

  v int main()
  {
    int a = fun1();

    A obj(1);
    auto f = fun4<A>(obj);
  }
```

问题 ③ 输出 调试控制台 终端 Jupyter

```
[Running] cd "d:\wujianjun\code\my\zancode\torchcpp\torchcpp\src\" && g++ test.cpp -o test && "d:\wujianjun\code\my\zancode\torchcpp\torchcpp\src\test"
std::enable_if<true, int>::type
std::enable_if<std::is_constructible<T, int>::value, B>::type
B is construct
```

## C++ variadic template recursive example

[https://raymii.org/s/snippets/Cpp\\_variadic\\_template\\_recursive\\_example.html](https://raymii.org/s/snippets/Cpp_variadic_template_recursive_example.html)

Variadic templates allow you to have a template with a variable number of arguments, also called a parameter pack.

*When it comes to handling variadic functions, you can't think in the standard iterative C++ style. You need to write such functions recursively, with a base case, and a recursive case that reduces, eventually, into a base case. This implies a separate function for each case.*

Placing the ellipsis to the left of the parameter name declares an parameter pack. You use this in the template declaration, like so:

```
template <typename First, typename... Args>
void Foo(First first, Args... args) { }
```

Placing the ellipsis to the right of the parameter will cause the whole expression that precedes the ellipsis to be repeated for every subsequent argument unpacked from the argument pack. In our example, it is used in the variadic function to call the base function:

```
Foo(args...);
```

The below Foo() example is recursive. The template function with First and Args... calls the template with Arg, which performs the actual action we want. Both functions have the same name, thus being overloaded. There also is a function (not template) which takes no arguments, but that is up to you if you need that. The functions could be named different (Base(Arg) and Other(First, Args...) e.g.).

The First argument is required to get the 'One or more' behaviour. If you would omit it, Foo(Args...) would accept zero or more parameters.

## Variadic template

[https://en.wikipedia.org/wiki/Variadic\\_template](https://en.wikipedia.org/wiki/Variadic_template)

Prior to C++11, templates could only take a fixed number of arguments. C++11 allows template definitions to take an arbitrary number of arguments of any type.

```
template<typename... Types>
class Demo{
public:
    // sizeof...()将返回template parameter pack中的类型个数。
    Demo(Types... params){ cout<< "there is "<< sizeof... (Types) << " types" << endl; }
};

int main(){
    int a = 10;
    string b = "hi";
    Demo<> d1;
    Demo<int, string> d2(a, b);
    Demo<int, int, string, string> d3(a, a, b, b);
}
```

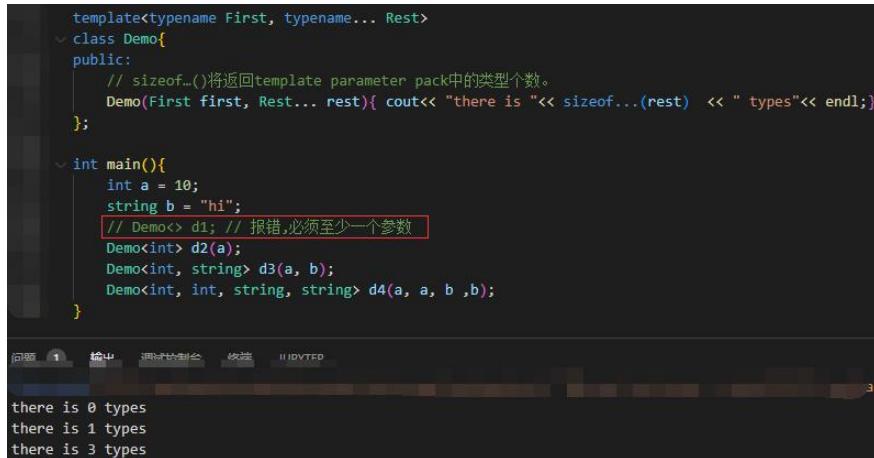
问题 1 检查 调试控制台 终端 JUPYTER

```
there is 0 types
there is 2 types
there is 4 types
```

- ◆ class... Ts 或 typename... Ts 表明 Ts 是一个模板参数包 template parameter pack。
- ◆ Ts... params 表示 params 是一个函数参数包。
- ◆ sizeof...() 将返回参数包中参数个数。

**Variadic templates may also apply to functions. If the variadic template should only allow a**

**positive number of arguments, then this definition can be used:**



```
template<typename First, typename... Rest> class tuple; // takes one or more arguments

template<typename First, typename... Rest>
class Demo{
public:
    // sizeof...()将返回template parameter pack中的类型个数。
    Demo(First first, Rest... rest){ cout<< "there is "<< sizeof...(rest) << " types"\n; }
};

int main(){
    int a = 10;
    string b = "hi";
    // Demo<> d1; // 报错,必须至少一个参数
    Demo<int> d2(a);
    Demo<int, string> d3(a, b);
    Demo<int, int, string, string> d4(a, a, b ,b);
}
```

The screenshot shows a C++ code editor with the following code. The code defines a variadic template class `tuple` and a class `Demo` that prints the size of its parameter pack. It also includes a `main` function with several calls to `Demo` with different numbers of arguments. The output window shows three lines of text: "there is 0 types", "there is 1 types", and "there is 3 types".

The ellipsis (...) operator has two roles. When it occurs to the left of the name of a parameter, it declares a parameter pack. Using the parameter pack, the user can bind zero or more arguments to the variadic template parameters. When the ellipsis operator occurs to the right of a template or function call argument, it unpacks the parameter packs into separate arguments.

The use of variadic templates is often recursive.

This is a recursive template. Notice that the variadic template version of `my_printf` calls itself, or (in the event that `args...` is empty) calls the base case.

There is no simple mechanism to iterate over the values of the variadic template. However, there are several ways to translate the argument pack into a single argument that can be evaluated separately for each parameter. Usually this will rely on function overloading, or — if the function can simply pick one argument at a time — using a dumb expansion marker:

## C++元编程

元编程在编译时 (compile time) 计算出运行时 (runtime) 需要的常数、类型、代码的方法。C++ 的抽象机制主要有两种：面向对象编程和模板编程。为了实现面向对象编程，C++ 提供了类(class)。而在模板编程方面，C++ 提供了模板 (template)。**模板编程的应用主要有两种：泛型编程 (generic programming) 和元编程 (meta-programming)**。前者注重于通用概念的抽象，设计通用的类型或算法；而后者注重于设计模板推导时的选择和迭代，通过模板技巧设计程序。**1995 年的 Todd Veldhuizen 在 C++ Report 上，首次提出了 C++ 模板元编程的概念，并指出了其在数值计算上的应用前景。**

目前最新的 C++ 将模板分成了 4 类：

- 类模板 (class template)，函数模板 (function template)。类模板 和 函数模板 分别用于定义具有相似功能的 类 和 函数 (function)，是泛型中对 类型 和 算法 的抽象。
- 别名模板 (alias template) ， 变量模板 (variable template)。分别在 C++ 11 和 C++ 14 引入，分别提供了具有模板特性的 类型别名 (type alias) 和 常量 (constant) 的简记方法。前者 类模板的嵌套类 等方法实现，后者则可以通过 constexpr 函数、类模板的静态成员、函数模板的返回值 等方法实现。例如，C++ 14 中的 别名模板 std::enable\_if\_t<T> 等价于 typename std::enable\_if<T>::type，C++ 17 中的 变量模板 std::is\_same<T, U> 等价于 std::is\_same<T, U>::value。尽管这两类模板不是必须的，但一方面可以增加程序的可读性（§ 4.1），另一方面可以提高模板的编译性能（§ 4.4）。

浅谈 C++ 元编程

<https://zhuanlan.zhihu.com/p/87917516>

c++模板编程

简单的 C++ 结构体字段反射

## 第三方 c++ 库

C++ 有没有像 pip、npm、gem 一样的包管理工具？

c++ 如何使用别人写好的库？也要用 include 命令把库文件包含进来吗？

### nullptr

看你用的库的类型。如果是 Header Only 的库，全都是头文件，典型的都是由.hpp 文件构成的(可能有的人用.h，但一定没有.cpp)，那你就只需要 include 进来相关的头文件即可，一般这种库，会有一个总的入口头文件，include 这一个就行，例如 boost 库中的 spirit 库就是这样的，只需要包含相关路径就行，至于用绝对路径还是相对路径，就看你的项目配置了。

如果是其他常规的库，也就是至少有一个.cpp 文件的，一般都需要把这个库编译为静态库或者动态库，然后在项目配置、或者 CMakeList 中配置好相应的库文件路径依赖，之后再 include 入口头文件。

### Michael Li

库文件并不需要 include 进来，但是在链接步骤需要指定。第三方源代码大致分为几种情况：

- ◆ 第三方库有源代码，但并未提供二进制库文件。比如 jsoncpp 这种轻巧的库，对于这种就需要和自己的源代码一起进行编译。需要在引用第三方库功能的源代码头部 include 第三方代码的头文件，并将源代码的实现部分放置在自己的源代码文件夹中。构建时让 C++ 编译器一起编译成二进制代码，再链接到一起。
- ◆ 第三方库代码庞大或者仅提供头文件和编译好的二进制文件。对于像 openssl、ffmpeg 等等大块头的开源项目，即使是拿到了源代码也不建议将其与自己的源代码合并编译。因为他们本身就很复杂且搭建编译环境也要费一番功夫。一般这些项目都提供已经编译好的二进制库文件。这些项目提供的已经编译好的库文件.so .o .lib .dll 等与对应的头文件，配合使用添加到自己的源代码项目中。使用时，将第三方头文件 include 到使用这些第三方库的源代码中，然后在链接环节指定链接器链接这些已经编译好的二进制库文件。

## MySQL Connector/C++ 8.0 Developer Guide

<https://dev.mysql.com/doc/connector-cpp/8.0/en/>

看看目前我们使用的 mysql 版本:

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.7.38   |
+-----+
1 row in set (0.00 sec)
```

看看我们目前的 OS:

```
(base) [root@bigdata-dev01-e9cd wujianjun]# cat /proc/version
Linux version 3.10.0-1160.41.1.el7.x86_64 (mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623 (Red Hat 4.8.5-44) (GCC) ) #1 SMP Tue Aug 31 14:52:47 UTC 2021
(base) [root@bigdata-dev01-e9cd wujianjun]# cat /etc/os-release
NAME="CentOS Linux"
VERSION="7 (Core)"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="7"
PRETTY_NAME="CentOS Linux 7 (Core)"
```

Version 8.0 of Connector/C++ implements three different APIs which can be used by applications:

- ◆ The new **X DevAPI for applications written in C++**.
- ◆ The new **X DevAPI for C** for applications written in plain C.
- ◆ The **legacy JDBC4-based API** also implemented in version 1.1 of the connector.

Connector/C++ applications that use X DevAPI or X DevAPI for C require a MySQL server that has X Plugin enabled. Connector/C++ applications that use the legacy JDBC-based API neither require nor support X Plugin. MySQL 5.7 发布时自带 MySQL X 插件，这个插件是默认加载的，`show plugins` 可以看到。

```
mysql>SHOW PLUGINS;
+-----+-----+-----+-----+
| Name      | Status  | Type   | Library | License
+-----+-----+-----+-----+
| mysqlx      | ACTIVE  | Daemon | NULL    | GPL
| mysqlx_cache_cleaner | ACTIVE  | RUMIT  | NULL    | GPL
+-----+-----+-----+-----+
```

X-plugin 使用单独的协议(X protocol)来实现与客户端的交互，这个新协议利用了 protobuf。

我们决定使用 JDBC 接口，因为 X DevAPI 看起来更复杂，且我们的 mysql 不支持 X-plugin。

Connector/C++ implements the JDBC 4.0 API。The JDBC 4.0 API defines approximately 450 methods for the classes just mentioned. Connector/C++ implements approximately 80% of these.

### Installation on Linux

RPM packages are available for Linux. The packages are distinguished by their base names:

- ◆ `mysql-connector-c++`: This package provides the shared connector library implementing X DevAPI and X DevAPI for C.
- ◆ `mysql-connector-c++-jdbc`: This package provides the shared legacy connector library implementing the JDBC API.
- ◆ `mysql-connector-c++-devel`: This package installs development files required for building applications that use Connector/C++ libraries provided by the other packages, and static connector libraries. It cannot be installed by itself without the other two packages.

```
(base) [root@bigdata-dev01-e9cd wujianjun]# yum search mysql-connector
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
 * base: mirrors.bupt.edu.cn
 * centos: mirrors.huaweicloud.com
 * centos-sclo-scl0: mirrors.bupt.edu.cn
 * epel: mirrors.bfsu.edu.cn
 * extras: mirrors.huaweicloud.com
 * updates: mirrors.bfsu.edu.cn
=====
N/S matched: mysql-connector =====
mysql-connector-c++.x86_64 : MySQL database connector for C++
mysql-connector-c++-devel.x86_64 : Development header files and libraries for MySQL C++ client applications
mysql-connector-c++-jdbc.x86_64 : MySQL Driver for C++ which mimics the JDBC 4.0 API
mysql-connector-java.search : Standardized MySQL database driver for Java
```

于是，我们 `yum install` 这三个包。

## Building Connector/C++ Applications

The API an application uses determines which Connector/C++ header files it should include. we assume that `$MYSQL_CPPCONN_DIR` is the Connector/C++ installation location. Pass an `-I $MYSQL_CPPCONN_DIR/include` option on the compiler invocation command to ensure this.

```
(base) [root@bigdata-dev01-e9cd wujianjun]# rpm -qa | grep mysql-connector
mysql-connector-c++-devel-8.0.29-1.el7.x86_64
mysql-connector-c++-jdbc-8.0.29-1.el7.x86_64
mysql-connector-c++-8.0.29-1.el7.x86_64
(base) [root@bigdata-dev01-e9cd wujianjun]# rpm -ql mysql-connector-c++-devel-8.0.29-1.el7.x86_64
/usr/include/mysql-cppconn
/usr/include/mysql-cppconn-8
/usr/include/mysql-cppconn-8/jdbc
/usr/include/mysql-cppconn-8/jdbc/cppconn
/usr/include/mysql-cppconn-8/jdbc/cppconn/build_config.h
```

For applications that use the legacy JDBC API, the header files are version dependent:

As of Connector/C++ 8.0.16, a single #include directive suffices:

```
#include <mysql/jdbc.h>
```

看看是否有所说的头文件:

```
(base) [root@bigdata-dev01-e9cd mysql-cppconn-8]# pwd
/usr/include/mysql-cppconn-8
(base) [root@bigdata-dev01-e9cd mysql-cppconn-8]# cd mysql
(base) [root@bigdata-dev01-e9cd mysql]# ls
jdbc.h
```

## Link Libraries

我们使用动态链接库。

The Connector/C++ dynamic library name depends on the platform. These libraries implement X DevAPI and X DevAPI for C, where A in the library name represents the ABI version:

- ◆ libmysqlcppconn8.so.A (Unix)

For the legacy JDBC API, the dynamic libraries are named as follows, where B in the library name represents the ABI version:

- ◆ libmysqlcppconn.so.B (Unix)

```
(base) [root@bigdata-dev01-e9cd wujianjun]# rpm -ql mysql-connector-c++-devel-8.0.29-1.el7.x86_64 | grep -v /usr/include/mysql-cppconn
/usr/lib64/libmysqlcppconn-static.a
/usr/lib64/libmysqlcppconn.so
/usr/lib64/libmysqlcppconn8-static.a
/usr/lib64/libmysqlcppconn8.so
```

To build code that uses X DevAPI or X DevAPI for C, add `-lmysqlcppconn8` to the linker options. To build code that uses the legacy JDBC API, add `-lmysqlcppconn`.

## MySQL Connector/C++ 1.1 Developer Guide

<https://dev.mysql.com/doc/connector-cpp/1.1/en/>

<https://cpp.hotexamples.com/zh/examples/-/ResultSet/getInt/cpp-resultset-getint-method-examples.html>

<https://dev.mysql.com/doc/connectors/en/connector-j-usagenotes-last-insert-id.html>

示例如下：

```

#include <iostream>
#include <string>
#include <mysqlcppconn/mysql/jdbc.h>
using namespace std;

void select_mysql()
{
    // step1. 建立连接
    sql::mysql::MySQL_Driver *driver;
    sql::Connection *conn;
    driver = sql::mysql::get_mysql_driver_instance();
    conn = driver->connect("tcp://10.151.40.62:3306", "root", "DM@zal123456#");

    // step2. 执行查询
    string sql = "select memberId,true_age from dml.za_recall_users_v2 limit 10";
    sql::Statement *stmt = conn->createStatement();
    sql::ResultSet *res = stmt->executeQuery(sql.c_str());
    while (res->next()) {
        // getString("xxxx")
        // getInt("xxxx")
        // getInt64("xxxx")
        // getUInt("xxxx")
        cout << "memberId = " << res->getInt64("memberId") << ",true_age=" << res->getInt("true_age") << endl;
    }
    delete stmt;
    delete res;

    // step3. 关闭连接
    delete conn;
}

void insert_mysql()
{
    // step1. 建立连接
    sql::mysql::MySQL_Driver *driver;
    sql::Connection *conn;
    driver = sql::mysql::get_mysql_driver_instance();
    conn = driver->connect("tcp://10.151.40.62:3306", "root", "DM@zal123456#");

    // step2. 执行更新
    string sql = "insert into tarec.profile_cachefile_raw_info(profile_name,profile_version,profile_path,developer) \
                  values('za_user_profile','0.1','/home/ly/pz/20220729/za_user_profile','jianjun.wu')";
    sql::Statement *stmt = conn->createStatement();
    stmt->executeUpdate(sql.c_str());
    delete stmt;

    // step3. 关闭连接
    delete conn;
}

int main()
{
    // select_mysql();
    insert_mysql();
}
// g++ t.cpp -lmysqlcppconn -o t

```

运行结果为：

```
(base) [root@bigdata-dev01-e9cd wujianjun]# g++ -std=c++11 -lmysqlcppconn t.cpp -o t
(base) [root@bigdata-dev01-e9cd wujianjun]# ./t
memberId = 414044,true_age=53
memberId = 927010,true_age=49
memberId = 1067708,true_age=54
memberId = 776716,true_age=44
memberId = 2984791,true_age=42
memberId = 1564517,true_age=48
memberId = 584015,true_age=49
memberId = 2229940,true_age=46
memberId = 1316599,true_age=68
memberId = 1625149,true_age=62
finished!!!!
```

注意，在 cmake 中使用时，需要加入下面红色的部分：

```
target_link_libraries(${so_name} PRIVATE Threads::Threads mysqlcppconn)
```

## Dynamically load a function from a DLL

I'm having a little look at .dll files, I understand their usage and I'm trying to understand how to use them. I have created a .dll file that contains a function that returns an integer named func() using this code, I (think) I've imported the .dll file into the project (there's no complaints):

```
#include <windows.h>
#include <iostream>

int main() {
    HINSTANCE hGetProcIDDLL = LoadLibrary("C:\\Documents and Settings\\User\\Desktop\\fgfdg\\dgdg\\test.dll");

    if (hGetProcIDDLL == NULL) {
        std::cout << "cannot locate the .dll file" << std::endl;
    } else {
        std::cout << "it has been called" << std::endl;
        return -1;
    }

    int a = func();
}

return a;
}
```

Leniency

`LoadLibrary` does not do what you think it does. It loads the DLL into the memory of the current process, but it does not magically import functions defined in it! This wouldn't be possible, as function calls are resolved by the linker at compile time while `LoadLibrary` is called at runtime.

You need a separate WinAPI function to get the address of dynamically loaded functions: `GetProcAddress`.

c++ 小库

## cxxopts

cxxopts is a lightweight C++ option parser library. Options can be given as:

```
--long=argument  
-long argument
```

基本用法如下：

```
#include <cxxopts.hpp>

// Create a cxxopts::Options instance
cxxopts::Options options("MyProgram", "One line description of MyProgram");

// Then use add_options.
options.add_options()
    ("d,debug", "Enable debugging") // a bool parameter
    ("i,integer", "Int param", cxxopts::value<int>())
    ("f,file", "File name", cxxopts::value<std::string>())
    ("v,verbose", "Verbose output", cxxopts::value<bool>() -> default_value("false"))
;

/*
Options are declared with a long and an optional short option.
The third argument is the value, if omitted it is boolean.
*/

// To parse the command line do:
auto result = options.parse(argc, argv);

// to get its value
result["opt"].as<type>()
```

### fast\_double\_parser: 4x faster than strtod

[https://github.com/lemire/fast\\_double\\_parser](https://github.com/lemire/fast_double_parser)

We do not sacrifice accuracy. The function will match exactly the result of a standard function like strtod. We require C++11.

You should be able to just drop the header file into your project, it is a header-only library.

The current API is simple enough:

```
#include "fast_double_parser.h" // the file is in the include directory

double x;
char * string = ...
const char * endptr = fast_double_parser::parse_number(string, &x);
```

You must check the value returned (`endptr`): if it is `nullptr`, then the function refused to parse the input. Otherwise, we return a pointer (`const char *`) to the end of the parsed string. *The provided pointer (string) should point at the beginning of the number: if you must skip whitespace characters, it is your responsibility to do so.* the parser will reject overly large values that would not fit in `binary64`. It will not accept `NaN` or infinite values.

我们对比测试一下：

The screenshot shows a terminal session on a CentOS VM. The user first lists files in the current directory, then cat'ing the source code of t.cpp which includes the fast\_double\_parser library. The code then benchmarks two parsing methods: one using chrono::duration\_cast<chrono::microseconds> and another using std::stof. The results show that the fast\_double\_parser method is significantly faster, taking approximately 564 microseconds compared to 1798 microseconds for the standard method.

```
(base) [root@VM-40-14-centos wujianjun]# ls
data  fast_double_parser  src  t  t.cpp  zacpp  zacpp.zip
(base) [root@VM-40-14-centos wujianjun]# cat t.cpp
#include <iostream>
#include <string>
#include <chrono>
using namespace std;
#include "fast_double_parser/include/fast_double_parser.h"

int main()
{
    auto start1 = chrono::system_clock::now(); // 结束时间
    for(int i = 0; i < 10000; i++)
    {
        string str = "4665.2656626e2";
        double x;
        const char * endptr = fast_double_parser::parse_number(str.c_str(), &x);
    }
    auto endl = chrono::system_clock::now(); // 结束时间
    auto duration1 = chrono::duration_cast<chrono::microseconds>(endl - start1);
    cout << "总耗时1:" << duration1.count() << " 微秒" << endl;

    auto start2 = chrono::system_clock::now(); // 结束时间
    for(int i = 0; i < 10000; i++)
    {
        string str = "4665.2656626e2";
        double x = stof(str);
    }
    auto end2 = chrono::system_clock::now(); // 结束时间
    auto duration2 = chrono::duration_cast<chrono::microseconds>(end2 - start2);
    cout << "总耗时2:" << duration2.count() << " 微秒" << endl;
}
(base) [root@VM-40-14-centos wujianjun]# g++ t.cpp -o t -O3
(base) [root@VM-40-14-centos wujianjun]# ./t
总耗时1:564 微秒
总耗时2:1798 微秒
(base) [root@VM-40-14-centos wujianjun]# ./t
总耗时1:701 微秒
总耗时2:1810 微秒
(base) [root@VM-40-14-centos wujianjun]#
```

可以看到 3 倍左右。

The `fast_float` offers an API resembling that of the C++17 `std::from_chars` functions. In particular, you can specify the beginning and the end of the string. Furthermore `fast_float` supports both 32-bit and 64-bit floating-point numbers. The `fast_float` library is part of Apache Arrow.

We encourage users to adopt `fast_float` library instead. It has more functionality and greater speed in some cases.

## fast\_float number parsing library: 4x faster than strtod

[https://github.com/lemire/fast\\_float](https://github.com/lemire/fast_float)

The *fast\_float* library provides fast header-only implementations for the C++ *from\_chars* functions for *float* and *double* types.

*fast\_float* provides the following two functions(the library itself requires C++11):

```
from_chars_result from_chars(const char* first, const char* last, float& value, ...);  
from_chars_result from_chars(const char* first, const char* last, double& value, ...);
```

The return type (*from\_chars\_result*) is defined as the struct:

```
struct from_chars_result {  
    const char* ptr;  
    std::errc ec;  
};
```

It parses the character sequence [first,last) for a number.

Given a successful parse, the pointer (ptr) in the returned value is set to point right after the parsed number, and the value referenced is set to the parsed value.

The implementation does not throw and does not allocate memory (e.g., with new or malloc).

**It will parse infinity and nan values.**

we have the following restrictions:

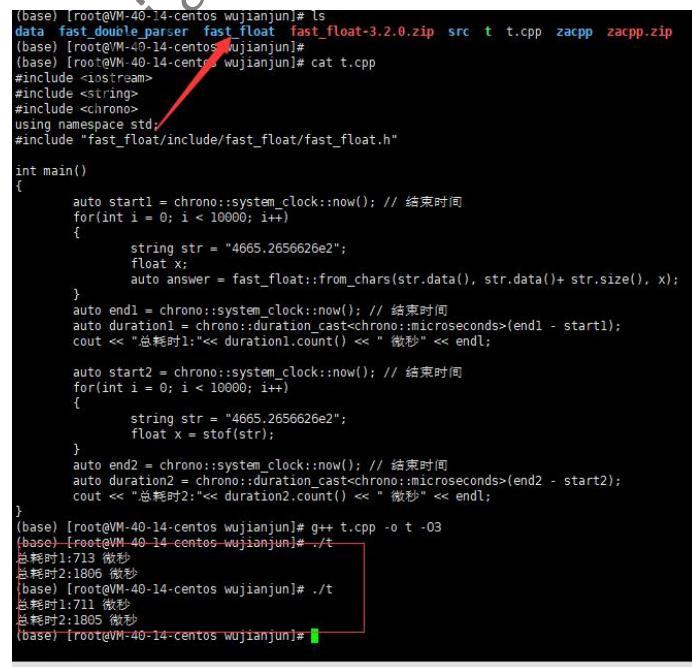
- ◆ The *from\_chars* function does not skip leading white-space characters.
- ◆ A leading + sign is forbidden.
- ◆ We only support the decimal format: we do not support hexadecimal strings.



```
(base) [root@VM-40-14-centos wujianjun]# ls  
(base) [root@VM-40-14-centos wujianjun]# data fast_double_parser fast_float src t.cpp zacpp zacpp.zip  
(base) [root@VM-40-14-centos wujianjun]# vi t.cpp  
(base) [root@VM-40-14-centos wujianjun]# ls  
data fast_double_parser fast_float src t.cpp zacpp zacpp.zip  
(base) [root@VM-40-14-centos wujianjun]# g++ t.cpp -o t -O3  
(base) [root@VM-40-14-centos wujianjun]# ./t  
65.2  
0  
65.2  
0.062  
(base) [root@VM-40-14-centos wujianjun]#
```

```
184 #include <chrono>  
185 using namespace std;  
186 #include "fast_float/include/fast_float/fast_float.h"  
187  
188 float strf(string str)  
189 {  
190     float x = 0;  
191     auto answer = fast_float::from_chars(str.data(), str.data() + str.size(), x);  
192     return x;  
193 }  
194  
195 int main()  
196 {  
197     cout << str2f("65.2") << endl;  
198     cout << str2f(" 65.2") << endl; // 前面有空格字符将报错  
199     cout << str2f("65.2 ") << endl;  
200     cout << str2f("6.2e-2") << endl; // 支持科学计数法
```

对比一下性能:



```
(base) [root@VM-40-14-centos wujianjun]# ls  
data fast_double_parser fast_float fast_float-3.2.0.zip src t t.cpp zacpp zacpp.zip  
(base) [root@VM-40-14-centos wujianjun]# (base) [root@VM-40-14-centos wujianjun]# cat t.cpp  
#include <iostream>  
#include <string>  
#include <chrono>  
using namespace std;  
#include "fast_float/include/fast_float/fast_float.h"  
  
int main()  
{  
    auto start1 = chrono::system_clock::now(); // 结束时间  
    for(int i = 0; i < 10000; i++)  
    {  
        string str = "4665.2656626e2";  
        float x;  
        auto end1 = chrono::system_clock::now(); // 结束时间  
        auto duration1 = chrono::duration_cast<chrono::microseconds>(end1 - start1);  
        cout << "总耗时1:" << duration1.count() << " 微秒" << endl;  
  
        auto start2 = chrono::system_clock::now(); // 结束时间  
        for(int i = 0; i < 10000; i++)  
        {  
            string str = "4665.2656626e2";  
            float x = stof(str);  
        }  
        auto end2 = chrono::system_clock::now(); // 结束时间  
        auto duration2 = chrono::duration_cast<chrono::microseconds>(end2 - start2);  
        cout << "总耗时2:" << duration2.count() << " 微秒" << endl;  
    }  
(base) [root@VM-40-14-centos wujianjun]# g++ t.cpp -o t -O3  
(base) [root@VM-40-14-centos wujianjun]# ./t  
总耗时1:713 微秒  
总耗时2:1806 微秒  
(base) [root@VM-40-14-centos wujianjun]# ./t  
总耗时1:711 微秒  
总耗时2:1805 微秒  
(base) [root@VM-40-14-centos wujianjun]#
```

快 2.6 倍。

## Benchmark of major hash maps implementations

<https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html>

This benchmark compares different C++ implementations of hashmaps. The main contestants are

- ◆ `tsl::hopscotch_map`,
- ◆ `tsl::robin_map`,
- ◆ `std::unordered_map`,
- ◆ `google::dense_hash_map`

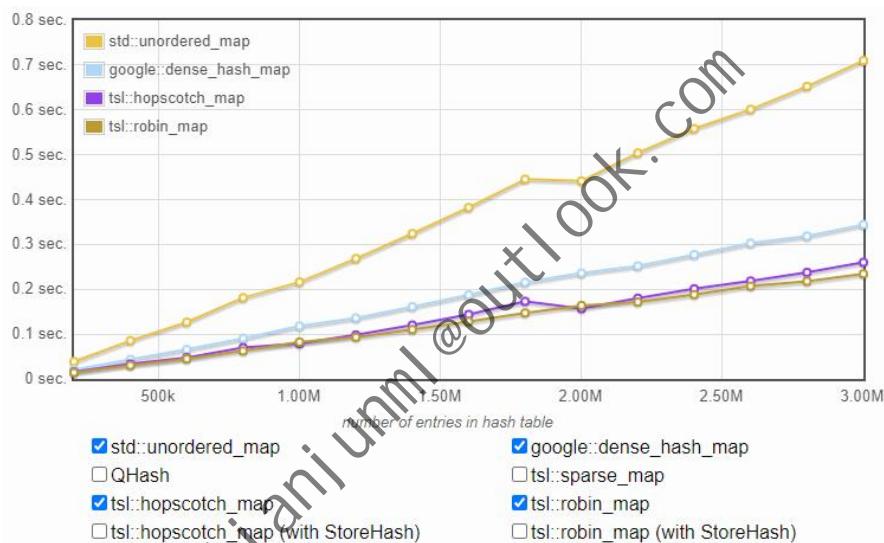
We will see how they perform in a large range of operations, both in terms of speed and memory usage.

### Benchmark

For the small string tests, we use hash maps with `std::string` as key and `int64_t` as value.

The size of each `std::string` object is 32 bytes on the used compiler.

Before the test, we insert `nb_entries` elements in the hash map as in the inserts test. We then read each key-value pair in a different and random order than the one they were inserted.



### Which hash map should I choose?

Before choosing a hash map, just try out `std::unordered_map`. Even though it is not the fastest hash map out there due to the **cache-unfriendliness of chaining**, the standard hash map just works well in most cases.

For speed efficiency. A hash map using an open addressing scheme should be your choice and I would recommend either hopscotch hashing with `tsl::hopscotch_map`.

## A C++ implementation of a fast hash map and hash set using hopscotch hashing

<https://github.com/Tessil/hopscotch-map>

The hopscotch-map library is a C++ implementation of a fast hash map and hash set using open-addressing and hopscotch hashing to resolve collisions. It is a **cache-friendly** data structure offering better performances than std::unordered\_map in most cases and is closely similar to google::dense\_hash\_map while using less memory and providing more functionalities.

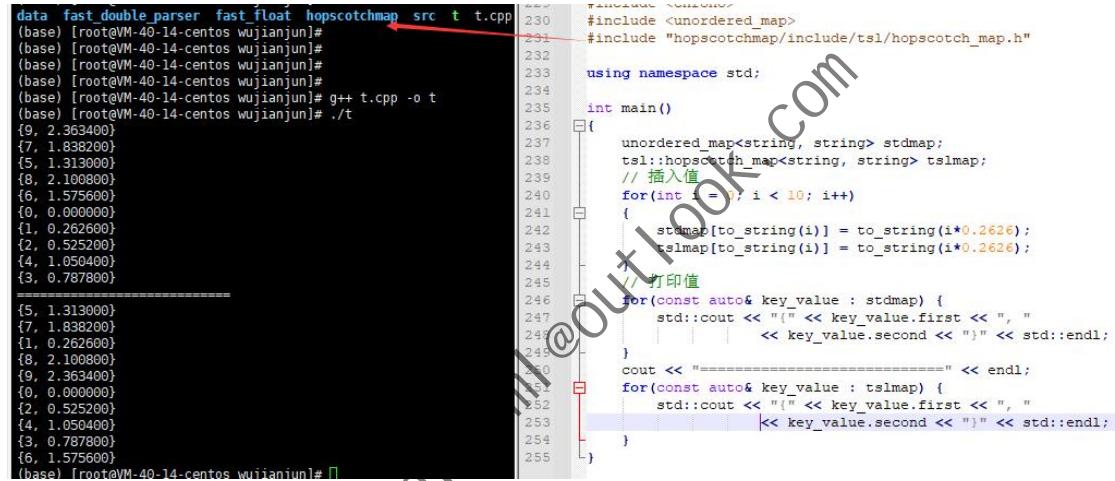
### Key features

- ◆ **Header-only library**, just add the include directory to your include path and you are ready to go. If you use CMake, you can also use the tsl::hopscotch\_map exported target from the CMakeLists.txt.
- ◆ API closely similar to std::unordered\_map and std::unordered\_set.

### Differences compared to std::unordered\_map

We tries to have an interface similar to std::unordered\_map, but some differences exist.

- ◆ In general any operation modifying the hash table, except erase, invalidate all the iterators .



```
data fast_double_parser fast_float hopscotchmap src t t.cpp
(base) [root@VM-40-14-centos wujianjun]# ./t
(base) [root@VM-40-14-centos wujianjun]# ./t
(base) [root@VM-40-14-centos wujianjun]# ./t
(base) [root@VM-40-14-centos wujianjun]# ./t
(base) [root@VM-40-14-centos wujianjun]# g++ t.cpp -o t
(base) [root@VM-40-14-centos wujianjun]# ./t
(base) [root@VM-40-14-centos wujianjun]#
[9, 2.363400]
[7, 1.838200]
[5, 1.313000]
[8, 2.100800]
[6, 1.575600]
[0, 0.000000]
[1, 0.262600]
[2, 0.525200]
[4, 1.050400]
[3, 0.787800]
=====
[5, 1.313000]
[7, 1.838200]
[1, 0.262600]
[8, 2.100800]
[9, 2.363400]
[0, 0.000000]
[2, 0.525200]
[4, 1.050400]
[3, 0.787800]
[6, 1.575600]
(base) [root@VM-40-14-centos wujianjun]#
```

```
#include <unordered_map>
#include "hopscotchmap/include/tsl/hopscotch_map.h"

using namespace std;

int main()
{
    unordered_map<string, string> stdmap;
    tsl::hopscotch_map<string, string> tsmap;
    // 插入值
    for(int i = 0; i < 10; i++) {
        stdmap[to_string(i)] = to_string(i*0.2626);
        tsmap[to_string(i)] = to_string(i*0.2626);
    }
    // 打印值
    for(const auto& key_value : stdmap) {
        std::cout << "(" << key_value.first << ", "
                           << key_value.second << ")" << std::endl;
    }
    cout << "======" << endl;
    for(const auto& key_value : tsmap) {
        std::cout << "(" << key_value.first << ", "
                           << key_value.second << ")" << std::endl;
    }
}
```

我们对比一下性能

### Installation

To use hopscotch-map, just add the include directory to your include path. It is a header-only library.

If you use CMake, you can also use the tsl::hopscotch\_map exported target from the CMakeLists.txt with target\_link\_libraries.

# Example where the hopscotch-map project is stored in a third-party directory

add\_subdirectory(third-party/hopscotch-map)

target\_link\_libraries(your\_target PRIVATE tsl::hopscotch\_map)

# 多线程与多核编程

## C++多线程并发基础入门教程

运行越多的线程，操作系统需要为每个线程分配独立的栈空间，需要越多的上下文切换。

### C++多线程并发基础知识

The screenshot shows a code editor with C++ code. The code defines a class `Para` with a constructor and a member function `proc`. The `proc` function takes a reference to a `Para` object and iterates through its array `a`, setting elements based on their index mod 2. The main function creates two threads, `th1` and `th2`, each running the `proc` function on different `Para` objects. The output window shows the final state of the arrays `a` for both threads.

```
#include<iostream>
#include<thread> // 线程的头文件,包括std::thread
using namespace std;

class Para
{
public:
    Para(){}
    Para(int* a, int len, bool mod)
    { this->a = a; this->len = len; this->mod = mod; }
    Para(const Para& para)
    { this->a = para.a; this->len = para.len; this->mod = para.mod; }
public:
    int* a; int len; bool mod;
};

void proc(Para& para)
{
    for(int i=0; i< para.len ; i++){
        if(para.mod){
            if(i % 2 == 0){
                para.a[i] = 1;
            }
        } else{
            if(i % 2 != 0){
                para.a[i] = 0;
            }
        }
    }
}

1391 int main()
1392 {
1393     int a [] = {1,2,3,4,5,6,7,8,9};
1394     int len = 9;
1395     Para para1(a , len , true);
1396     Para para2(a , len , false);
1397     // 创建并运行线程
1398     // 第一个参数为函数名，第二个参数为该函数的第一个参数，如果该函数接收多个参数就依次写在后面。
1399     thread th1(proc, ref(para1));
1400     thread th2(proc, ref(para2));
1401     cout << "主线程中显示子线程id为" << th1.get_id() << endl;
1402     cout << "主线程中显示子线程id为" << th2.get_id() << endl;
1403     th1.join(); //此时主线程被阻塞直至子线程执行结束。
1404     th2.join(); //此时主线程被阻塞直至子线程执行结束。
1405     for(int i=0; i< len ; i++){
1406         cout<< a[i] << " ";
1407     }
1408     cout<<endl;
1409     return 0;
1410 }
1411 }
```

问题 ② 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zancode\zacpp\zcpp\src\" && g++ test.cpp -o test && "d:\wujianjun\code\my\zancode\zacpp\zcpp\src\test"  
主线程中显示子线程id为2  
主线程中显示子线程id为3  
1 0 1 0 1 0 1 0 1

只要创建了线程对象，线程就开始执行。不应该在创建了线程后马上 `join`，这样会马上阻塞主线程。当线程启动后，一定要在和线程相关联的 `std::thread` 对象销毁前，对线程运用 `join()` 或者 `detach()` 方法。`join()` 与 `detach()` 的区别是是否等待子线程执行结束。调用 `join()` 会清理线程相关的存储部分，这代表了 `join()` 只能调用一次。使用 `joinable()` 来判断 `join()` 可否调用。同样，`detach()` 也只能调用一次，一旦 `detach()` 后就无法 `join()` 了。如果使用 `detach()`，就必须保证线程结束之前可访问数据的有效性。

如果用于创建线程的函数为含参函数，那么在创建线程时，要一并将函数的参数传入：

- ◆ 当传入参数为基本数据类型(`int`, `char`,`string` 等)时，会拷贝一份给创建的线程；
- ◆ 当传入参数为指针时，会浅拷贝一份给创建的线程。
- ◆ 当传入的参数为引用时，实参必须用 `ref()` 处理后传递给形参，此时不存在“拷贝”行为。

## 锁的使用

多线程编程时要考虑多个线程同时访问共享资源所造成的问题，因此可以通过加锁解锁来保证同一时刻只有一个线程能访问共享资源。使用锁的时候要注意可能出现的几种现象：

- ◆ 死锁：是指两个或两个以上的进程(或线程)在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。
- ◆ 活锁：任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。
- ◆ 锁饥饿：是指如果线程 T1 占用了资源 R，线程 T2 又请求封锁 R，于是 T2 等待。T3 也请求资源 R，当 T1 释放了 R 上的封锁后，系统首先批准了 T3 的请求，T2 仍然等待。然后 T4 又请求封锁 R，当 T3 释放了 R 上的封锁之后，系统又批准了 T4 的请求...，T2 可能永远等待。

同步与互斥的区别为：

- ◆ 同步指维护任务片段的先后顺序；直观的表现就是若 A 片段执行完才能执行 B 片段，线程 1 执行 A 片段，线程 2 执行 B 片段，在 B 片段执行前申请锁 l，在 A 片段执行结束后解锁 l；未申请到锁 l 即 A 片段还未执行完，线程 1 等待线程 2 执行。
- ◆ 互斥就是保证资源同一时刻只能被一个进程使用；互斥是为了保证数据的一致性，如果 A 线程在执行计算式 A 的时候，某个量被 B 线程改掉了，这可能会出现问题，于是要求资源互斥。

C++ 中定义很多种锁：

- 互斥锁：C++11 引入 mutex(不能递归使用)，加锁的资源支持互斥访问；

```
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex

volatile int counter(0); // non-atomic counter

// smutex 不允许拷贝构造，也不允许 move 拷贝，最初产生的 mutex 对象是处于 unlocked 状态的。
std::mutex mtx;

void attempt_10k_increases()
{
    for (int i=0; i<10000; ++i)
    {
        //lock(), 调用线程将锁住该互斥量。
        // 1. 如果该互斥量当前没有被锁住，则调用线程将该互斥量锁住，直到调用 unlock 之前，该线程一直拥有该锁。
        // 2. 如果当前互斥量被其他线程锁住，则当前的调用线程被阻塞住。
        // 3. 如果当前互斥量被当前调用线程锁住，则会产生死锁(deadlock)。
        // try_lock, 尝试锁住互斥量。
        // 1. 如果当前互斥量没有被其他线程占有，则该线程锁住互斥量，直到该线程调用 unlock 释放互斥量。
        // 2. 如果当前互斥量被其他线程锁住，则当前调用线程返回 false，而并不会被阻塞掉。
        // 3. 如果当前互斥量被当前调用线程锁住，则会产生死锁(deadlock)。
        if (mtx.try_lock()) // 尝试锁住互斥量,
        {
            ++counter;
            mtx.unlock(); // 解锁
        }
    }
}

int main (int argc, const char* argv[])
{
    std::thread threads[10];
    for (int i=0; i<10; ++i)
    {
        threads[i] = std::thread(attempt_10k_increases);
    }

    for (auto& th : threads)
    {
        th.join();
    }
    std::cout << counter << " successful increases of the counter.\n";
    return 0;
}
```

不推荐直接去调用成员函数 lock()，因为如果忘记 unlock()，将导致锁无法释放，使用 lock\_guard 或者 unique\_lock 则能避免忘记解锁带来的问题。

- ◆ 读写锁: C++14 提供了 `shared_mutex`, 也就是读写锁, 读写锁同时只能有一个写者或多个读者, 但不能同时既有读者又有写者。`shared_mutex` 通过 `lock_shared`, `unlock_shared` 进行读者的锁定与解锁; 通过 `lock`, `unlock` 进行写者的锁定与解锁。

```

shared_mutex s_m;

std::string book;

void read()
{
    s_m.lock_shared();
    cout << book;
    s_m.unlock_shared();
}

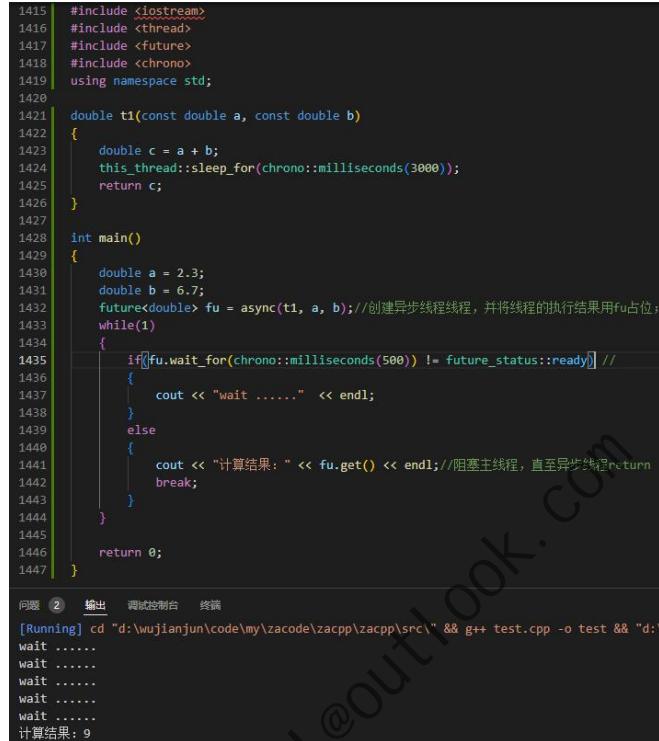
void write()
{
    s_m.lock();
    book = "new context";
    s_m.unlock();
}

```

- ◆ 自旋锁: 自旋锁在获取锁失败的时候不会使得线程阻塞而是一直自旋尝试获取锁, 此时 CPU 不能做其他事情, 而是一直处于轮询忙等的状态。自旋锁主要适用于被持有时间短, 线程不希望在重新调度上花过多时间的情况。实际上多数互斥锁在试图获取锁的时候会先自旋一小段时间, 然后才会休眠。
- ◆ 条件变量: C++11 引入了 `<condition_variable>`, 当 `std::condition_variable` 对象的某个 `wait` 函数被调用的时候, 它使用 `std::unique_lock`(通过 `std::mutex`) 来锁住当前线程。当前线程会一直被阻塞, 直到另外一个线程在相同的 `std::condition_variable` 对象上调用了 `notification` 函数来唤醒当前线程。
- ◆ 原子变量: 原子操作指“不可分割的操作”, 也就是说这种操作状态要么是完成的, 要么是没完成的, 不存在“操作完成了一半”这种状况。比如, 定义一个共享的变量(`int i=0`), 多个线程会用到这个变量, 那么每次操作这个变量时, 都需要 `lock` 加锁, 操作完毕 `unlock` 解锁, 这就很麻烦。现在实例化了一个类对象(`std::atomic<int> i=0`)来代替以前的那个变量, 每次操作它时, 就不用 `lock` 与 `unlock`, 这个对象自身就具有原子性。`std::atomic<>` 提供了常见的原子操作: `store` 是原子写操作, `load` 是原子读操作, `exchange` 是于两个数值进行交换的原子操作。即使使用了 `std::atomic<>`, 也要注意执行的操作是否支持原子性, 也就是说, 你对它进行的运算不支持原子性的话, 也不能实现其原子效果。

## 异步线程

c++11 中引入了<future>头文件，其中定义了异步线程。async 是一个函数模板，用来启动一个异步任务，它返回一个 future 类模板对象，**future 对象起到了占位的作用，刚实例化的 future 是没有储存值的，但在调用 future 对象的 get()成员函数时，主线程会被阻塞直到异步线程执行结束，并把返回结果传递给 std::future。**



The screenshot shows a code editor with a C++ file containing the following code:

```
1415 #include <iostream>
1416 #include <thread>
1417 #include <future>
1418 #include <chrono>
1419 using namespace std;
1420
1421 double t1(const double a, const double b)
1422 {
1423     double c = a + b;
1424     this_thread::sleep_for(chrono::milliseconds(3000));
1425     return c;
1426 }
1427
1428 int main()
1429 {
1430     double a = 2.3;
1431     double b = 6.7;
1432     future<double> fu = async(t1, a, b); //创建异步线程线程，并将线程的执行结果用fu占位
1433     while(1)
1434     {
1435         if(fu.wait_for(chrono::milliseconds(500)) != future_status::ready) //如果未完成
1436         {
1437             cout << "wait ....." << endl;
1438         }
1439         else
1440         {
1441             cout << "计算结果: " << fu.get() << endl; //阻塞主线程，直至异步线程return
1442             break;
1443         }
1444     }
1445
1446     return 0;
1447 }
```

Below the code editor is a terminal window showing the execution of the program:

```
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\src" && g++ test.cpp -o test && "d:\test"
wait .....
wait .....
wait .....
wait .....
wait .....
计算结果: 9
```

**std::future 的 get()成员函数是转移数据所有权;std::shared\_future 的 get()成员函数是复制数据。因此： future 对象的 get()只能调用一次；无法实现多个线程等待同一个异步线程，一旦其中一个线程获取了异步线程的返回值，其他线程就无法再次获取。std::shared\_future 对象的 get()可以调用多次；可以实现多个线程等待同一个异步线程，每个线程都可以获取异步线程的返回值。**

## 线程池

线程池所解决的问题：

- ◆ 需要频繁创建与销毁大量线程的情况下，由于线程预先就创建好了，接到任务就能马上从线程池中调用线程来处理任务，减少了创建与销毁线程带来的资源开销。
- ◆ 需要并发的任务很多时候，使用线程池可以将提交的任务挂在任务队列上，等到池中有空闲线程时就可以为该任务指定线程。

线程池是难点。

## 硬件与并发

### 磁盘多线程

- ◆ 全随机写无疑是最快的写入方式，测试中发现，将 200M 的内存数据随机的写入到 100G 的磁盘数据里面，竟然要 2 个小时之多。原因就是 200 万次随机写。可以简单的在内存中缓存一段时间，然后排序，使得在写盘的时候，磁头的移动只向一个方向。根据测试，这使得写盘的速度提高了 5 倍。
- ◆ 增加线程数，可以有效的提升程序整体的 io 处理速度(10 倍以上)，但同时，每个 io 请求的响应时间上升很多。

读线程数	读出100次耗时 ( um )	读平均相应时间 ( um )
1	1329574	13291
5	251765	12976
10	149206	15987
20	126755	25450
50	96595	48351

内核在处理 io 请求的时候，并不是简单的先到先处理，而是根据磁盘的特性，使用某种电梯算法，在处理完一个 io 请求后，会优先处理最临近的 io 请求。这样可以有效的减少磁盘的寻道时间。但对于每一个 io 请求来看，由于可能需要等待，所以响应时间会有所提升。

- ◆ 是否使用 direct io:
- ◆ 磁盘预读: Linux 只对顺序读(sequential read)进行预读。

## 多线程与 OOP

首先我们看下面的例子：

```
1535 void func(string data)
1536 {
1537     cout<<data<<endl;
1538 }
1539
1540 int main()
1541 {
1542
1543     string data1 = "1111";
1544     thread th1(func, data1);
1545
1546     string data2 = "2222";
1547     thread th2(func, data2);
1548
1549     th1.join();
1550     th2.join();
1551
1552     return 0;
1553 }
1554
1555 
```

问题 ③ 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zcode\zcode"  
1111  
2222

再看与 OOP 结合时的两种写法：

```
1508 class Task
1509 {
1510     public:
1511         Task(){}
1512         void thread_run()
1513         {
1514             string data1 = "1111";
1515             thread th1(func, data1);
1516
1517             string data2 = "2222";
1518             thread th2(func, data2);
1519
1520             th1.join();
1521             th2.join();
1522         }
1523     public:
1524         void func(string data)
1525         {
1526             cout<<data<<endl;
1527         }
1528     };
1529
1530     int main()
1531     {
1532         Task task;
1533         task.thread_run();
1534
1535         return 0;
1536     }
1537 
```

问题 ③ 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zcode\zcode"  
In file included from test.cpp:1506:  
C:/Program Files/LLVM/lib/gcc/x86\_64-w64-mingw32/6.3.0/include/c++/x86\_64-w64-mingw32/bits/stl\_function.h:115:26: required from here  
C:/Program Files/LLVM/lib/gcc/x86\_64-w64-mingw32/6.3.0/include/c++/x86\_64-w64-mingw32/bits/stl\_function.h:115:26: note: #include <functional>

```
1508 class Task
1509 {
1510     public:
1511         Task(){}
1512         void thread_run()
1513         {
1514             string data1 = "1111";
1515             thread th1(func, data1);
1516
1517             string data2 = "2222";
1518             thread th2(func, data2);
1519
1520             th1.join();
1521             th2.join();
1522         }
1523     public:
1524         // 必须加入static关键字
1525         static void func(string data)
1526         {
1527             cout<<data<<endl;
1528         }
1529     };
1530
1531     int main()
1532     {
1533         Task task;
1534         task.thread_run();
1535
1536         return 0;
1537     }
1538 
```

问题 ③ 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zcode\zcode"  
2222  
1111

可以看到左边报错了，所以如果线程要执行类的某个函数，那么必须是类的 static 函数。

- ◆ 类的对象用多线程执行调类的函数时，被执行函数必须是 static 的
- ◆ 当在多线程执行时，需要访问本对象的数据，则将自己的指针传给 static 函数

```

1510 class Task
1511 {
1512     public:
1513         Task(string name, int age)
1514     {
1515         this->name = name;
1516         this->age = age;
1517     }
1518     void thread_run()
1519     {
1520         // 类的对象用多线程执行调类的函数时，被执行函数必须是static的
1521         // 当在多线程执行时，需要访问本对象的数据，则将自己的指针传给static函数
1522         thread th1(func, this);
1523         thread th2(func, this);

1524         std::this_thread::sleep_for(std::chrono::milliseconds(3000));

1525         th1.join();
1526         th2.join();
1527     }
1528     public:
1529         static void func(Task *task)
1530     {
1531         cout<<task->name<<endl;
1532         cout<<task->age<<endl;
1533     }
1534     public:
1535         string name;
1536         int age;
1537     };
1538
1539
1540
1541 int main()
1542 {
1543     Task task("AAAAAA",10.0);
1544     task.thread_run();
1545
1546     return 0;
1547 }
```

问题 ③ 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zancode\zacpp\zacpp\src\" & g++ test.cpp -o test  
AAAAAA  
10  
AAAAAA  
10

再看一个例子：

```

1510 class Task;
1511 void func(Task *task, string& info);
1512
1513 class Task
1514 {
1515     public:
1516         Task(string name, int age){ this->name = name;this->age = age; }
1517         void thread_run(string& info);
1518     public:
1519         static void func(Task *task, string& info);
1520     public:
1521         string name;
1522         int age;
1523     };
1524
1525
1526 void Task::thread_run(string& info)
1527 {
1528     // 如果在本类外存在一个一样的函数 func，那么优先调用本类的
1529     // 必须使用ref包一下参数!!!!无论这个参数在线程函数中是值还是引用!!!!!!
1530     thread th1(func, this, ref(info));
1531     thread th2(func, this, ref(info));
1532     std::this_thread::sleep_for(std::chrono::milliseconds(3000));
1533     th1.join();
1534     th2.join();
1535 }
1536 void Task::func(Task *task, string& info)
1537 { cout<<"inner:"<<task->name<<","<<task->age<<","<<info<<endl; }
1538
1539 void func(Task *task, string& info)
1540 { cout<<"outer:"<<task->name<<","<<task->age<<","<<info<<endl; }

1541
1542 int main()
1543 {
1544     Task task("AAAAAA",10.0);
1545     string info = "two threads";
1546     task.thread_run(info);
1547
1548     return 0;
1549 }
```

问题 ① 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zancode\zacpp\zacpp\src\" & g++ test.cpp -o test  
inner:AAAAAA,10,two threads  
inner:AAAAAA,10,two threads

- ◆ 线程函数中的指针不能是数组名。

## 多线程的坑集

### C++ terminate called without an active exception

Eric Leschinski:

How to reproduce that error:

```
#include <iostream>
#include <stdlib.h>
#include <string>
#include <thread>
using namespace std;
void task1(std::string msg){
    cout << "task1 says: " << msg;
}
int main() {
    std::thread t1(task1, "hello");
    return 0;
}
```

Compile and run:

```
el@defiant ~/foo4/39_threading $ g++ -o s s.cpp -pthread -std=c++11
el@defiant ~/foo4/39_threading $ ./s
terminate called without an active exception
Aborted (core dumped)
```

You get that error because you didn't join or detach your thread.

passing a local variable to thread. is it possible?

我们为两个线程分配了任务，任务用局部变量表示，但是第二个线程却拿到了第一个线程的任务参数：

```
2344 void tfun(int start_index, int end_index){
2345     cout << "thrad_id:" << this_thread::get_id()
2346     << ",start_index=" << start_index << ",end_index=" << end_index << endl;
2347 }
2348
2349 void start_tasks(){
2350     thread threads[2];
2351     for(int i = 0; i < 2; i++){
2352         int start_index = i + 1;
2353         int end_index = start_index*100 ;
2354         cout << "start_index=" << start_index << ",end_index=" << end_index << endl;
2355         threads[i] = thread(tfun, ref(start_index), ref(end_index));
2356     }
2357     for(int i = 0; i < 2; i++){
2358         threads[i].join();
2359     }
2360 }
2361
2362 int main()
2363 {
2364     cout << "start ....." << endl;
2365     start_tasks();
2366     this_thread::sleep_for(chrono::seconds(1));
2367 }
```

问题 7    输出    调试控制台    终端    JUPYTER  
[Running] cd "d:\wujianjun\code\my\zancode\zacpp\zacpp\src\" && g++ test.cpp -o test && "d:\wujianjun\code\my\zancode\zacpp\zacpp\src\test"  
start .....  
start\_index=1,end\_index=100  
start\_index=2,end\_index=200  
thrad\_id:2,start\_index=2,end\_index=200  
thrad\_id:3,start\_index=2,end\_index=200

其实，我们不能从主线程向子线程传递一个栈上的变量，应该传递堆上的变量：

```
2344 class TaskPara
2345 {
2346     public:
2347         TaskPara(int start_index_, int end_index_):start_index(start_index_), end_index(end_index_){}
2348         int start_index; int end_index;
2349     };
2350     void tfun(TaskPara* para){
2351         cout << "thrad_id:" << this_thread::get_id()
2352         << ",start_index=" << para->start_index << ",end_index=" << para->end_index << endl;
2353     }
2354     void start_tasks(){
2355         thread threads[2]; TaskPara* paras[2];
2356
2357         for(int i = 0; i < 2; i++){
2358             int start_index = i + 1;
2359             int end_index = start_index*100 ;
2360             paras[i] = new TaskPara(start_index, end_index);
2361             cout << "start_index=" << start_index << ",end_index=" << end_index << endl;
2362             threads[i] = thread(tfun, ref(paras[i]));
2363         }
2364         for(int i = 0; i < 2; i++){
2365             threads[i].join(); delete paras[i];
2366         }
2367     }
2368     int main()
2369     {
2370         cout << "start ....." << endl;
2371         start_tasks();
2372         this_thread::sleep_for(chrono::seconds(1));
2373     }

```

问题 8 输出 调试控制台 终端 JUPYTER

```
start .....
start_index=1,end_index=100
start_index=2,end_index=200
thrad_id:2,start_index=1,end_index=100
thrad id:3,start index=2,end index=200
```

## 定时器线程

### 简易 C++ Timer 的实现

<https://stackoverflow.com/questions/14650885/how-to-create-timer-events-using-c-11>

<https://www.cnblogs.com/yunlambert/p/10226468.html>

```
#include <iostream>
#include <string>
#include <thread>
#include <chrono>
#include <functional>
#include <future>

using namespace std;

class Timer {
    bool clear = false;
public:
    template <class callable, class... arguments>
    void setInterval(int interval, callable&& f, arguments&&... args);
    void stop();
};

template <class callable, class... arguments>
void Timer::setInterval(int interval, callable&& f, arguments&&... args) {
    // function<typename result_of<callable(arguments...)>::type()> task(bind(forward<callable>(f), forward<arguments>(args)...));
    // this->clear = false;
    // 线程中使用lambda函数
    thread t([=]{
        // 死循环+sleep
        while (true) {
            task(); // 执行函数
            std::this_thread::sleep_for(std::chrono::seconds(interval));
            if (this->clear) {
                cout << "timer finised!!!!!" << endl;
                return;
            }
        }
    });
    t.detach();
}

void Timer::stop() {
    this->clear = true;
}

void testfun(int a, string info)
{
    cout << a << ":" << info << endl;
    return;
}

int main() {
    Timer t = Timer();
    t.setInterval(1, &testfun, 10, "testfun");
    cout << "I am main thread" << endl;
    std::this_thread::sleep_for(std::chrono::seconds(5));
    t.stop();
    std::this_thread::sleep_for(std::chrono::seconds());
    cout << "all finised!!!!!" << endl;
}
```

运行结果为：

```
[Running] cd "d:\wujianjun\code\my\zacode\zacpp\src\" && g++ test.cpp -o test && "d:\wujianjun\code\my\zacode\zacpp\src\"test
I am main thread
10:testfun
10:testfun
10:testfun
10:testfun
10:testfun
10:testfun
timer finised!!!!!
all finised!!!!!

[Done] exited with code=0 in 14.398 seconds
```

这个实现有几个问题：

- ◆ clear 字段不是多线程安全的。

## 线程的资源管理

### 孤儿进程和僵尸进程

- ◆ 僵尸进程：当进程 `exit()` 退出之后，他的父进程没有通过 `wait()` 系统调用回收他的进程描述符的信息，该进程会继续停留在系统的进程表中，占用内核资源，这样的进程就是僵尸进程。
- ◆ 孤儿进程：当一个进程正在运行时，他的父进程忽然退出，此时该进程就是一个孤儿进程。作为一个进程，需要找到一个父进程，否则这种进程在退出之后没人回收他的进程描述符，空耗内存。此时该进程会找到一个父进程，如果自己所在的进程组没人收养，那就作为 `init` 进程的子进程。

### `join()`与`detach()`

```
int main()
{
    d2 = std::thread(download2);
    download1();
    d2.join(); //主线程main需要等d2子线程(download2)结束了才会往下走去执行后面的process
    process();
}
```

- ◆ 主线程调用 `d2.join()` 后，需要等 `d2` 待子线程运行结束了，才会继续往下走。
- ◆ `detach()` 调用会让线程在后台运行，即说明主线程不会等待子线程运行结束才结束。分离的线程在线程退出时系统会自动回收资源。

在一个线程中，开了另一个线程去干另一件事，使用 `join` 函数后，原始线程会等待新线程执行结束之后，再去销毁线程对象。使用 `detach`，那么新线程就会与原线程分离，如果原线程先执行完毕，销毁线程对象及局部变量，并且新线程有共享变量或引用之类，此时新线程可能使用的变量就变成未定义，产生异常或不可预测的错误。

创建一个可运行的线程对象后，要么调用 `join()`，要么调用 `detach()`，否则线程对象析构时程序将直接退出。

C++11 中，线程对象(`std::thread`)创建后，有两种状态：

- ◆ `joinable`: 线程对象通过有参构造函数创建后状态为 `joinable`。
  - ◆ `nonjoinable`: 线程对象通过默认构造函数构造后状态为 `nonjoinable`;
- `joinable` 状态的线程对象被调用 `join()` 或者 `detach()` 会变成 `nonjoinable` 状态。

### linux 线程资源回收方法

一个进程中的线程之间是没有父子之分的，都是平级关系。即线程都是一样的，退出了一个不会影响另外一个。但是所谓的"主线程"`main`, 执行完之后，会调用 `exit()`。`exit()` 会让整个进程 over 终止，那所有线程自然都会退出。

- ◆ 父线程是主线程，则父线程退出后，子线程一定会退出。
- ◆ 父线程不是主线程，则父线程退出后，子线程不受影响，继续运行。

## How much overhead is there when creating a thread?

Nafnlaus

To resurrect this old thread, I just did some simple test code:

```
#include <thread>

int main(int argc, char** argv)
{
    for (volatile int i = 0; i < 500000; i++)
        std::thread([](){}).detach();
    return 0;
}
```

I compiled it with `g++ test.cpp -std=c++11 -lpthread -O3 -o test`. I then ran it three times in a row on an old (kernel 2.6.18) heavily loaded (doing a database rebuild) slow laptop (Intel core i5-2540M). Results from three consecutive runs: 5.647s, 5.515s, and 5.561s. So we're looking at a tad over 10 microseconds per thread on this machine, probably much less on yours.

That's not much overhead at all. Of course there's various additional thread losses one can get involving passed/captured arguments, cache slowdowns between cores (if multiple threads on different cores are battling over the same memory at the same time), etc.

Note that thread pools can in some cases be a slowdown versus unique threads, since one of the biggest slowdowns with threads is synchronizing cached memory used by multiple threads at the same time, and thread pools by their very nature of having to look for and process updates from a different thread have to do this. By contrast, in an ideal situation, a unique processing thread for a given task only has to share memory with its calling task once (when it's launched) and then they never interfere with each other again.

## Cost of a thread in C++ under Linux

If you wanted to increment a counter using a C++ thread, you could do it in this manner:

```
auto mythread = std::thread([] { counter++; });
mythread.join();
```

Creating a new thread is not free. I wrote a small benchmark where I just create a thread, increment a counter and let the thread die. For simplicity, I am just going to report the means:

system	time per thread
Ampere server (Linux, ARM)	200,000 ns
Skylake server (Linux, Intel)	9,000 ns
Rome server (Linux, AMD)	20,000 ns

1s = 1000ms, 1ms = 1000μs(微秒), 1μs = 1000ns

## How long does thread creation and termination take under Windows?

Jerry Coffin

写了一个例子，同时创建 32 个线程，每个线程循环 10 万次 Sleep(0)。结果如下：

```
Milliseconds to create thread: 0.015625
Microseconds per thread switch: 0.0479687
```

也就是创建一个线程平均需要 0.015ms，但是每次线程切换的时间约 0.047ms。

也就是说，线程的创建与销毁代价不大，但是线程的切换和同步，是最花时间的。

## 多线程的内存模型

<https://zhuanlan.zhihu.com/p/382372072>

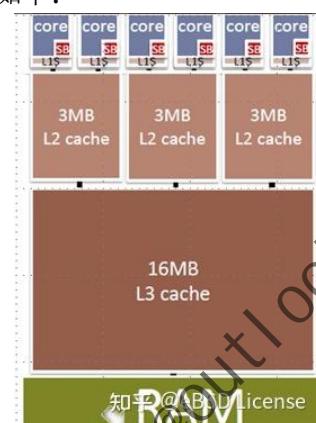
### 编译器和 CPU 的优化

此处说的内存模型是指多线程编程方面，而非对象的内存布局内存对齐之类。

如图 1 所示，我们编程都是基于此种理想架构，CPU Core 直接对 RAM 进行读写，无需考虑二者速度差异。



实际的 CPU/RAM 架构更复杂，如下：



在 CPU Core 和 RAM 之间，还有共三层的 Cache 结构(L1/L2/L3)，此外还有 **Store Buffer (SB)**，且 SB 和 L1 是每个 Core 独享，而 L2 是两个 Core 共享，L3 是所有 Core 共享。

为什么非要引入 Cache？根据 Jeff Dean 对程序运行方面的一些时间数据的论述，如图 3 所示：

- L1 cache reference 0.5 ns
- Branch mispredict 5 ns
- L2 cache reference 7 ns
- Mutex lock/unlock 100 ns
- Main memory reference 100 ns
- Compress 1K bytes with Zippy 10,000 ns
- Send 2K bytes over 1 Gbps network 20,000 ns
- Read 1 MB sequentially from memory 250,000 ns
- Round trip within same datacenter 500,000 ns
- Disk seek 10,000,000 ns
- Read 1 MB sequentially from network 10,000,000 ns
- Read 1 MB sequentially from disk 30,000,000 ns
- Send packet CA->Netherlands->CA 150,000,000 ns

在速度上，CPU 和内存之间有两个数量级的差异。如果没有 Cache，CPU 每执行一条指令，都要去内存取下一条，而执行一条指令也就几 ns，而取指令却要上百 ns，这将导致 CPU 大部分时间都在等待状态，进而导致执行效率低下。引入了 Cache 解决了 CPU 等待的问题，但却产生了很多新的问题，例如 Cache 的一致性，Cache 的缺失，**Cache 的乒乓效应**等，为了解决这些问题，各 CPU 都有自己的解决方案，软件层面(编译器)也会有对应的优化。这导致不同 CPU 会采用不同的技巧来优化执行同一份程序。若我们从比互斥锁更为底层地去了解多线程间的同步机制，我们势必会看到 CPU 平台和编译器的差异。现代 C++ 的内存模型，便是为了屏蔽这些差异，而让你可以不用去了解特定平台特定编译器，也不用依赖互斥锁，就可以完成线程间的同步。C++11 开始提供的 std::atomic<>类模板，便可以作为更为底层的同步工具，这也是内存模型起作用的地方。

优化主要包括 Reorder(重排)/Invert/Remove。

对于重排(Reordering)，以下面代码为例，

```
//reordering 重排示例代码
int A = 0, B = 0;
void foo()
{
    A = B + 1; // (1)
    B = 1;      // (2)
}
// g++ -std=c++11 -O2 -S test.cpp
// 编译器重排后的代码
// 注意第一句汇编，已经将B最初的值存到了
// 寄存器eax，而后将该eax的值加1，再赋给A
movl B(%rip), %eax
movl $1, B(%rip)           // B = 1
addl $1, %eax              // A = B + 1
movl %eax, A(%rip)
```

MOV 指令的基本格式：`movx source, destination`。GNU 汇编器为 `mov` 指令添加了一个维度，声明要传送的数据元素的长度。`movx`，其中 `x` 可以是下面的字符：

- l 用于 32 位的长字值
- w 用于 16 位的字值
- b 用于 8 位的字节值

`eax` 是"累加器"(accumulator)，它是很多加法乘法指令的缺省寄存器。

可以看到，**B=1** 居然先于 **A=B+1** 被执行，不过，最终的结果仍然与重排前一致。对于单线程来说，这没什么问题。但是，对于多线程，若另一个线程依赖于未重排的假设，当 `B` 为 1 时，它认为 `A=B+1` 已经执行了。这就会发生非预期的行为。重排有什么好处呢？其实 Cache 的读写单位是 `Cache line`，即一块内存区域，大小一般为 64 字节。若需要读写该区域中的某个变量，则 CPU 会将该区域整体进行读写。假设 `A` 和 `B` 在不同的 `Cache line`，且假定没有重排优化，当 CPU 将 `B` 读入 Cache 之后，Cache 已经满了，若要接着读入 `A`，这时候只能移出某些 `Cache line`，刚好 `B` 所在的 `Cache line` 被移出，这样导致最后一句给 `B` 赋值时，又要再次读取 `B` 到 Cache。而重排只需读取主存一次，而未重排则需要两次读取。

## c++的锁

mutex 是最基本的互斥类：

- ◆ lock: 若加锁则将该 mutex 锁住，如已加锁调用线程将阻塞直到锁被释放。
- ◆ try\_lock: 若加锁则将该 mutex 锁住，如已加锁则调用线程立即返回 false 不阻塞。
- ◆ unlock: 调用线程释放锁。

**mutex 不可以被同一个线程在未释放时重复加锁，否则将出现死锁。recursive\_mutex 则可以在未释放时重复加锁。**

另外，mutex 相关类不支持拷贝构造、赋值，也不支持移动构造以及移动赋值。

例子如下：

```
void tFunc(mutex& m, vector<int>& data)
{
    for(int i = 0; i < 3; i++)
    {
        m.lock();
        cout << this_thread::get_id() << " got lock" << endl ;
        for(int j = 0; j < 100; j++)
        {
            data.push_back(i*j);
        }
        m.unlock();
        cout << this_thread::get_id() << " released lock" << endl ;
        std::this_thread::sleep_for(std::chrono::microseconds(30));
    }
}

int main()
{
    mutex m;
    vector<int> data;
    thread t1(tFunc, ref(m), ref(data)); thread t2(tFunc, ref(m), ref(data));
    t1.join(); t2.join();
    int sum = 0;
    for(const auto& item : data)
    {
        sum += item;
    }
    cout << sum << endl;
}
```

加锁和不加锁运行结果分别如下：

```
(base) [root@bigdata-dev01-e9cd zacpp]# g++ t.cpp -lpthread -o t
(base) [root@bigdata-dev01-e9cd zacpp]# ./t
140667006310144 got lock
140667006310144 released lock
140667006310144 got lock
140667006310144 released lock
140667006310144 got lock
140667006310144 released lock
140667014702848 got lock
140667014702848 released lock
140667014702848 got lock
140667014702848 released lock
140667014702848 got lock
140667014702848 released lock
29700
(base) [root@bigdata-dev01-e9cd zacpp]# ./t
140132901996288 got lock
140132901996288 released lock
140132901996288 got lock
140132901996288 released lock
140132901996288 got lock
140132901996288 released lock
140132893603584 got lock
140132893603584 released lock
140132893603584 got lock
140132893603584 released lock
140132893603584 got lock
140132893603584 released lock
29700
(base) [root@bigdata-dev01-e9cd zacpp]# ./t
140125219489536 got lock got lock
140125219489536 released lock
140125219489536 got lock
140125219489536 released lock
140125219489536 got lock
140125219489536 released lock
Segmentation fault
```

可以看到不加锁很容易出现段错误，加锁后再无此错。

oop, 多线程, 指针与锁的综合小例子如下:

```
class A
{
public:
    A(vector<int>& data_)
    {
        data.swap(data_);
    }
    void myf(int* cdata, int dlen)
    {
        thread t1(tFunc, ref(m), cdata, ref(dlen));
        thread t2(tFunc, ref(m), cdata, ref(dlen));
        t1.join();
        t2.join();
        int sum = 0;
        for(int j = 0; j < dlen; j++)
        {
            sum += cdata[j];
        }
        cout << sum << endl;
    }
    static void tFunc(mutex& m, int* cdata, int dlen)
    {
        for(int i = 0; i < 5; i++)
        {
            m.lock();
            cout << this_thread::get_id() << " got lock" << endl ;
            for(int j = 0; j < dlen; j++)
            {
                cdata[j] += j;
            }
            m.unlock();
            cout << this_thread::get_id() << " released lock" << endl ;
            std::this_thread::sleep_for(std::chrono::microseconds(30));
        }
    }
private:
    mutex m; // 无需显示初始化
    vector<int> data;
};

int main()
{
    vector<int> data;
    A a(data);

    int cdata[3] = {0,0,0};
    int dlen = 3;
    a.myf(cdata, dlen);
}
```

运行结果如下:

```
(base) [root@bigdata-dev01-e9cd zacpp]# g++ t.cpp -lpthread -o t
(base) [root@bigdata-dev01-e9cd zacpp]# ./t
140246695601920 got lock
140246695601920 released lock
140246687209216 got lock
140246687209216 released lock
140246695601920 got lock
140246695601920 released lock
140246687209216 got lock
140246687209216 released lock
140246695601920 got lock
140246695601920 released lock
140246687209216 got lock
140246687209216 released lock
140246695601920 got lock
140246695601920 released lock
140246695601920 got lock
140246695601920 released lock
30
(base) [root@bigdata-dev01-e9cd zacpp]#
```

# 高性能多线程

## Poor performance in multi-threaded C++ program

<https://stackoverflow.com/questions/15177726/poor-performance-in-multi-threaded-c-program>

I have a C++ program running on Linux in which a new thread is created to do some computationally expensive work independent of the main thread. However, I'm getting relatively poor performance. *I've tried a couple of things, including pthread\_setaffinity\_np to set the thread to a single CPU, as well as pthread\_setschedparam to set the scheduling policy.* But the effects of these have so far been negligible.

Mats Petersson:

This is because of one or more of these things:

### 1. Some lock in a thread is blocking the second thread from running.

If there is a thread that takes a lock, and another thread wants to use the resource that is locked by this thread, it will have to wait. Using some code to identify if locks are holding your code:

```
while (!tryLock(some_lock))
{
    tried_locking_failed[lock_id][thread_id]++;
}
total_locks[some_lock]++;
```

Printing some stats of the locks would help to identify where the locking is contentious.

### 2. Sharing of data between threads (either true or "false" sharing)

If two threads use the same variable, then the two threads will have to swap "I've updated this" messages, and the CPU's have to fetch the data from the other CPU before it can continue.

### 3. Cache thrashing.

If two threads do a lot of memory reading and writing, the cache of the CPU may be constantly throwing away good data to fill it with data for the other thread. If the data is  $2^n$  and fairly large (a multiple of the cache-size), it's a good idea to "add an offset" for each thread. When the second thread reads the same distance into the data region, it will not overwrite exactly the same area of cache that the first thread is currently using.

### 4. Competition for some external resource causing thrashing and/or blocking.

If two threads are reading or writing from/to the hard-disk, network card, or some other shared resource, this can lead to one thread blocking another thread. It is also possible that the code detects different threads and does some extra flushing to ensure that data is written in the correct order or similar, before starting work with the other thread.

### 5. Generally bad design

This is a "catchall" for "lots of other things that can be wrong".

If the result from one calculation in one thread is needed to progress the other, obviously, not a lot of work can be done in that thread.

Too small a work-unit, so all the time is spent starting and stopping the thread, and not enough work is being done..

## Optimizations for C++ multi-threaded programming

<https://medium.com/distributed-knowledge/optimizations-for-c-multi-threaded-programs-33284dee5e9c>

### Software prerequisites

Sample code snippets in this article are benchmarked and profiled using GoogleBenchmark and perf profiler in Linux. Please spend some time to download and set them up before running the code.

### Understanding your hardware

Hardware knowledge is fundamental for writing fast code. Knowing how long it takes for a computer to execute an instruction or how data is stored and fetched to different types of memory allows programmers to optimize their code optimally. Here's an example for a typical modern CPU:

## 为什么多线程会带来性能问题

### 主要有开销有：

- ◆ 上下文切换：线程数往往是大于 CPU 核心数的，操作系统会按照调度算法给每个线程分配时间片运行，此时就会引起上下文切换，其开销是比较大的。
- ◆ 缓存失效：一旦进行了线程调度，切换到其他线程，CPU 就会去执行不同的代码，原有的缓存就很可能失效了，需要重新缓存新的数据，这也会造成一定的开销。
- ◆ 协作开销：线程之间如果有共享数据，为了避免数据错乱，保证线程安全，就有可能禁止编译器和 CPU 对其进行重排序等优化，也可能出于同步的目的，反复把线程工作内存的数据 flush 到主存中，然后再从主内存 refresh 到其他线程的工作内存中等等。

## Top 20 C++ multithreading mistakes and how to avoid them

<https://www.acodersjourney.com/top-20-cplusplus-multithreading-mistakes/>

## 现代 C++ 的内存模型和高性能的多线程编程

<https://skyscribe.github.io/post/2019/11/04/cpp-memory-model-and-order/>

## c++性能优化指南

<https://yun.weicheng.men/Book/C%2B%2B%E6%80%A7%E8%83%BD%E4%BC%98%E5%8C%96%>

[E6%8C%87%E5%8D%97.pdf](#)

## OpenMP

OpenMP 是一套支持跨平台**共享内存方式**的多线程并发的编程 API。支持 OpenMP 的编译器包括 GCC、LLVM。**OpenMP 提供了对并行算法的高层的抽象描述，程序员通过在源代码中加入专用的 `pragma` 来指明自己的意图，由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信。**当选择忽略这些 `pragma`，或者编译器不支持 OpenMP 时，程序又可退化为通常的程序(一般为串行)，程序码仍然可以正常运作。

在 C/C++ 中，OpenMP 的函数都声明在头文件 `omp.h` 中。

## C/C++ 多线程编程/ 绑定 CPU

<https://zhuanlan.zhihu.com/p/57470627>

<https://blog.csdn.net/zgcjaxj/article/details/106594672>

## 编译优化与体系结构

### 基本概览

现代 C/C++ 编译器有多智能？能做出什么厉害的优化？

<https://www.zhihu.com/question/43598164>

发现很多时候自以为的优化其实编译器早就优化过了。举个栗子：

1. 循环展开，大部分编译器设置 `flag` 后会自动展开；
2. 顺序 SIMD 优化，大部分编译器设置 `flag` 后也会自动优化成 SIMD 指令；
3. 减少中间变量，大部分编译器会自动优化掉中间变量；

## gcc -O0 -O1 -O2 -O3 四级优化选项及每级分别做什么优化

gcc 提供了近百种优化选项，用来对{编译时间，目标文件长度，执行效率}这个三维模型进行不同的取舍和平衡。优化的方法不一而足，总体上将有以下几类：

- ◆ 精简操作指令；
- ◆ 尽量满足 cpu 的流水操作；
- ◆ 通过对程序行为地猜测，重新调整代码的执行顺序；
- ◆ 充分使用寄存器；
- ◆ 对简单的调用进行展开等等。

-O1：使用本级别编译器会尝试减小生成代码的尺寸以及缩短执行时间。

-O2：O2 优化增加了编译时间的基础上，提高了生成代码的执行效率。

-O3：在包含了 O2 所有的优化的基础上，又打开了以下优化选项：

- ◆ I -finline-functions：内联简单的函数到被调用函数中。由编译器启发式的决定哪些函数足够简单可以做这种内联优化。

优化代码有可能带来的问题

- ◆ 调试问题：任何级别的优化都将带来代码结构的改变。例如：对分支的合并和消除，将会使目标代码的执行顺序变得面目全非，导致调试信息严重不足。
- ◆ 内存操作顺序改变所带来的问题：例如：-fforce-mem 有可能导致内存与寄存器之间的数据产生类似脏数据的不一致等。可以采用 volatile 关键字限制变量的操作方式，或者利用 std::barrier 迫使 cpu 严格按照指令序执行的。

## C++服务编译耗时优化原理及实践

<https://tech.meituan.com/2020/12/10/apache-kylin-practice-in-meituan.html>

## GCC 编译优化指南

[http://www.jinbuguo.com/linux/optimize\\_guide.html](http://www.jinbuguo.com/linux/optimize_guide.html)

## cache 友好编程

现在主存的速度已经超过 CPU 的速度，那么 CPU 片内的 cache 是否可以取消？

作者：超合金彩虹糖

缓存目前还是不可缺少的，缓存的带宽以及延迟都是远远吊打主流内存的：

AIDA64 Cache & Memory Benchmark				
	Read	Write	Copy	Latency
Memory	54220 MB/s	53113 MB/s	51691 MB/s	48.8 ns
L1 Cache	3399.1 GB/s	1723.5 GB/s	3443.9 GB/s	1.0 ns
L2 Cache	1018.1 GB/s	517.74 GB/s	792.32 GB/s	2.7 ns
L3 Cache	328.23 GB/s	199.91 GB/s	263.02 GB/s	12.3 ns

L1、L2、L3 不管是延迟还是带宽都是远远强于内存的，内存相比之下还是太慢了。

### Cache 的基本原理

接触 cache 之前，先为你准备段 code 分析。

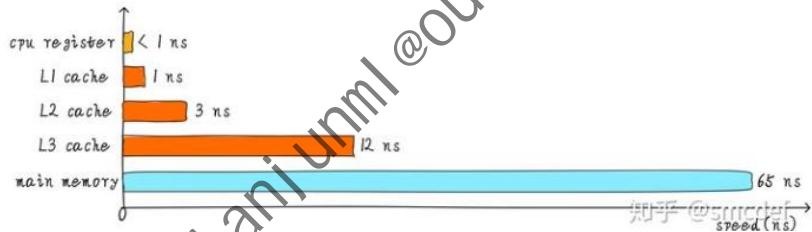
```
int arr[10][128];  
  
for (i = 0; i < 10; i++)  
    for (j = 0; j < 128; j++)  
        arr[i][j] = 1;
```

你有没有想过这段 code 还有下面的一种写法。

```
int arr[10][128];  
  
for (i = 0; i < 128; i++)  
    for (j = 0; j < 10; j++)  
        arr[j][i] = 1;
```

功能完全一样，但是我们一直在重复着第一种写法，你是否想过这其中的缘由?(二者速度可以相差一倍)。cache 是如何影响这 2 段 code 的呢?

**CPU register 的速度一般小于 1ns，主存的速度一般是 65ns 左右。**当 CPU 试图从主存中 load/store 操作时，CPU 不得不等待这漫长的 65ns 时间。我们制作一块速度极快但是容量极小的存储设备(称之为 cache memory)。我们将 cache 放置在 CPU 和主存之间，当 CPU 试图从主存中 load/store 数据的时候，CPU 会首先从 cache 中查找对应地址的数据是否缓存在 cache 中。如果其数据缓存在 cache 中，直接从 cache 中拿到数据并返回给 CPU。我们在 L1 cache 后面连接 L2 cache，在 L2 cache 和主存之间连接 L3 cache。不同等级 cache 速度之间关系如下：



通常 L1 和 L2 缓存是不同内核单独享有的，L3 缓存是多个内核共享的。Cache 中的内容用 LRU 算法进行淘汰。

### C++性能榨汁机之循环展开

循环展开(Loop unwinding 或 loop unrolling)，是一种牺牲程序的尺寸来加快程序的执行速度的优化方法。首先看未经过循环展开优化的代码：

```
238 int main(){
239
240     auto start = std::chrono::system_clock::now();
241
242     int sum = 0;
243     int count = 1000000;
244     //循环10000次累加
245     for(int i = 0;i < count;i++){
246         sum += i;
247     }
248
249     auto end = std::chrono::system_clock::now();
250
251     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
252     std::cout <<"共耗时："<< duration.count() << " 微秒" << std::endl;
253     return 0;
254 }
```

问题 1 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ main.cpp -o main && "d:\wujianjun\co  
共耗时： 1993 微秒

下面我们将循环展开一次，即把上述代码中的循环改为如下代码：

```
238 int main(){
239
240     auto start = std::chrono::system_clock::now();
241
242     int sum = 0;
243     int count = 1000000;
244     for(int i = 0;i < count;i += 2)
245     {
246         sum += i;
247         sum += i+1;
248     }
249
250     auto end = std::chrono::system_clock::now();
251
252     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
253     std::cout <<"共耗时："<< duration.count() << " 微秒" << std::endl;
254     return 0;
255 }
```

问题 1 输出 调试控制台 终端  
[Running] cd "d:\wujianjun\code\my\zacpp\src\src\" && g++ main.cpp -o main && "d:\wujianjun\co  
共耗时： 1026 微秒

即每次循环将 *i* 和 *i+1* 一起累加到 *sum* 变量上，这样可以把循环次数降低一半，同时代码耗时缩短近一半。

# 内核与网络

## C++致命错误

### segmentation fault

What is a bus error? Is it different from a segmentation fault?

bltxd:

Bus errors are rare nowadays on x86, typically:

- ◆ using a processor instruction with an address that does not satisfy its alignment requirements.

Segmentation faults occur when accessing memory which does not belong to your process. They are very common and are typically the result of:

- ◆ using a pointer to something that was deallocated.
- ◆ using an uninitialized hence bogus pointer.
- ◆ using a null pointer.
- ◆ overflowing a buffer.

bus error 和 segment fault:

我们经常会发现有两种内存转储(core dump)

- ◆ segment fault:通常是在一个非法的地址上进行取值赋值操作造成。
- ◆ bus error:通常是指针强制转换，导致CPU读取数据违反了某个总线规则。

bus error 几乎都是有内存未对齐读引起的。一般情况下，编译器都会做好内存对齐工作。很多情况就是由指针和强制类型转换引起的。

What is your way to generate a "stack overflow" in a C/C++ program?

Aaron Swords:

```
int dieViolently () {
    dieViolently();
}
```

## c++ debug 工具

### c++性能优化概述

C++ tricks for competitive programming (for C++ 11)

<https://www.geeksforgeeks.org/c-tricks-competitive-programming-c-11/>

c++性能榨汁机

惊群问题

循环展开

无锁编程

局部性原理

伪共享

分支预测器

switch 语句的背后

**指针与引用**

**虚函数的开销**

**CPU 亲和性**

wujianjunml@outlook.com