遇到的问题

内置

<algorithm>库

partition() & stable_partition()

- partition 可直译为“分组”，partition() 函数可根据用户自定义的筛选规则，重新排列指定区域内存储的数据，使其分为 2 组，第一组为符合筛选条件的数据，另一组为不符合筛选条件的数据。
    - 第一个第二个参数是给定容器的范围，第三个参数是给定的分组规则，返回值为true的所有元素作为第一个子序列，返回值为false的作为第二个子序列， 返回值是 指向第二个子序列的首元素迭代器...
- partition() & stable_partition() 两个方法的区别在于， partition()对于两个子序列中的元素并不排序，而stable_partition()则对两个子序列的元素也进行排序。

```
1
2
```

unique函数

类属性算法unique的作用是从输入序列中“**删除”所有相邻的重复元素**。

该算法删除相邻的重复元素，然后重新排列输入范围内的元素，并且返回一个迭代器（容器的长度没变，只是元素顺序改变了），表示无重复的值范围得结束。

注意，words的大小并没有改变，依然保存着10个元素；只是这些元素的顺序改变了。调用unique“删除”了相邻的重复值。给“删除”加上引号是因为unique实际上并没有删除任何元素，而是将无重复的元素复制到序列的前段，从而覆盖相邻的重复元素。unique返回的迭代器指向超出无重复的元素范围末端的下一个位置。

注意：算法不直接修改容器的大小。如果需要添加或删除元素，则必须使用容器操作。

```
/*删除重复的元素并排序：
sort进行排序
earse的功能是删除指定范围内的所有函数
unique将相邻的重复的元素移到最后*/
// sort words alphabetically so we can find the duplicates
sort(words.begin(), words.end());
     /* eliminate duplicate words:
      * unique reorders words so that each word appears once in the
      *    front portion of words and returns an iterator one past the
unique range;
      * erase uses a vector operation to remove the nonunique elements
      */
 vector<string>::iterator end_unique =  unique(words.begin(), words.end());
 words.erase(end_unique, words.end());
```

类型转换

std::stod & std::stoi

```
/*
 * 作用：将string转为double
 * 语法：
 * double stod (const string&  str, size_t* idx = 0);
 * double stod (const wstring& str, size_t* idx = 0);
 */

double a = std::stod("234.3345");
int b = std::stoi("234.3345")
/* a = 234.3345
   b = 234  ??*/

```

内存管理

内存分配函数

malloc:

```
extern void *malloc(unsigned int num_bytes);
```

功能：分配长度为num_bytes字节的内存块

返回值：如果分配成功则返回指向被分配内存的指针(此存储区中的初始值不确定)，否则返回空指针NULL。当内存不再使用时，应使用free()函数将内存块释放。函数返回的指针一定要适当对齐，使其可以用于任何数据对象。返回类型是 void* 类型，void* 表示未确定类型的指针。C,C++规定，void* 类型可以强制转换为任何其它类型的指针。

备注：void* 表示未确定类型的指针，更明确的说是指申请内存空间时还不知道用户是用这段空间来存储什么类型的数据（比如是char还是int或者...）

从函数声明上可以看出。malloc 和 new 至少有两个不同: new 返回指定类型的指针，并且可以自动计算所需要大小。

calloc: 是一个C语言函数

```
void *calloc(unsigned n,unsigned size);
```

功 能: 在内存的动态存储区中分配n个长度为size的连续空间，函数返回一个指向分配起始地址的指针；如果分配不成功，返回NULL。

跟malloc的区别：calloc在动态分配完内存后，自动初始化该内存空间为零，而malloc不初始化，里边数据是随机的垃圾数据。

realloc:

```
extern void *realloc(void *mem_address, unsigned int newsize);

//使用方法
指针名=（数据类型*）realloc（要改变内存大小的指针名，新的大小）。
  //新的大小一定要大于原来的大小不然的话会导致数据丢失!
```

头文件：#include <stdlib.h> 有些编译器需要#include <malloc.h>，在TC2.0中可以使用alloc.h头文件

功能：先判断当前的指针是否有足够的连续空间，如果有，扩大mem_address指向的地址，并且将mem_address返回，如果空间不够，先按照*newsize*指定的大小分配空间，将原有数据从头到尾拷贝到新分配的内存区域，而后释放原来mem_address所指内存区域，同时返回新分配的内存区域的首地址。即重新分配存储器块的地址。

返回值：如果重新分配成功则返回指向被分配内存的指针，否则返回空指针NULL。

注意：这里原始内存中的数据还是保持不变的。当内存不再使用时，应使用free()函数将内存块释放。

alloca:

函数原型为：

```
void * __cdecl alloca(size_t);
```

内存分配函数,与malloc,calloc,realloc类似.
　　但是注意一个重要的区别,_alloca是在栈*(stack)*上申请空间,用完马上就释放.
　　包含在头文件malloc.h中
　　在某些系统中会宏定义成_alloca使用.

memset()

- Memset 用来对一段内存空间全部设置为某个字符，一般用在对定义的字符串进行初始化为‘ ’或‘/0’;

```
/* buffer: 某段内存空间的指针
 * c: 要给buffer赋的值
 * count: buffer 的长度
 */
extern void *memset(void *buffer, int c, int count);

```

memcpy()

- memcpy指的是c和c++使用的内存拷贝函数，memcpy函数的功能是从源所指的内存地址的起始位置开始拷贝n个字节到目标所指的内存地址的起始位置中。

```
/*函数原型
 *参数1: 目的地址
 * 参数2: 源地址
 * 参数3: 需要复制的 字节数
*/
_CRTIMP int __cdecl __MINGW_NOTHROW
  memcmp (const void* dest, const void* , size_t) __MINGW_ATTRIB_PURE;
```

- 情况1：目的地址与原地址数据宽度相同的时候。

```c
int main()
{
    char a[4] = "mmmm";
    char b[7] = "123455";
    memcpy(b,a,3);
    printf("%d\n\r",sizeof(b));
    printf("%s\n",b);
      //mmm455
}
```

- 目的地址的宽度比原地址数据宽度大

```c
int main()
{
    char a[8] = "abcdef";
    short b[4] = {0x17,0x18,0x19,0x19};
//目的地址数据宽
    memcpy(b,a,6);
    printf("b[0]的值是%c\n",b[0]);  //a
    printf("b[0]的值是%c\n",b[0]>>8);//b
    printf("b[1]的值是%x\n",b[1]);//6463  先存储低地址，再存储高地址
    printf("b[2]的值是%x\n",b[2]);//6665
    printf("b[3]的值是%x\n",b[3]);//19
   return 0;
}

```

- 目的地址的宽度比原地址的宽度小

```c
int main()
{
//源地址数据宽
    short c[5] = {0x1234,0x5678,0x2345,0x3390};
    char  d[10] = {0};
    memcpy(d,c,6);
    for(int i = 0; i < sizeof(d); i++)
        printf("d[%d]的值是%x\n\r",i,d[i]);
  //34
  //12
  //78
  //56
  //45
  //23
  //0000
}

```

(元组)tuple

tuple是一个固定大小的不同类型值的集合，是泛化的std::pair。和c#中的tuple类似，但是比c#中的tuple强大得多。我们也可以把他当做一个通用的结构体来用，不需要创建结构体又获取结构体的特征，在某些情况下可以取代结构体使程序更简洁，直观。

`<fstream>`

ofstream/ifstream/fstream

打开文件

```
// 第一参数指定要打开的文件的名称和位置，
// 第二个参数定义文件被打开的模式。
void open(const char *filename, ios::openmode mode);
//ios::app
//ate
//in
//out
//trunc
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

关闭文件

当 C++ 程序终止时，它会自动关闭刷新所有流，释放所有分配的内存，并关闭所有打开的文件。但程序员应该养成一个好习惯，在程序终止前关闭所有打开的文件。

下面是 close() 函数的标准语法，close() 函数是 fstream、ifstream 和 ofstream 对象的一个成员。

```
void close();
```

写入文件

> 在 C++ 编程中，我们使用流插入运算符（ << ）向文件写入信息，就像使用该运算符输出信息到屏幕上一样。唯一不同的是，在这里您使用的是 **ofstream** 或 **fstream** 对象，而不是 **cout** 对象。

读取文件

> 在 C++ 编程中，我们使用流提取运算符（ >> ）从文件读取信息，就像使用该运算符从键盘输入信息一样。唯一不同的是，在这里您使用的是 **ifstream** 或 **fstream** 对象，而不是 **cin** 对象。

文件位置指针

istream 和 **ostream** 都提供了用于重新定位文件位置指针的成员函数。这些成员函数包括关于 istream 的 **seekg**（"seek get"）和关于 ostream 的 **seekp**（"seek put"）。

seekg 和 seekp 的参数通常是一个长整型。第二个参数可以用于指定查找方向。查找方向可以是 **ios::beg**（默认的，从流的开头开始定位），也可以是 **ios::cur**（从流的当前位置开始定位），也可以是 **ios::end**（从流的末尾开始定位）。

文件位置指针是一个整数值，指定了从文件的起始位置到指针所在位置的字节数。

```
// 定位到 fileObject 的第 n 个字节（假设是 ios::beg）
fileObject.seekg( n );

// 把文件的读指针从 fileObject 当前位置向后移 n 个字节
fileObject.seekg( n, ios::cur );

// 把文件的读指针从 fileObject 末尾往回移 n 个字节
fileObject.seekg( n, ios::end );

// 定位到 fileObject 的末尾
fileObject.seekg( 0, ios::end );
```

读写文件的步骤

1. 声明变量

```
std::ifstream vertexFile;
```

2. 打开文件

```
vertexFile.open(filepath);
//查找错误
vertexFile.exception(std::ifstream::failbit ||
                     std::ifstream::badbit);
//是否打开成功
if (!vertexFile.is_open()|| !fragmentFile.is_open() )
{
   //打开文件出错，丢出一个exception
   throw std::exception("Shader open file error");
}
```

3. 读写操作

```
std::stringstream vertexSStream;    //sstream
vertexSStream << vertexFile.rdbuf();
std::string vertexString = vertexSStream.str();
vertexSource = vertexString.c_str();
```

4. 关闭文件

```
//好习惯
vertexFile.close();
```

断言

static_assert(静态断言)

1. assert 和 #error

- assert是运行期断言，它用来发现运行期间的错误，不能提前到编译期发现错误，也不具有强制性，也谈不上改善编译信息的可读性。既然是运行期检查，对性能肯定是有影响的，所以经常在发行版本中，assert都会被关掉。
- #error可看作是预编译期断言（甚至都算不上断言），仅仅能在预编译时显示一个错误信息，可以配合#ifdef/ifndef参与预编译的条件检查。由于它无法获得编译信息，当然，也就做不了更进一步

分析了。

2. static_assert 静态断言

- 编译期间的断言，语法如下：如果第一个参数常量表达式的值为false，会产生一条编译错误。错误位置就是该static_assert语句所在行，第二个参数就是错误提示字符串。

```
static_assert(常量表达式，"提示字符串")
```

- 使用场景：
  1. 使用static_assert，可以在编译期发现更多的错误，用编译器来强制保证一些契约，帮助我们改善编译信息的可读性，尤其是用于模板时。
  2. 使用范围：static_assert可以用在全局作用域中，命名空间中，类作用域中，函数作用域中，几乎可以不受限制的使用。
  3. 常量表达式：static_assert的断言表达式的结果必须是在编译时期可以计算的表达式，即必须是常量表达式。

```
//该static_assert用来确保编译仅在32位的平台上进行，不支持64位的平台
//该语句可放在文件的开头处，这样可以尽早检查，以节省失败情况下耗费的编译时间
static_assert(sizeof(int) == 4, "64-bit code generation is not supported.");
```

  4. 检查模板参数：编译器在遇到一个static_assert语句时，通常立刻将其第一个参数作为常量表达式进行演算。但如果该常量表达式依赖于某些模板参数，则延迟到模板实例化时再进行演算，这就让检查模板参数也成为了可能。

```
#include <cassert>
#include <cstring>
using namespace std;

template <typename T, typename U> int bit_copy(T& a, U& b)
{
    assert(sizeof(b) == sizeof(a));
    //判断两个类型是否相同，不同则出错
    static_assert(sizeof(b) == sizeof(a), "template parameter size no equal!");
    memcpy(&a, &b, sizeof(b));
};

int main()
{
    int varA = 0x2468;
    double varB;
    bit_copy(varA, varB);
    getchar();
    return 0;
}
```

  5. 性能方面：由于static_assert是编译期间断言，不生成目标代码，因此static_assert不会造成任何运行期性能损失。

- 错误用法

1.
```cpp
int positive(const int n)
{
    static_assert(n > 0, "value must > 0");// n>0在编译的时候不能确定
    return 0;
}
```

STL

difference_type/value_type/

- value_type: 指的是容器中元素的类型

```cpp
vector<int> v;
vector<int>::value_type x;
//相当于 int x;
```

- difference_type: 同一容器对象中不同元素之间的距离

```cpp
vector<int>::iterator i1 = src.begin();
vector<int>::iterator i2 = src.end();
vector<int>::difference_type diff = i1 - i2;
//此时diff存储的是src容器的长度
```

关键字

const

1. 函数前后加const的区别：

- 函数前const：普通函数或成员函数（非静态成员函数）前均可加const修饰，表示函数的返回值为*const*，不可修改。

  `const returnType functionName(param list)`

- 函数后加const：只有类的非静态成员函数后可以加*const*修饰，表示该类的this指针为const类型，不能改变类的成员变量的值，即成员变量为read only（例外情况见2），任何改变成员变量的行为均为非法。此类型的函数可称为只读成员函数.

  `returnType functionName(param list) const`

说明：类中const（函数后面加）与static不能同时修饰成员函数，原因有以下两点：

- C++编译器在实现const的成员函数时，为了确保该函数不能修改类的实例状态，会在函数中添加一个隐式的参数`const this*`。但当一个成员为static的时候，该函数是没有this指针的，也就是说此时const的用法和static是冲突的；
- 两者的语意是矛盾的。static的作用是表示该函数只作用在类型的静态变量上，与类的实例没有关系；而const的作用是确保函数不能修改类的实例的状态，与类型的静态变量没有关系，因此不能同时用它们。

2、const与mutable的区别

在C++中，mutable也是为了突破const的限制而设置的。被mutable修饰的变量（成员变量）将永远处于可变的状态，即使在一个const函数中。因此，后const成员函数中可以改变类的mutable类型的成员变量。

3、const成员函数与const对象

const成员函数还有另外一项作用，即常量对象相关。对于内置的数据类型，我们可以定义它们的常量，对用户自定义的类类型也是一样，可以定义它们的常量对象。有如下规则：

const对象只能调用后const成员函数；

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

class A{
private:
   int m_a;
public:
   A():m_a(0){}
   int getA() const
   {
      return m_a;
   }
   int GetA() //非const成员函数，若在后面加上const修饰则编译通过
   {
      return m_a;
   }

int main()
{
   const A a2;//const对象
   int t;
   t = a2.getA();
   t = a2.GetA();//error:const object call non-const member function,only non-const object can call
}

```

非const对象既可以调用const成员函数，又可以调用非const成员函数。

extern

1. 基本解释

- 作用1：extern可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。

  当extern不与"C"在一起修饰变量或函数时，如在头文件中:`extern int g_Int`;它的作用就是声明函数或全局变量的作用范围的关键字，其声明的函数和变量可以在本模块活其他模块中使用，记住它是一个声明不是定义!也就是说B模块(编译单元)要是引用模块(编译单元)A中定义的全局变量或函数时，它只要包含A模块的头文件即可,在编译阶段，模块B虽然找不到该函数或变量，但它不会报错，它会在连接时从模块A生成的目标代码中找到此函数。

- 作用2：extern也可用来进行链接指定。

当它与"C"一起连用时，如: extern "C" void fun(int a, int b);则告诉编译器在编译fun这个函数名时按着C的规则去翻译相应的函数名而不是C++的，C++的规则在翻译这个函数名时会把fun这个名字变得面目全非，可能是fun@aBc_int_int#%$也可能是别的。

在使用extern时候要严格对应声明时的格式，在实际编程中，这样的错误屡见不鲜。

c++11/14 constexpr用法

1. 在C++11中：

   C++11中的constexpr指定的函数返回值和参数必须要保证是字面值，而且必须有且只有一行return代码。

2. C++14中：

   C++14只要保证返回值和参数是字面值就行了，函数体中可以加入更多的语句，方便了更灵活的计算。

3. const 和 constexpr的比较

   - const并不能代表“常量”，它仅仅是对变量的一个修饰，告诉编译器这个变量只能被初始化，且不能被直接修改（实际上可以通过堆栈溢出等方式修改）。而这个变量的值，可以在运行时也可以在编译时指定。
   - constexpr可以用来修饰变量、函数、构造函数。一旦以上任何元素被constexpr修饰，那么等于说是告诉编译器 “请大胆地将我看成编译时就能得出常量值的表达式去优化我”。

4. constexpr作用：

   1. 给编译器足够的信心在编译期去做被constexpr修饰的表达式的优化。

      ```
      const int func() {
          return 10;
      }
      main(){
        int arr[func()];
      }
      //error : 函数调用在常量表达式中必须具有常量值
      constexpr func() {
          return 10;
      }
      main(){
        int arr[func()];
      }
      //编译通过
      ```

   2. 欺骗编译器？？

c++11 noexcept

该关键字告诉编译器，函数中不会发生异常,这有利于编译器对程序做更多的优化。
如果在运行时，noexecpt函数向外抛出了异常（如果函数内部捕捉了异常并完成处理，这种情况不算抛出异常），程序会直接终止，调用std::terminate()函数，该函数内部会调用std::abort()终止程序。

注意：*C*++中的异常处理不是在编译时候进行的，而是在运行时候检测的，为了检测，编译器会增加在运行时候的代码。

C++的异常处理

C++中的异常处理是在运行时而不是编译时检测的。为了实现运行时检测，编译器创建额外的代码，然而这会妨碍程序优化。
在实践中，一般两种异常抛出方式是常用的：

- 一个操作或者函数可能会抛出一个异常;
- 一个操作或者函数不可能抛出任何异常。

后面这一种方式中在以往的C++版本中常用throw()表示，在C++ 11中已经被noexcept代替。

```
void swap(Type& x, Type& y) throw()   //C++11之前
    {
        x.swap(y);
    }
void swap(Type& x, Type& y) noexcept  //C++11
    {
        x.swap(y);
    }
```

有条件的noexcept

表明在一定条件下不发生异常。

```
void swap(Type& x, Type& y) noexcept(noexcept(x.swap(y)))    //C++11
    {
        x.swap(y);
    }
```

它表示，如果操作x.swap(y)不发生异常，那么函数swap(Type& x, Type& y)一定不发生异常。

肾么时候用noexcept？以下情形鼓励使用noexcept：

- 移动构造函数（move constructor）
- 移动分配函数（move assignment）
- 析构函数（destructor）
- 叶子函数（Leaf Function）。叶子函数是指在函数内部不分配栈空间，也不调用其它函数，也不存储非易失性寄存器，也不处理异常。

template

new

new开辟的空间在堆上，而一般声明的变量存放在栈上。通常来说，当在局部函数中new出一段新的空间，该段空间在局部函数调用结束后仍然能够使用，可以用来向主函数传递参数。

```
int *p = new int;
int pp = *new int;

//
int *p = new int[3];

struct student{
   int age;
  char* name;
};

int *sList = new student[3];

```

new的过程：获得内存空间，调用构造函数，返回正确指针（指针类型的转换）。

- 使用new，分配内存失败会调用new_handler。

C++中一提到new，至少可能代表以下三种含义：new operator、operator new、placement new：

- new operator的第一步分配内存实际上是通过调用operator new来完成的。第一步其实是把new当作一个操作符：
  - 默认情况下首先调用分配内存的代码，尝试得到一段堆上的空间，如果成功就返回，如果失败，则转而去调用一个new_hander，然后继续重复前面过程。
  - 如果我们对这个过程不满意，就可以重载operator new，来设置我们希望的行为。

```
class A
{
public:
   void* operator new(size_t size)
   {
        printf("operator new called/n");
        return ::operator new(size);
   }
};
```

更多用法和内容：https://blog.csdn.net/songthin/article/details/1703966?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromBaidu-1.control&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromBaidu-1.control

VS使用方法

参数提示： `ctrl + shift + space`

查找一切? `ctrl + ,`

常见错误处理

文件流

文件打开失败：

- 可能是因为命名格式不对，有不能出现的符号；