### 培养计划记录

|  |  |
| --- | --- |
| 时间 | 学习内容 |
| 7月14号 | Vs2013:  一、1.空项目2.动态库3.生成exe4.生成静态库5.生成动态库   1. 项目配置：1.输出目录2.中间目录3.调试目录4.拓展名5.附加包含目录6.预处理器7.预编译头8.调试信息9.附加库目录10.依赖库 |

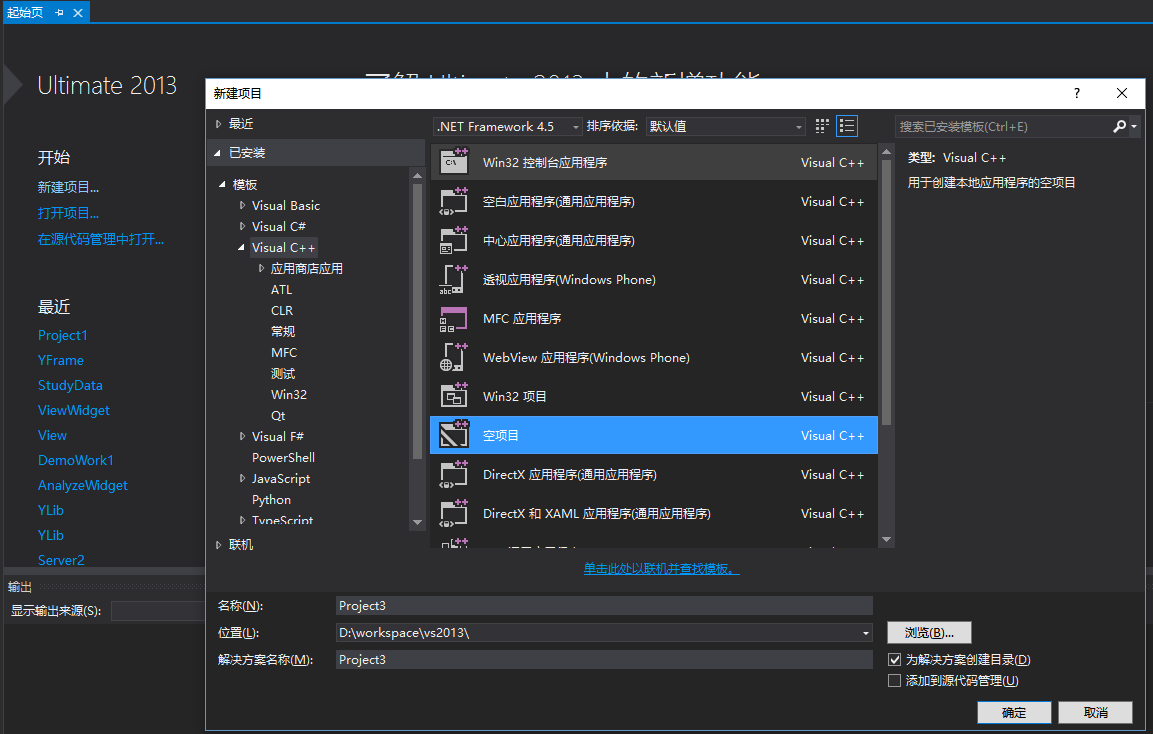
成果：

一、

1. **空项目**

新建项目->visual c++ ->空项目。

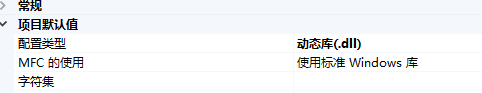
项目目录为解决方案建立目录，同一个解决方案下可以有多个项目（EXE、DLL、LIB）



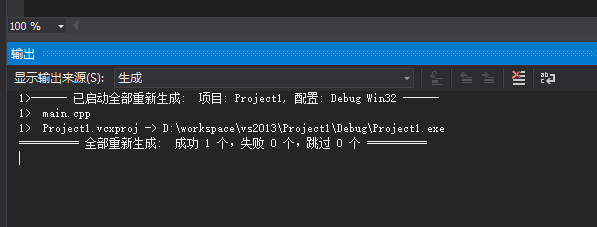
1. **动态库**

从空项目建立后属性页更改配置类型为dll。

也可以直接建立dll模板的项目。



1. **生成exe**

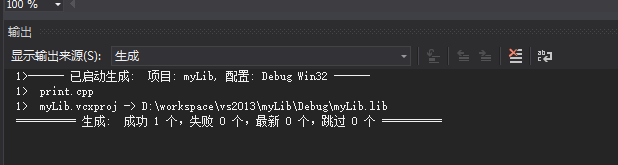


1. **生成静态库**

从空项目建立后属性页更改配置类型为lib。

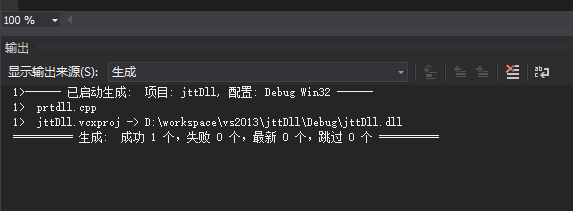
也可以直接建立lib模板的项目。





1. **生成动态库**

需要配置入口函数DllMain,如果没有编译器使用默认的(返回true)。可以添加def文件使用序号导出函数名字。



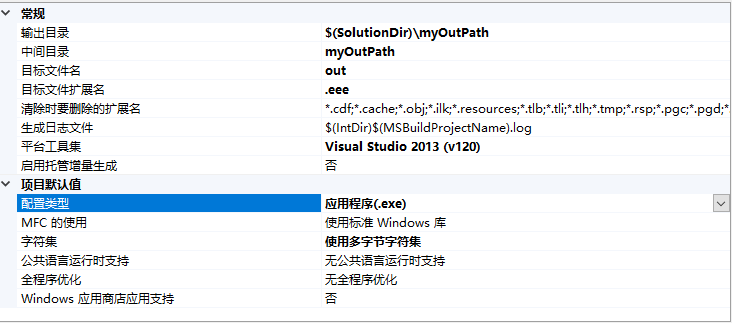
二、**项目配置**

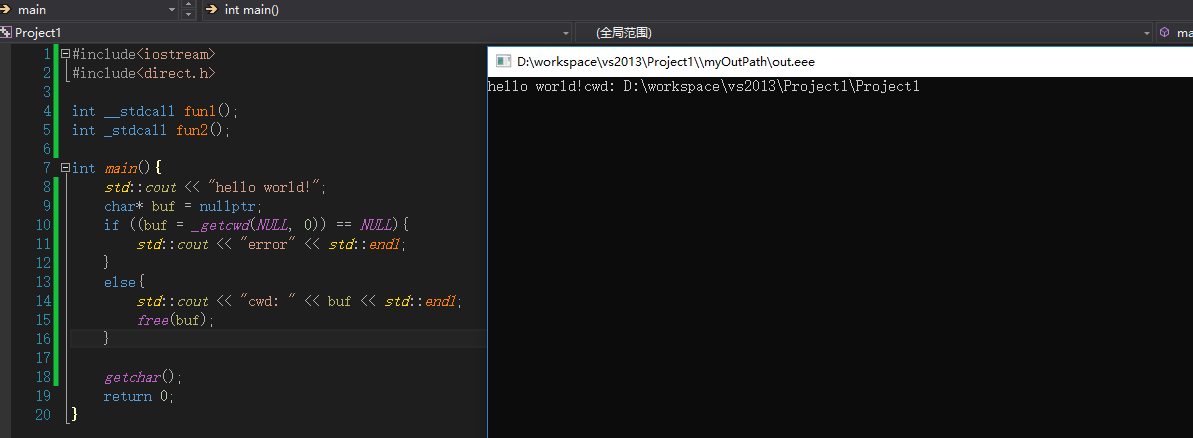
要求：1.输出目录2.中间目录3.调试目录4.拓展名5.附加包含目录6.预处理器7.预编译头8.调试信息9.附加库目录10.依赖库

项目设置可以分成两类

**一类是基础设置**：1.输出目录2.中间目录3.调试目录4.拓展名6.预处理器7.预编译头8.调试信息

如下图将项目输出到解决方案目录下的myOutPath 、中间目录同名为项目目录下的myOutPath 、调试目录为工作目录即执行文件默认运行的位置（见下第二图）、生成的可执行文件和拓展名为 out.eee





预定义宏：solutionDir为解决方案目录 configuration默认为中间文件和调试和release可执行文件位置 ProjectName为可执行文件的名字 目标拓展名为可执行文件拓展名

预编译头可以包含常用通用的头文件和代码，其他的代码都包含这个头文件。

调试信息主要有变量窗口和堆栈窗口，可以通过菜单栏找到更多的选项和调试信息。

**另一类是依赖设置**：5.附加包含目录9.附加库目录10.依赖库（用于添加静态库和动态库）

**添加静态库步骤**：（1）进入项目属性页，在“vc++目录”下的“库路径”，添加LIB文件所在的目录。（2）头文件的路径添加到“包含目录”（3）进入“链接器-->输入”设置，将LIB文件名称添加到“其他依赖项”。见结果展示。

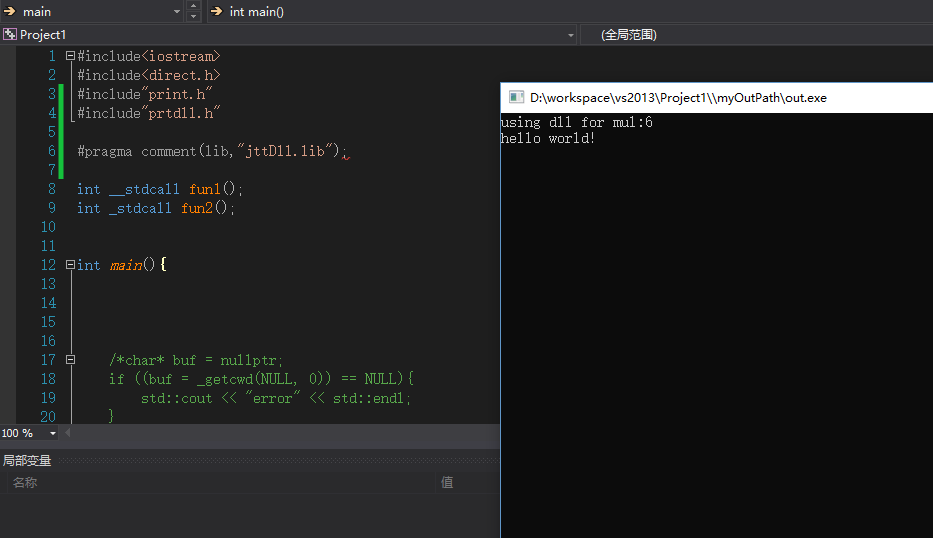
**添加动态库步骤**：dll动态库可以和静态库执行同样的操作进行静态连接。

但通常单独使用，不链接。

Vs中的步骤为：（1）将dll文件复制到项目输出文件夹，或者是到dll的标准Windows搜索路径中的另一个文件夹。参考动态链接库的搜索顺序。

（1）执行和静态库一样的设置vc++目录中的“库路径”和“包含路径”，添加“链接器->输入”中的“其他依赖项”。

结果展示：通过动态库提供mul乘法函数，调用返回乘积。



**使用代码添加静态库和动态库**：

静态库：

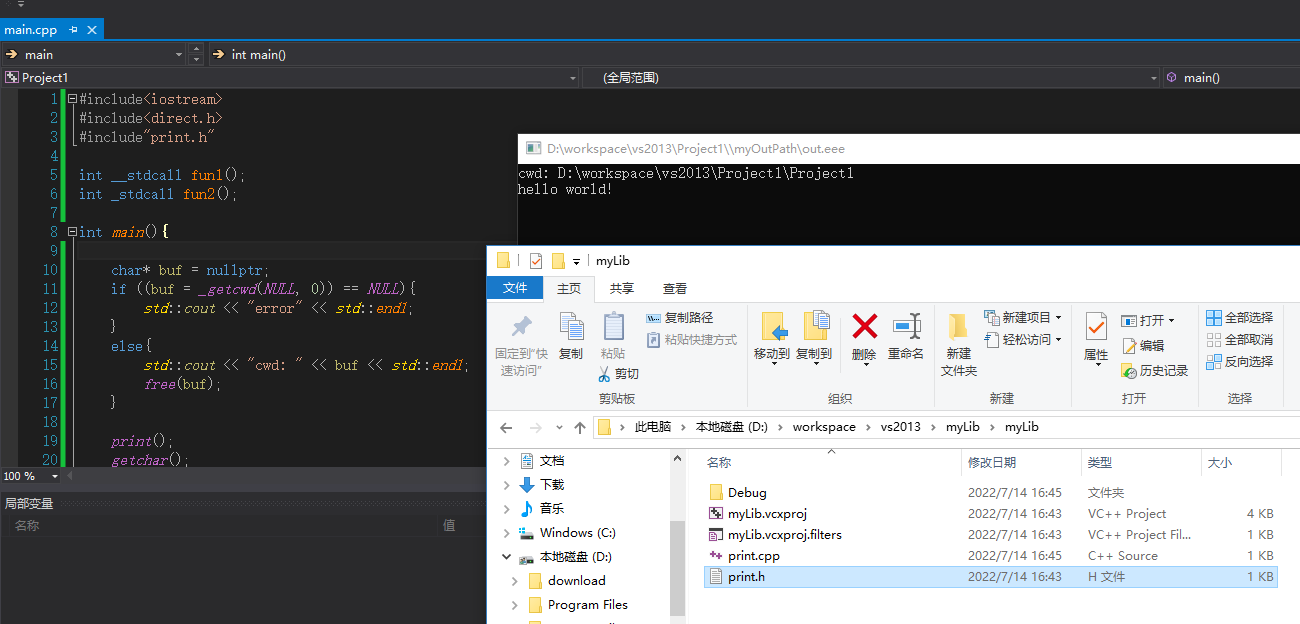
1. 添加库依赖

#pragma comment(lib,”相对\绝对路径\文件名”)

1. 添加头文件

#Include”\*/\*/head.h”

结果展示：通过静态库提供print函数，调用打印hello world；



**动态库**：

<https://blog.csdn.net/qq_44471236/article/details/121152342>

(两种方式，这是使用前加载，还有使用时加载)

1. 拷贝文件3个（lib文件、头文件、和dll文件）。

注意事项:包含头文件时需要添加导出声明：

#include”head.h”

\_declspec(dllexport) int add(int a , int b);

1. 添加库依赖

#pragma comment(lib,”./\*/DLL.lib”)

1. 添加头文件

#include”./include/head.h”

动态库导出方式：

1. 使用\_declspec(dllexport) 优点：简单 缺点：通用性差 定义导出函数时需使用\_stdcall调用约定（调用约定在返回值和函数名之间，常见的有\_stdcall\\_cdecl\\_fastcall\\_thiscall）(注意是一个横线还是两个)

参考：<https://blog.csdn.net/huang714/article/details/88962771>

1. def文件 优点：通用性好 缺点：麻烦 。此方法按照序号而不是名称从dll导出函数。

|  |  |
| --- | --- |
| 时间 | 学习内容 |
| 7月15号 | c语言：各类基础数据类型、全局、局部、静态变量和常量、各数据内存大小、对应数据类型输入输出方法，字符串处理，内存对齐规则，位域，大小端数据转换  c语言：宏定义、函数定义、传值和传指针、重载（c++）与覆盖，命令行参数，函数递归 |

c语言是一种强类型的语言，数据类型决定了数据的内部存储方式和存储大小。

c语言基础的数据类型：char、short、int、long、float、double、（void、结构体、enum、union）

全局变量：指的是具有文件作用域的变量，默认初始化为零。

局部变量：定义在函数内的变量，随机初始化，作用域仅在函数结束前。

静态变量：static修饰的，分为全局静态和局部静态变量，全局静态限制作用域为本文件不可extend。局部静态变量的作用域为函数被调用到程序结束。

常量：使用const 或者 constexpr（vs2015开始支持）修饰的变量，或者是使用预处理宏定义的量，这种变量只读不可写。Const修饰为只读，但数据可能改变，constexpr为常量，推荐区分。

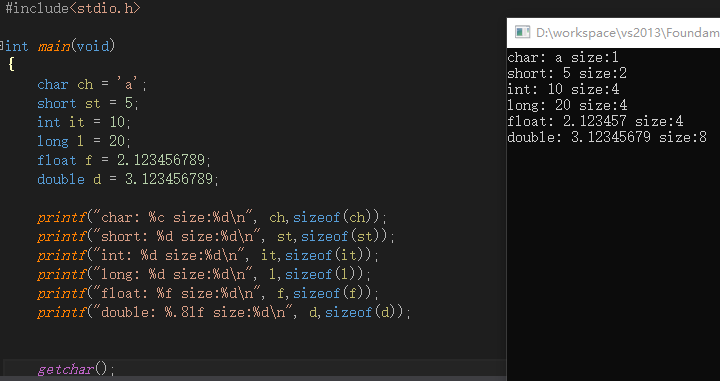
基本数据类型的格式化输出：（整数）o八进制 x，X十六进制 d十进制 %smd域宽和填充为s，m为输出宽度

（浮点数）f精确到小数点后6位 lf精确到小数点后16位，默认6位

c字符 s字符串 p指针 %%一个百分号 u十进制无符号整数 e，E指数 g，G以f和e中宽度小的输出 标志- 右对齐 标志+输出正负号 标志空格 输出空格（正数）负号（负数）

#o x e,g,f会输出前缀给出小数点

展示输出数据内存大小和格式化输出：

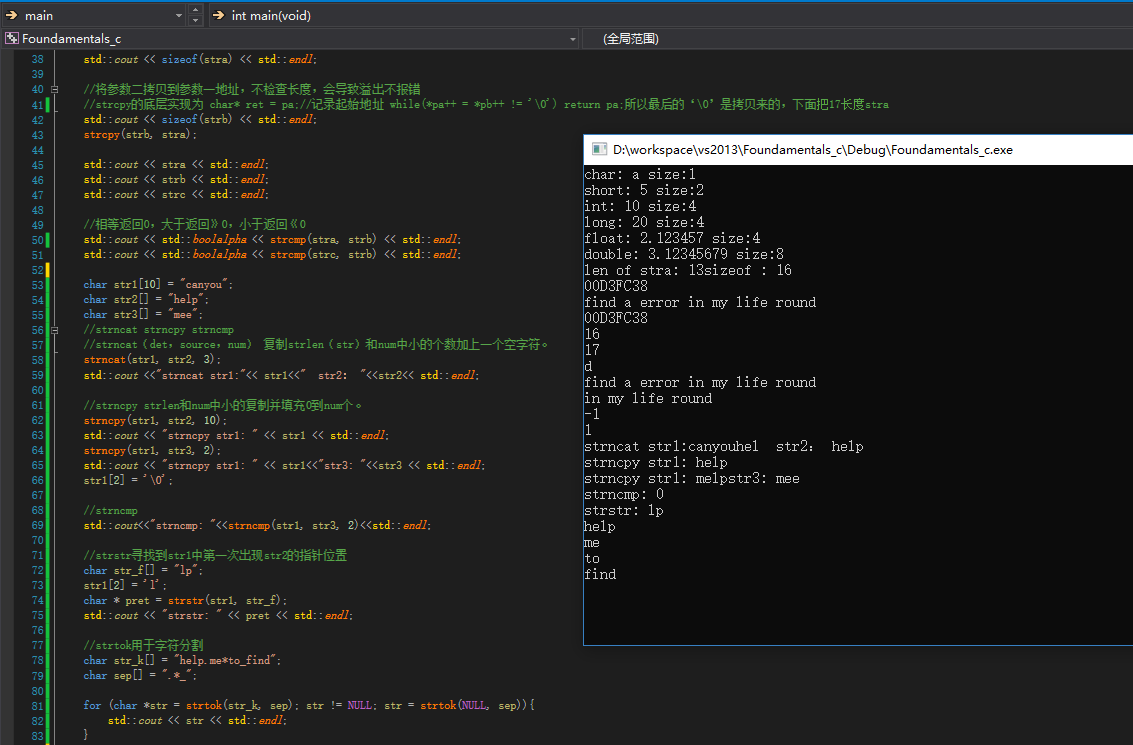


字符串处理：//参考https://m.jb51.net/article/247795.htm

输入输出：printf和scanf, puts和gets stdio/cstdio头文件中

其他：strlen，strcat，strcpy，strcmp，strncat,strncpy,strncmp,strstr,strtok等在string.h/cstring头文件中。注意事项见代码注释。

结果：



内存对齐规则：（目的是减少取地址的次数，以内存换性能）

alignof和sizeof运算符可以获得对齐值和内存大小，对于基本数据类型返回值相同，对于结构体和类不同。alignof会返回成员数据中最大的值，而sizeof会返回它们的和。

指定对齐值：

#pragma pack(k) 对齐值为k，默认值和平台相关。

**最终有效对齐值为自身对齐值和指定对齐值中较小的值**。**最终存储起始地址%k == 0.**

Struct的对齐值是成员中对齐值最大的值。例如类里面有结构体，且结构体为最大成员对齐值，则类的对齐值为结构体对齐值。

**对齐的三个原则：**

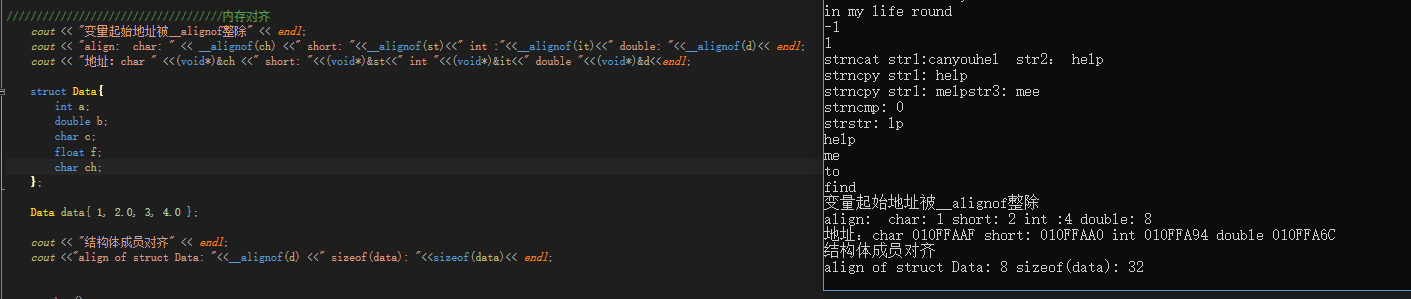
1. **变量的起始地址能够被对齐值整除。**
2. **结构体成员变量相对起始地址的偏移能够被自身对齐值整除，如不能则在前面填充字节。**
3. **结构体整体大小能够被最宽的成员大小整除，如不能则在后面填充字节。**

**另一种针对结构体的三个原则描述：**

1. **结构体变量的首地址能够被其最宽成员大小与对齐系数的较小者整除。**
2. **成员偏移量被成员对齐值和对齐系数较小者整除。不足前面填充。**
3. **结构体总体大小为最宽成员大小与对齐系数的较小者的整数倍。不足后面填充。**

结果：

结构体的



位域：参考https://wk.baidu.com/view/2466430dcf9b069dc5022aaea998fcc22bcd14361

1. 位域的定义和声明。

位域声明struct 位域名{类型说明符 位域名：位域长度} 其中类型只能是整型，（signed/unsigned）char short int long.

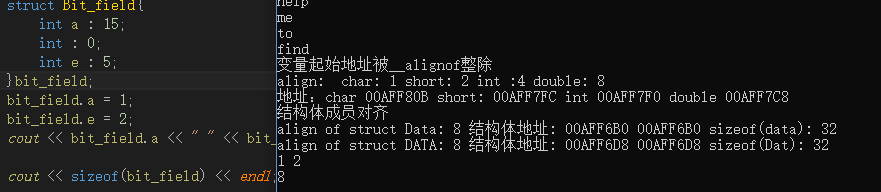
1. 位域的要求。
2. 如果当前数据类型不够存放下一个定义的位域时候，编译器会自动将其拓展到下一个数据类型区域，所以，为了可读性应该显式地使用空域将其调整到下一个数据域。

Struct{

Int a:4;

Int :0; //如果前面刚好满一个数据类型则没有任何效果，否则跳到下一个数据类型区域的开始

Int b:4;//b将会放到下一个int的开始，所以sizeof返回8；



1. 位域的长度不能够大于数据类型的大小，比如int为32bit，类型的位域最大长度为32.
2. 位域可以有无名，用来填充和调整位置，或用长度为0的调整。

（问题）内存中的位域是按照（大小端模式）从数据区域中顺序连续存放。

实际测试中，double的起始地址是按照4对齐的，包含double的结构体的起始地址也是4的倍数。但是在计算内存大小（sizeof）和结构体内对齐时对齐值又是按照8进行的。

及计算起始地址时是4，计算对齐值对齐时是8。（个人在win32下的测试结果不严谨）。

**位域的对齐方式**和结构体有一点不一样，**即相邻的位域会长度和小于sizeof数据类型则会在同一个数据类型中存放，直到不能存放，调整到下一个数据类型区域开始**。数据类型的对齐方式依然遵循结构体的偏移为数据类型对齐值的整数倍的规则，结构体整体大小为最宽数据类型的对齐值和指定对齐值小的整数倍。对于**不同的数据类型之间，各个编译器实现不同，在vc中采用的是不压缩的方式**，在gcc和devc++中采用的是压缩的方式（位域字段中间穿插了非位域字段则不进行压缩）。

结果：



大小端数据转换：

大端模式：数据的高位放在低地址，低位放在高地址中。和阅读方式相同。

小端模式：数据的低位放在低地址，高位放在高地址中。地址和权值大小对应。

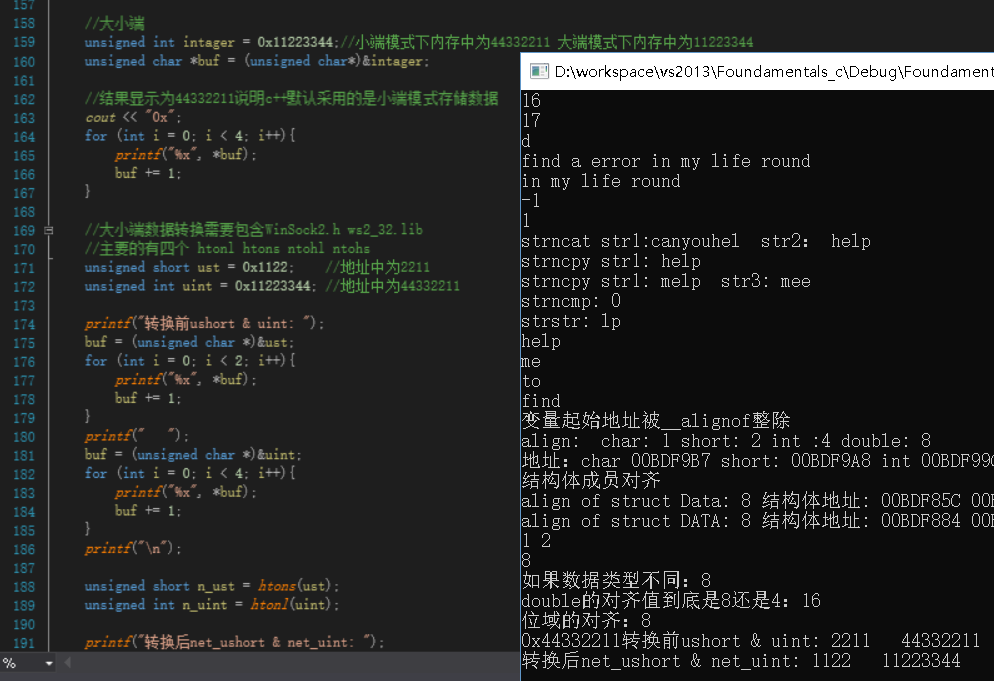
一般的系统多用小端模式，mac os采用的是大端。

网络传输：网络传输为了统一，采用的是网络字节序，即大端模式。

Windows下的大小端转换函数在WinSock2.h头文件，并且导入ws2\_32.lib库。

转换的函数主要有四个实际两个为转换16位数据，另外两个转换32位的数据，htonl,htons,ntohl,ntohs.

结果：

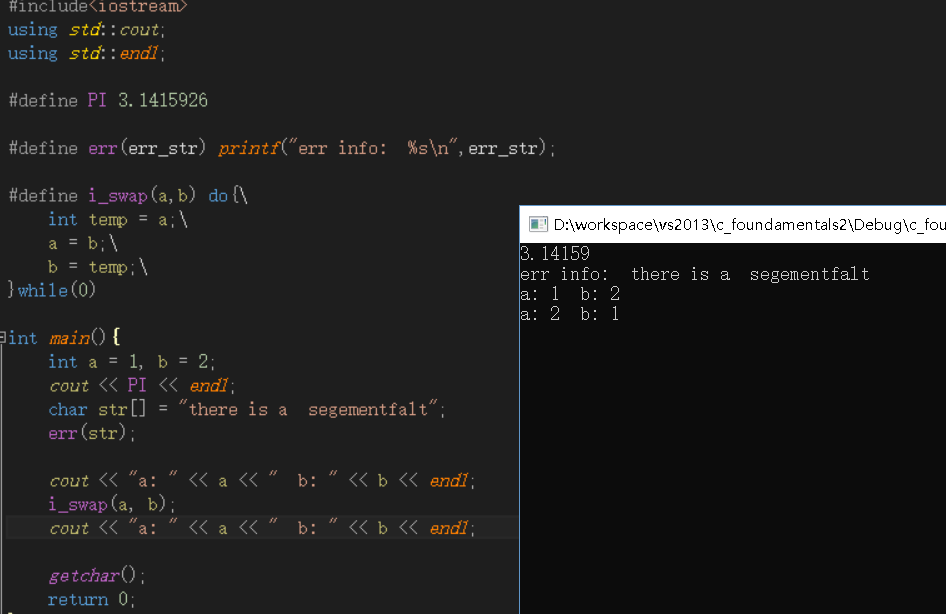


宏：可分为对象宏和函数宏。

1.对象宏用于定义常量。形式为#define 宏名 值

2函数宏，如果在宏名后面加上参数即为函数宏，实际是对参数的替换。

在c++中也可以用常量和内联函数替换这两种做法。

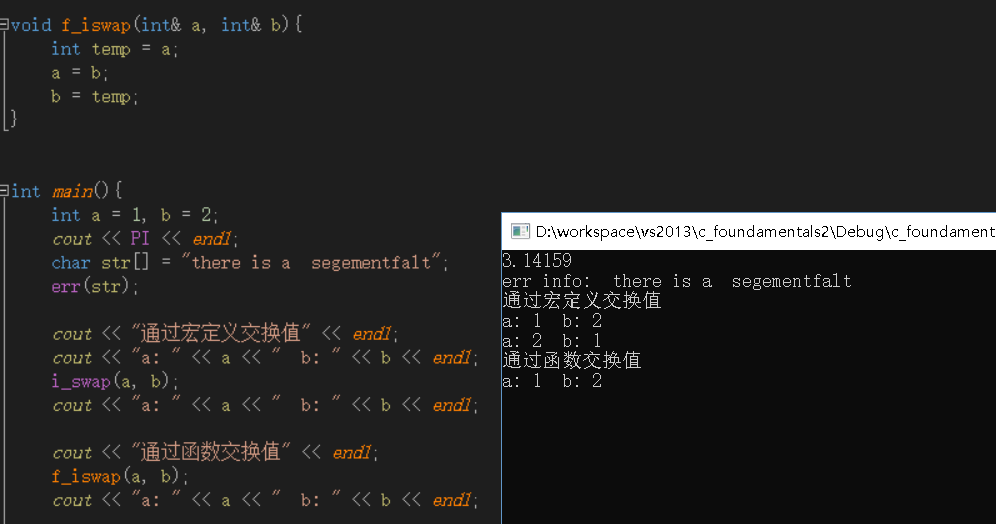


函数定义：一个完整的函数定义包含函数名称、形式参数列表、函数体。

函数声明：c编译器编译是根据文件顺序编译的，所以如果调用时函数还没有实现，就无法调用，声明函数编译器就可以识别，从后面去找它的实现，函数的定义是通过参数传递是通过值拷贝实现的，参数为函数内的局部变量。

函数声明一般以头文件提供，而实现放在源文件中，防止简单的重复定义错误。重复定义错误有两种类型一种是一个源文件（目标文件obj）中重复包含了头文件中的实现；另一种是不同的源文件（目标文件obj）中包含了同一符号的定义。前者可以通过#pragma once 或者#ifdef endif条件编译防止重复包含解决。后者则不可以，防止后者的方式是不要在头文件中定义函数或变量，仅仅进行声明即可。具体的定义放到唯一一个源文件中，保证定义唯一。

错误出现的两个环节分别为编译环节和链接环节。



传值和传引用或者指针。

函数参数默认传递的是值，进行值的复制，无法更改外部的变量值，仅能通过返回值的方式交互，这种方式使用复杂，比较时候可以直接使用指针和引用修改外部变量的值。

重载（c++）和覆盖（多态的两种方式）：

重载和重写、覆盖的区别：

重载（overload）：

1. 重载在同一范围，例如同为全局，或者同为成员函数。
2. 参数不同（类型和顺序）
3. Virtual可有可无

覆盖（override）：（能多态要求**1.虚函数2参数相同**）

1. 是子类和基类同名同参函数的不同实现。
2. 参数相同。
3. 基类函数需要virtual关键词，基类函数为虚函数，子类对应函数自动转化为虚函数。

隐藏/改写（overwrite）：（**不能进行多态的情况**）

1. 子类和基类
2. 若参数不同，则基类函数被隐藏；若参数相同，且基类没有virtual关键词，则隐藏。（有参数则被覆盖）。

**以上的说法是标准定义（包含了隐藏规则），但是不好记**。

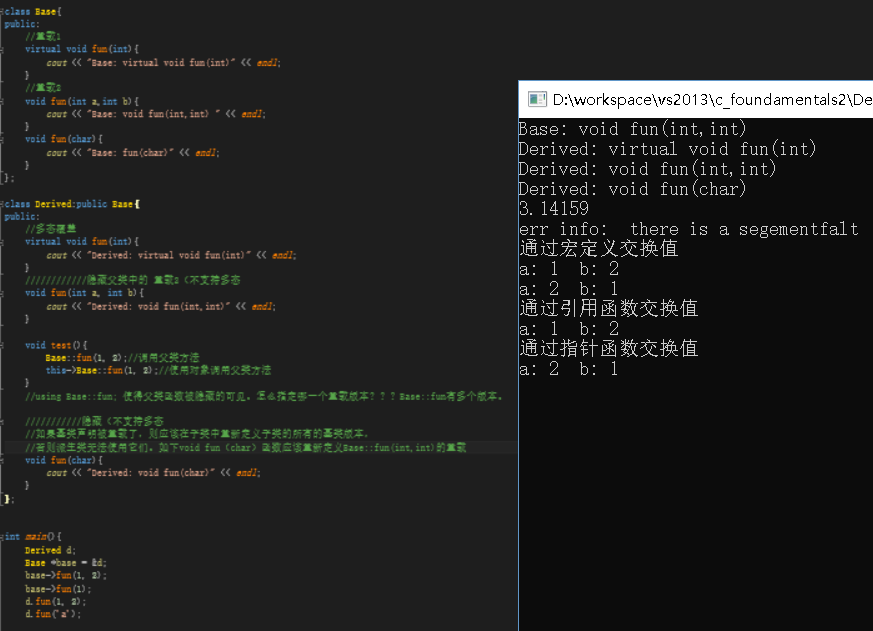
**只需要记住虚函数才支持多态性，多态函数的覆盖要求参数完全相同**。

**重载是在同一个类或者范围；而隐藏/改写是在基类和子类之间，子类和基类函数同名，函数不同一定是隐藏，相同基类不是虚函数也是隐藏，是虚函数则是覆盖。**

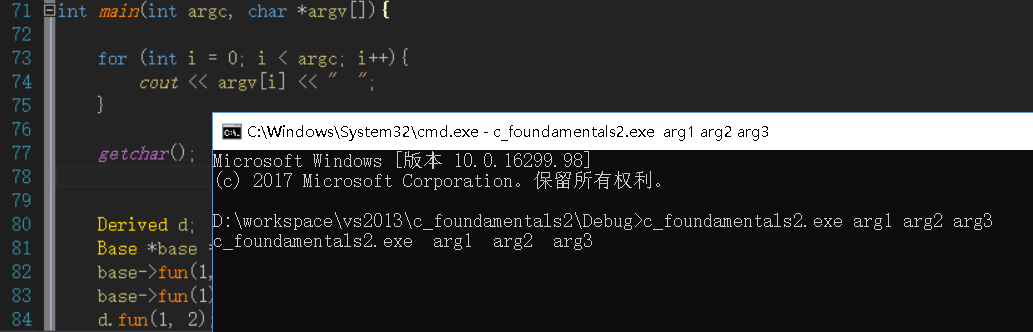
如果不是多态调用调用时看的是指针本身的类型对应的函数（对象模型进一步理解原理），例如

Base \*base = &derived；base->fun()调用的是Base的函数而非子类函数。

结果：



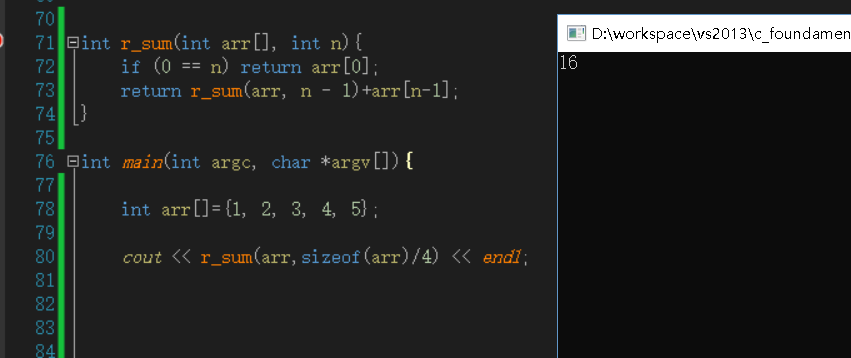
命令行参数：



函数递归：

函数递归即函数里调用自身，利用堆栈逻辑简化逻辑设计，符合人的逻辑思维习惯，但是需要保证不能递归过深，到时堆栈溢出。

结果：简单的例子是递归求数组和。



|  |  |
| --- | --- |
| 时间 | 学习内容 |
| 7月18日 | C++：输入输出流、string、map、vector、list、set |
| C++：类的封装、构造、析构、拷贝构造、复制赋值、重载操作符、类继承、虚函数、类型转换 |
| C++：引用与指针，多维数组，命名空间，异常 |

一、输入输出流:

1.c语言：

（1）

int getchar(void);

int putchar(int c);

char\* gets(char \*str);

int puts(const char \*str);错误返回EOF

int printf(const char\* str,...);

int scanf(cosnt char\* str,...);

FILE\* fopen(const char\* filename,const char\* mode);

Mode:

r 读

w 写

a 添加

rb 二进制读

wb 二进制写

ab 二进制添加

+ 表示可以进行读和写

r+ 读写打开

W+ 读写打开

A+ 读写附加

R+b w+b a+b 二进制形式

文件打开失败fopen返回空指针。

Int fclose(FILE\* fp);返回0 成功

Int putc(int ch,FILE\* fp);

Int getc(int ch,FILE\* fp);

Fgetc 同getc

Int fgets(const char\* str,int len,FILE\* fp);

Int fputs(const char\* str,FILE\* fp);

Fseek() 定位，

ftell返回当前文件位置

Fprintf()

Fscanf()

Int feof(FILE\* fp);

Int ferror(FILE\* fp);返回是否文件操作出错

Void rewind(FILE\* fp);文件指针到起始位置

Int remove(const char \*filename)清除文件，成功返回0，失败非零

Int fflush(FILE\* fp);清空输出流的内容，输入缓冲区写入文件，fp为空时特殊。

二进制读写：fread 和fwrite 文件打开模式需要有b

Size\_t fread(void\* buffer,size\_t size,size\_t count,FILE\* fp);size每个数据的大小，count数据块个数，总共读写了size\*count个字节数据。

Size\_t fwrite(void\* buffer,size\_t size,size\_t count,FILE\* fp);

最常用的是可以读写结构体类型的数据到二进制文件中。

两个函数返回读写成功的块数，count，如果比count小：fwrite写入错误，用ferror检测。Fread()可能读到了文件尾或者读错误，用feof和ferror检测。

Int fseek(FILE\* fp,long int size,int origin);

size移动的距离，origin ：起点模式（三种）SEEK\_SET开始 SEEK\_CUR当前 SEEK\_END结尾

Long int ftell(FILE\* fp);返回当前文件指针位置，失败返回-1

**代码规范中禁止使用非流类输入输出，不做具体测试**。

1. c++流类库

对象化的IO库。

/\*

->ifstream/istringstream

->istream

ios\_base->ios ->(继承istream+ostream)iostream ->fstream/stringstream

->ostream

->ofstream/ostringstream

streambuf ->filebuf/stringbuf

\*/

类的继承体系如上。

Std中有四个常用的流对象，cin，cout ，cerr，clog标准输入、输出、错误、日志。

输入流常用函数：

Read(char\*,int n)

Get();get(char& ch);get(char\*,int size,int delim)

Getline(char\* buf,int size,[delim])

Gcount()返回上一次输入操作读取的字符数

Ignore(int n,char eof);读取并忽略n或者直到eof字符（包括例如换行符）

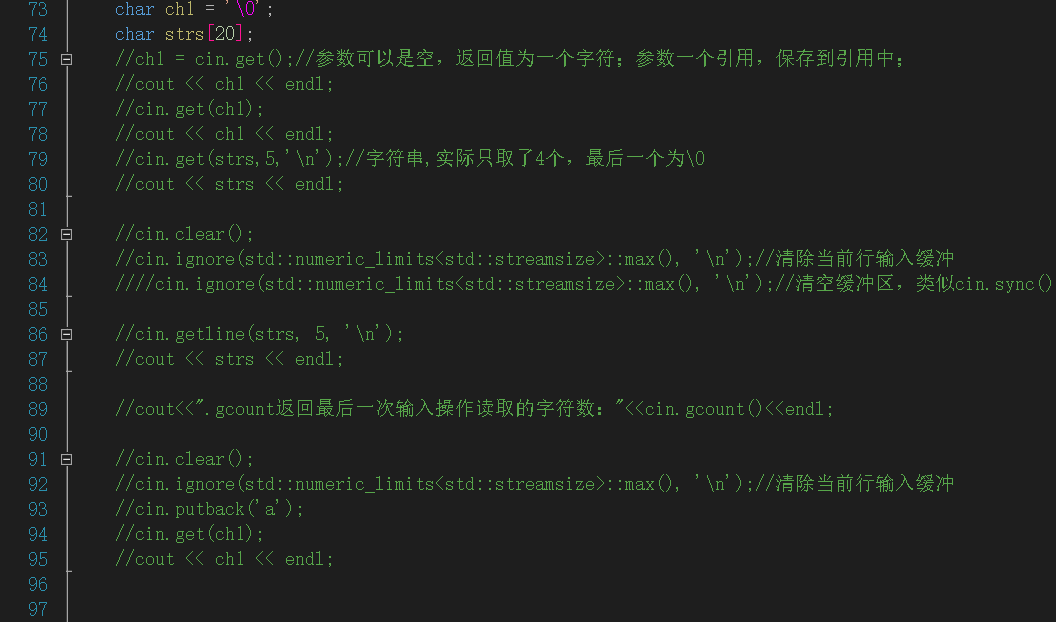
Putback(char ch)在输入流后面放入一个字符

Peek()函数返回输入流中下一个字符但字符任然留在输入流中，流指针也不移动。

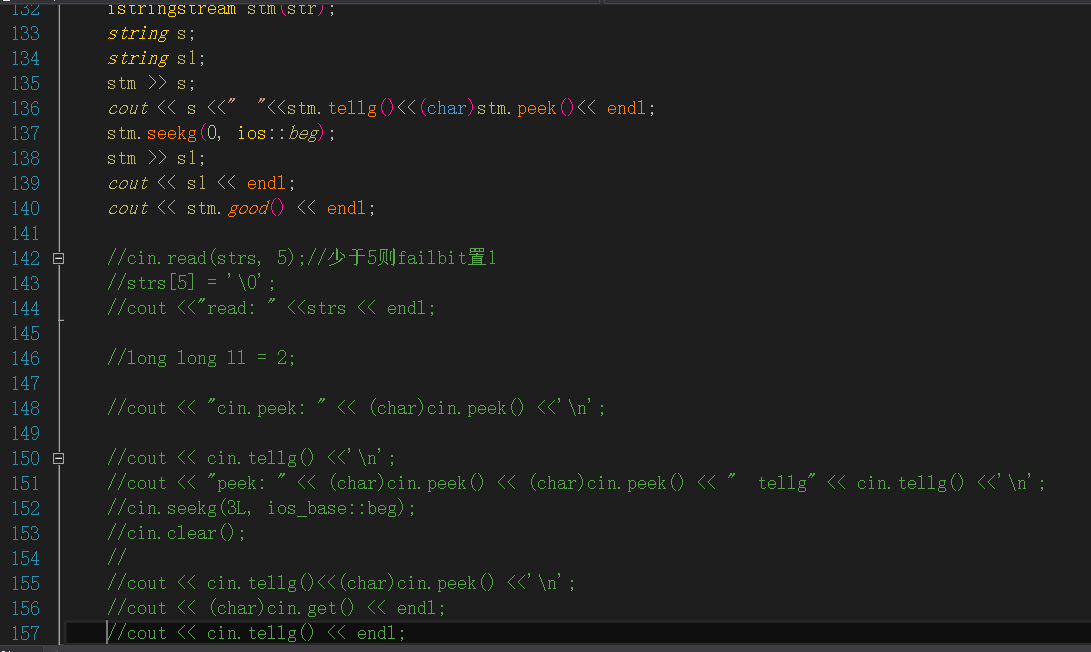
Seekg（pos）/(off,dir(beg/cur/end))将输入流指针移动到指定位置或者是偏移从beg/cur/end

Tellg()返回当前流指针相对于开头beg的位置。

结果：







输出流常用函数：

Freopen(char \* filename,char\* mode,stdin/stdout/FILE \* p\_file);文件重定向

例子：

Freopen(“1.txt”,”r”,stdlin);输入重定向为文件

Freopen(“1.txt”,”2”,stdout);输出重定向为文件

无格式（char 字节）输出：

Write(const cahr\* ch,int n);//无格式块输出

Put(char ch);//无格式字符输出

Int Isdigit(int c);可以判断参数是否代表数字

Flush()将输出缓冲写入磁盘

寻位：

Seekp(off,origin);

Tellp();

错误信息：

Rdstate()返回错误信息

Setstate(iostate)

Bad() eof() fail() good();

Clear(iostate flags = ios::goodbit）清除错误标志

格式化输出：

Copyfmt(ios& other) 复制格式化信息

Char Fill（[char ch]） 控制填充字符,返回填充前的值，改为ch

Flags([fmtflags flags]) 返回先前格式信息，改为flags格式

Setf（flags, [mask]） 设置flags，或是按照mask遮罩下的flags

Unsetf(flags) 反设置flags标志

Fmtflags标志常量有

Dec 十进制

Oct 八进制

Hex 十六进制

Basefield dec|oct|hex .适合掩码运算mast参数，比如反至所有进制位unsetf(basefield)

Left 左对齐，填充字符在右

Right 右对齐，填充字符在左

Intenal 填充字符在中间

Adjustfield left|right|intenal 用于掩码运算

Scientific 用科学计数法表示浮点 若组合fixed则用十六进制法表示

Fixed 用定点记法表示浮点 若组合scientific则用十六进制表示

Floatfield scientific|floatfield 用于掩码运算

Boolalpha 用字符表示bool

Showbase 为整数输出基数的前缀

Showpoint 无条件为小数点生成小数点字符

Showpos 为非负值生成+符号前缀

Skipws 在具体操作前跳过前导空白符

Unitbut 每次输出后刷新输出，即无缓冲方式输出

Uppercase 输出中大写模式

格式控制符

许多和fmtflags相同，取消加前缀no，

特有的<istream>

Ws 消耗空白符，用法，cin>>ws;

Ends 输出‘\0’

Flush 刷新输出流

Endl 换行加刷新

《iomanip》

Resetiosflags()清除指定标志

Setiosflags()设置标志

Setbase() 设置基数

Setfill() 更改填充字符

Setprecision 更改浮点精度

Setw 更改域宽

Openmode:

App : 每次写入强寻位到流结尾

Binary: 二进制模式

In: 读

Out: 写

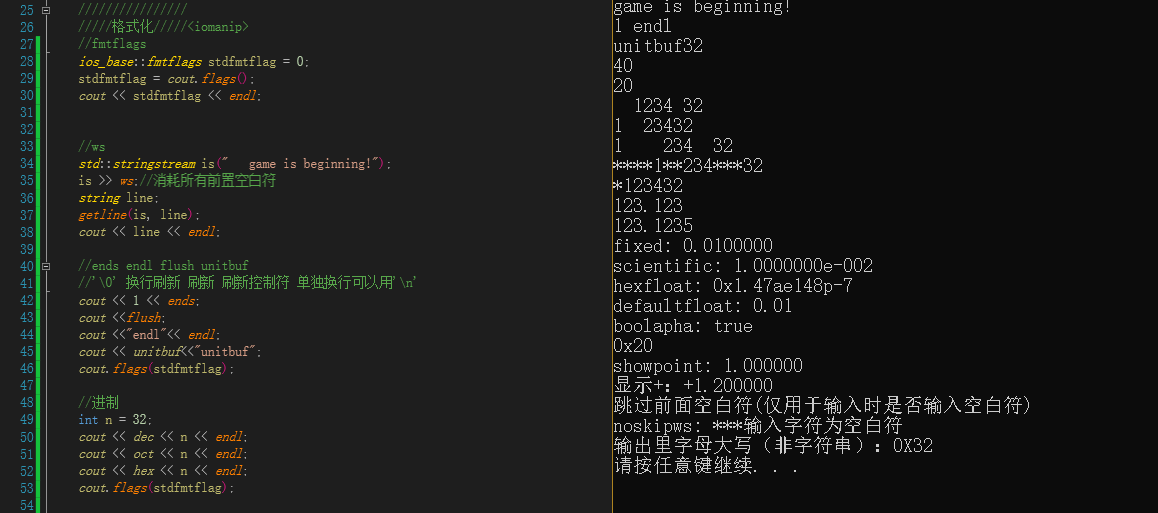
Trunc: 舍弃流内容

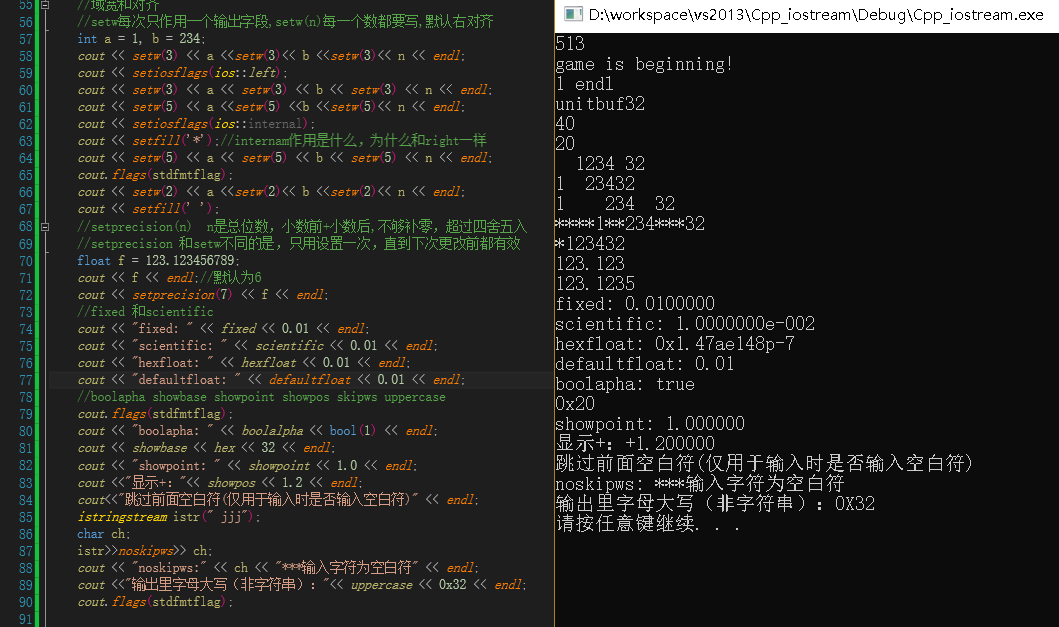
Ate: 文件打开后立即寻位到流结尾

Seekdir: beg 流开始 cur 流当前 end 流结尾

结果：







文件流：参考http://www.cppcns.com/ruanjian/c/318100.html

Ifstream

Ofstream

Fstream

派生自istream/ostream/iosstream 所以支持常用的上面的流的用法：输入输出，格式控制等。

特有的函数：

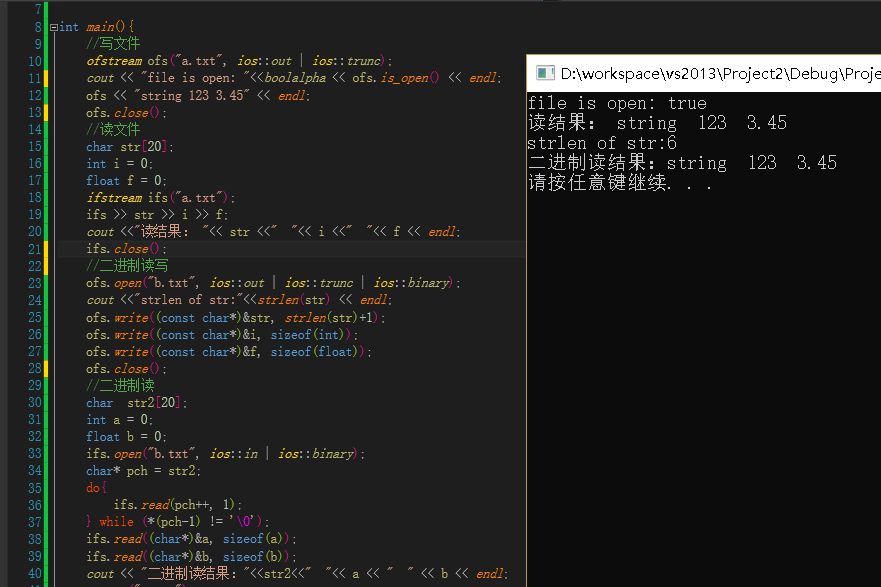
Isopen() 等效调用 rdbuf()->is\_open().

Open(char\*/string,open\_mode)

分别等效rdbuf()->open(filename,mode)/rdbuf()->open(filename.c\_str(),mode)

Void Close(void) 出错设置failbit 关闭文件，等效rdbuf()->close()

结果：





//如果是完整的文件读取采用如下结构

// while(1){

// ifstream.read(data);//读

// if(ifstream.eof()) break;

// data确认有效，才保存读取数据

//或者如下写

/\*ifstream.read(data);

while (!ifstream.eof()){

data; 确认有效，保存读取数据

ifstream.read(data);

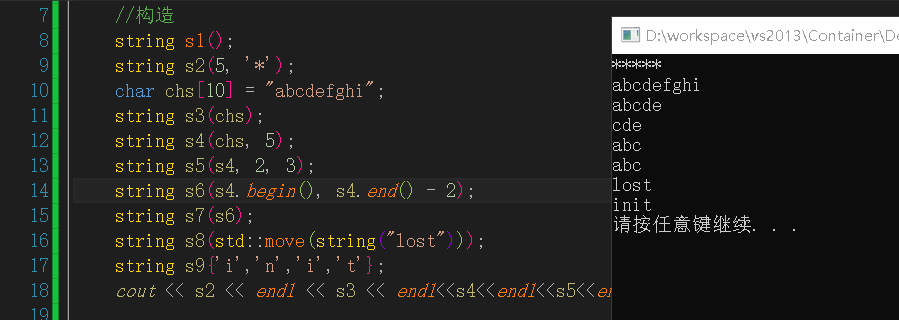
}\*/

应为eof当文件读完后并不会设置为true,只有再读一次，读到文件尾才设置，并且读取无效。

标准库容器：string、map、vector、list、set

**String:**

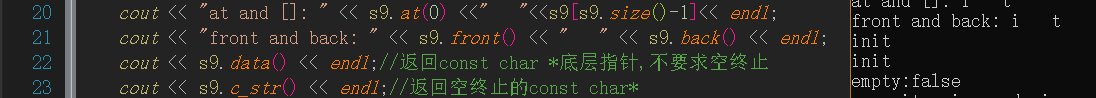
（1）构造：



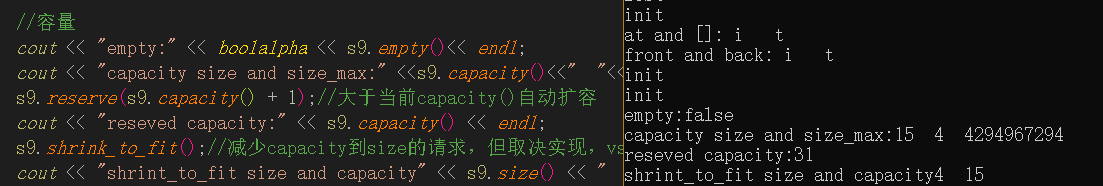
operator= 除去模板有五种方式：引用、移动、字符串、字符、初始化列表。

assign() 和一样有除去空的和模板有八种方式，和构造一模一样。

1. 访问：at[有边界检查]、 operator[]、front、back、data、c\_str



1. 容量 empty size max\_size reserve capacity shrink\_to\_fit resize(数据个数更改)



1. 操作

Erase()

Clear() 通过erase(begin(),end()),移除所有字符

Insert() 插入字符，字符串，string，initialer\_list,

Pop\_back(); 删除尾部字符

Push\_back();尾部插入字符

Append() 相当于insert没有位置参数（尾部插入）

+= 尾部插入字符，string,字符串，initialize\_list（都是完整的）

Compare()

可以分成两种，一种是和string比较，另一种是和c字符串比较

每一种都有三种：

Compare(str&/c\_str);

Compare(pos,count,str&/c\_str);

Compare(pos1,count1,str&,pos2,count2)/Compare(pos1,count1,c\_str,count2);c字符串没有起始范围，除模板共计六种

Replace() 分成迭代器和整数类型指定位置

替换Rnge表示其中一种:（pos,count）/[it\_first,it\_last)范围的字符，两种方式指定被替换范围基本都可以

Replace(Rnge,string);两个函数

Replace(pos,count,str,pos2,coun2)

Replace(itf1,itl1,itf2,itl2) 这两个是一对

Replace(Rnge,c\_str,[count]);四个函数

Replace(Rnge,count2,ch);两个函数

Replace(it\_first,it\_last,initializer\_list);

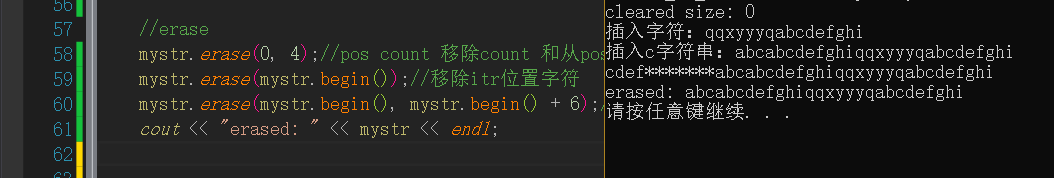
Substr(pos=0,count = npos);没有重载函数,默认拷贝全部

Copy(char\* dest,count,pos = 0);弃用

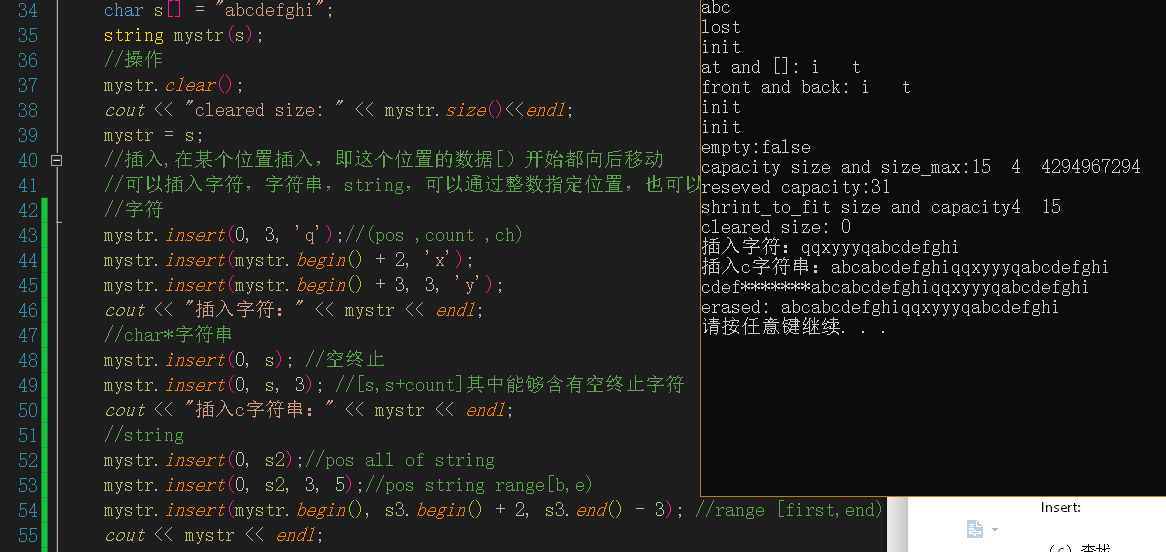
\_Copy\_s(char\* dest,des\_count,count,pos);

Swap(sting & other);没有重载，也没有异常抛出

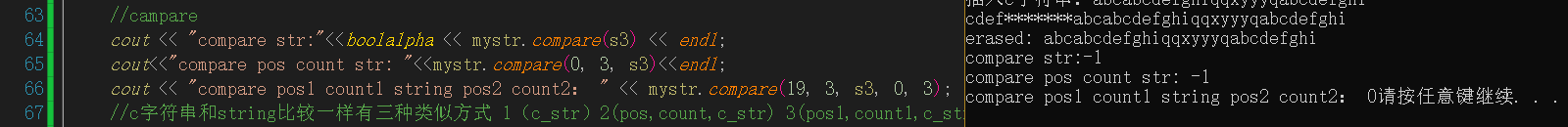
Erase:



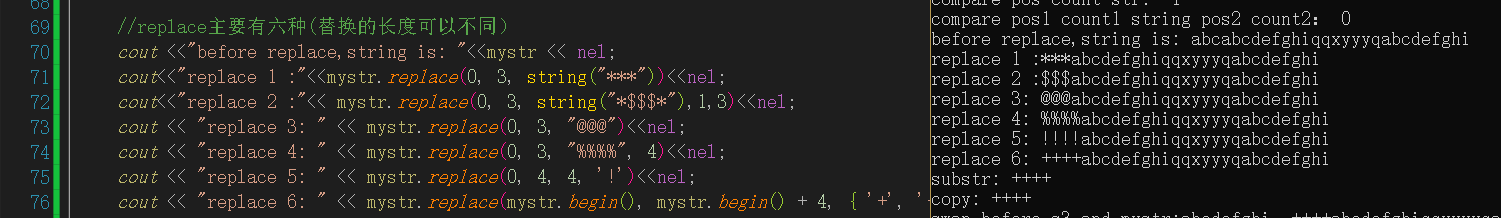
Insert:



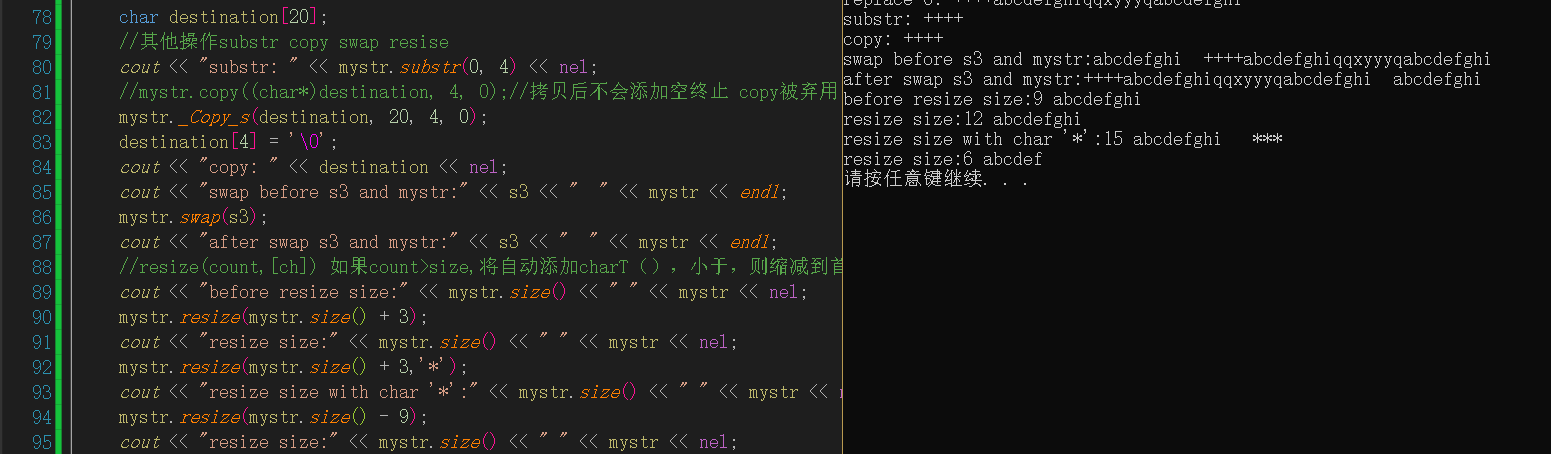
Compare:



Replace:



其他：substr、copy、swap、resize



1. 查找

Find rfind find\_frist\_of find\_first\_not\_of find\_last\_of find\_last\_not\_of

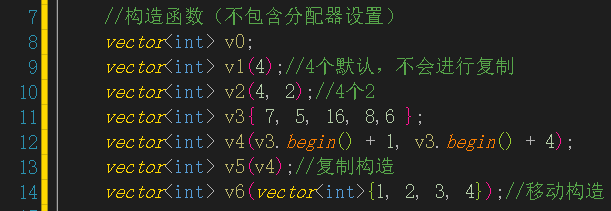
分成两类一类是查找首次出现，默认从pos=0开始查

另一类是查找最后一次出现或缺失（带not）带fist和last的为查找字符(参数为string和char\*时，其中的任意一个字符)，不带为子串（find/rfind）。



Vector:

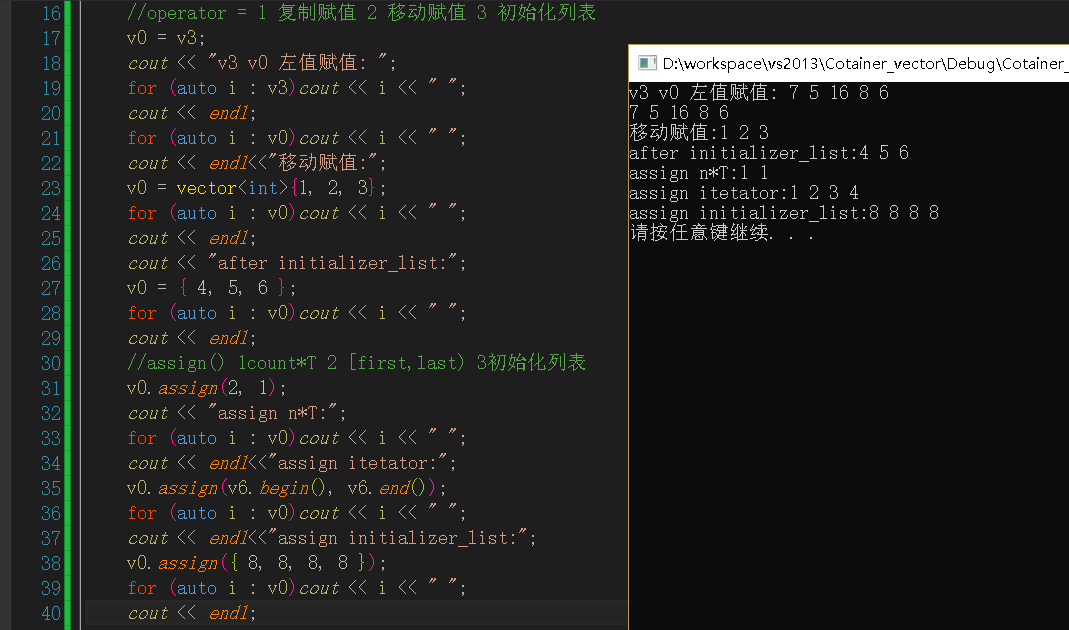
//构造函数（不包含分配器设置）



operator = 1 复制赋值 2 移动赋值 3 初始化列表

assign() 1count\*T 2 [first,last) 3初始化列表

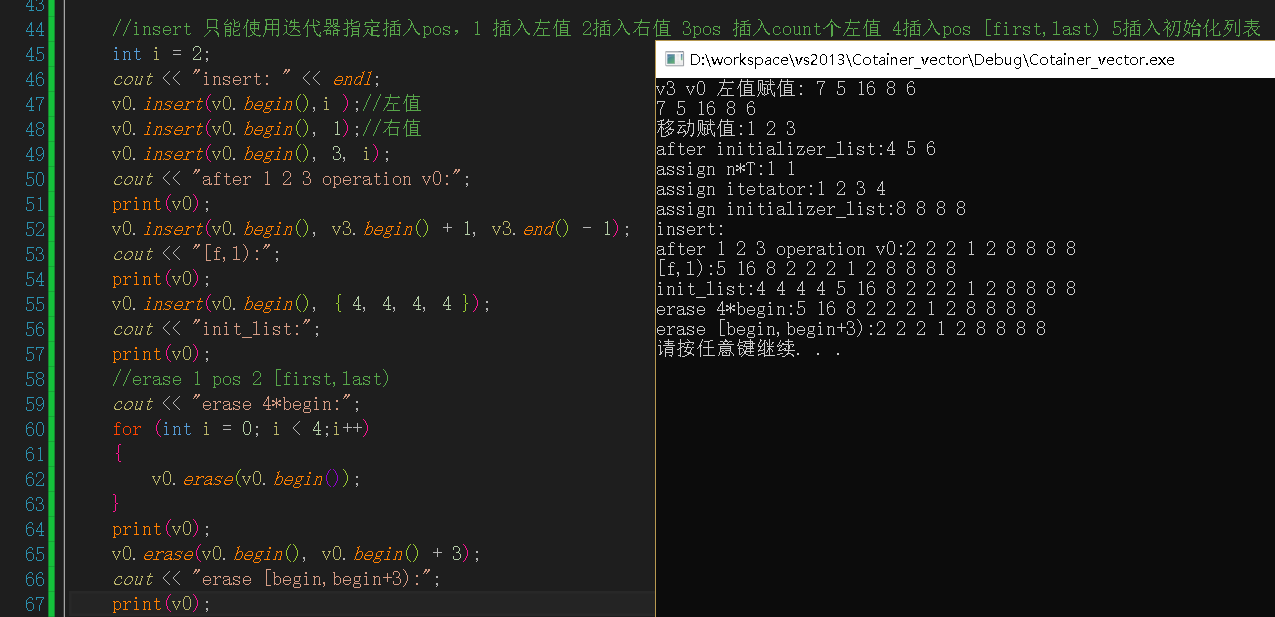
= and assign：



insert 只能使用迭代器指定插入pos，1 插入左值 2插入右值 3pos 插入count个左值 4插入pos [first,last) 5插入初始化列表

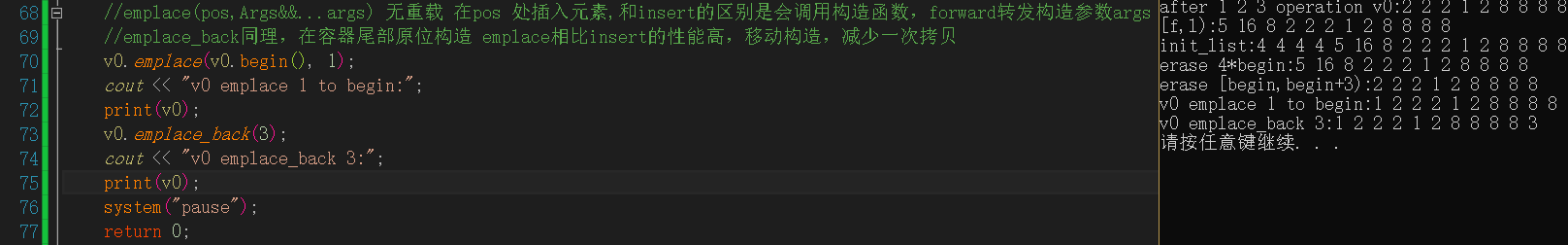
erase 1pos 2 [first,last)

Insert and erase:



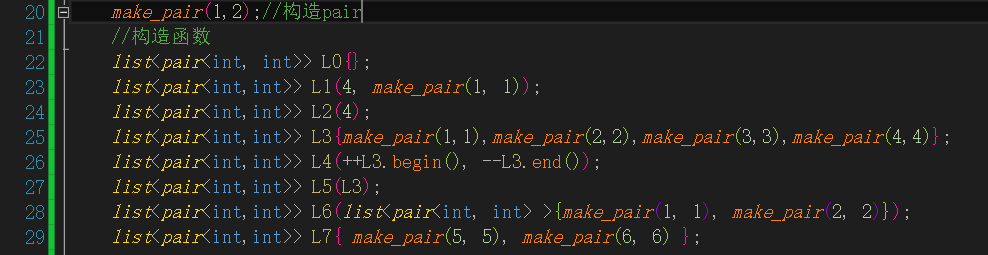
emplace(pos,Args&&...args) 无重载 在pos 处插入元素,和insert的区别是会调用构造函数，forward转发构造参数args

emplace\_back同理，在容器尾部原位构造



**List:**

构造函数:



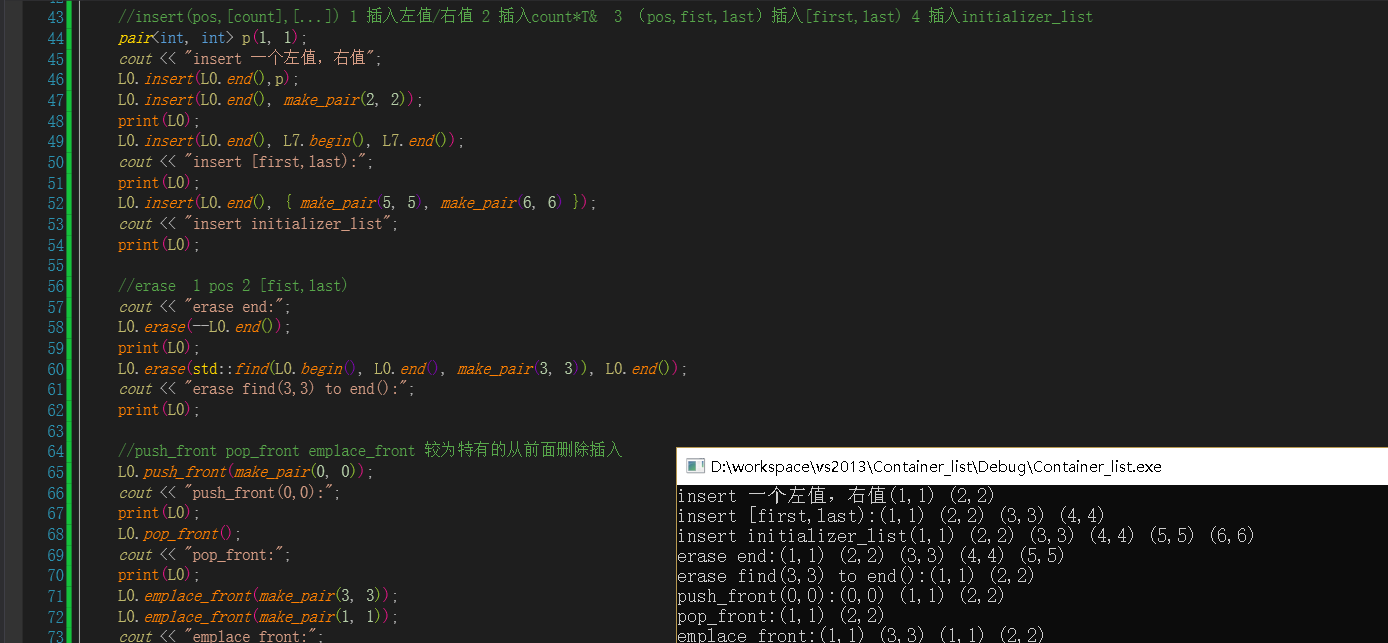
insert(pos,[count],[...]) 1 插入左值/右值 2 插入count\*T& 3 （pos,fist,last）插入[first,last) 4 插入initializer\_list

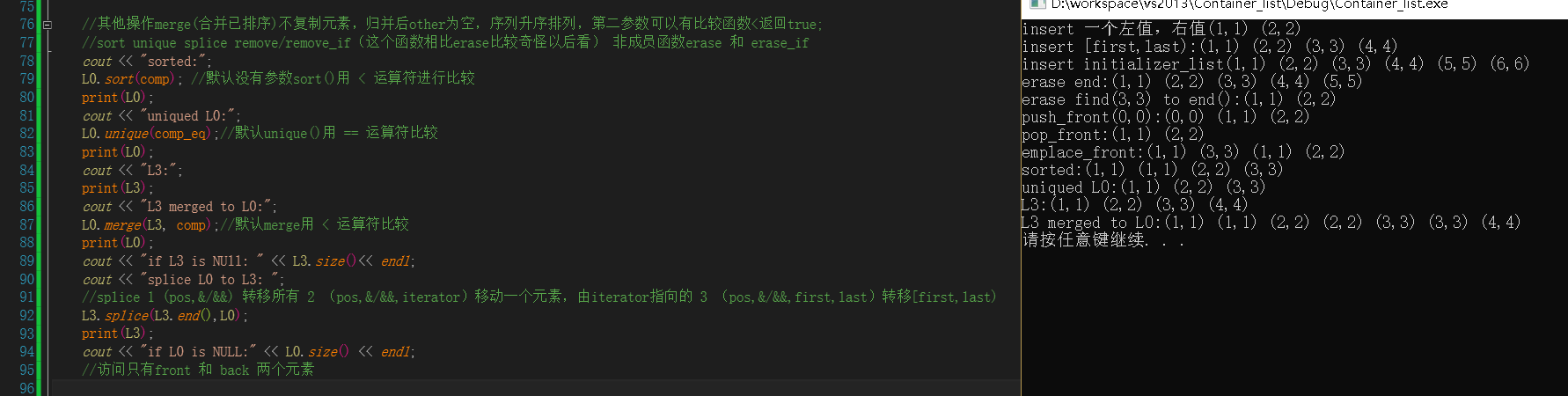
erase 1 pos 2 [fist,last)

push\_front pop\_front emplace\_front 较为特有的从前面删除插入

其他操作merge splice remove/remove\_if unique sort 非成员函数erase 和 erase\_if

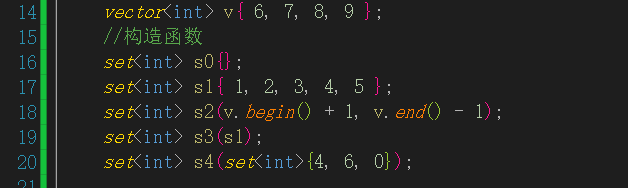
结果：





**Set:**

构造函数:



Insert:

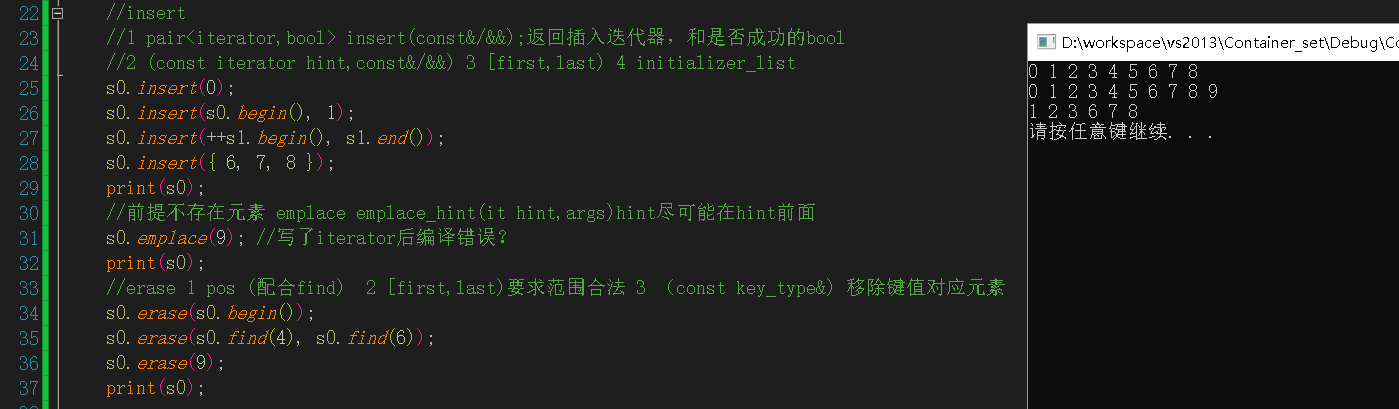
1 pair<iterator,bool> insert(const&/&&);返回插入迭代器，和是否成功的bool

2 (const iterator hint,const&/&&) 3 [first,last) 4 initializer\_list

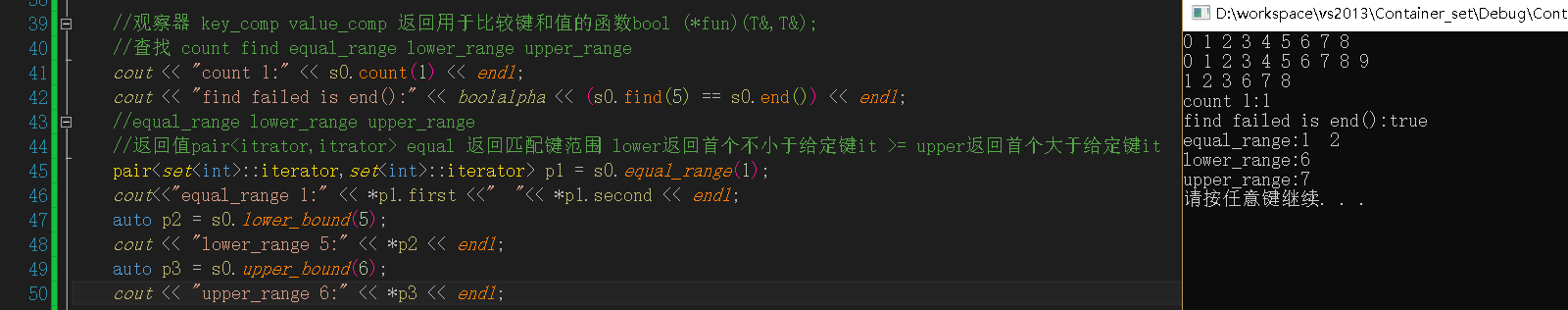
前提不存在元素 emplace emplace\_hint(it hint,args)hint尽可能在hint前面

erase 1 pos (配合find) 2 [first,last)要求范围合法 3 （const key\_type&) 移除键值对应元素

观察器 key\_comp value\_comp 返回用于比较键和值的函数bool (\*fun)(T&,T&);

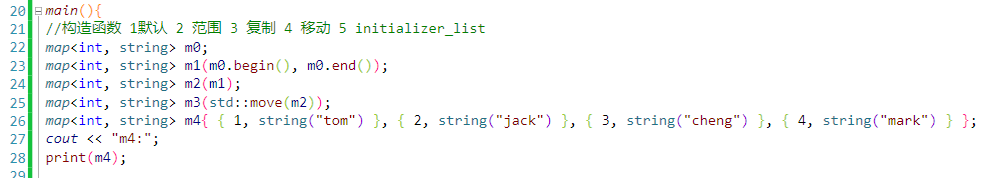


查找 count find equal\_range lower\_range upper\_range



**Map:**

构造函数 1默认 2 范围 3 复制 4 移动 5 initializer\_list



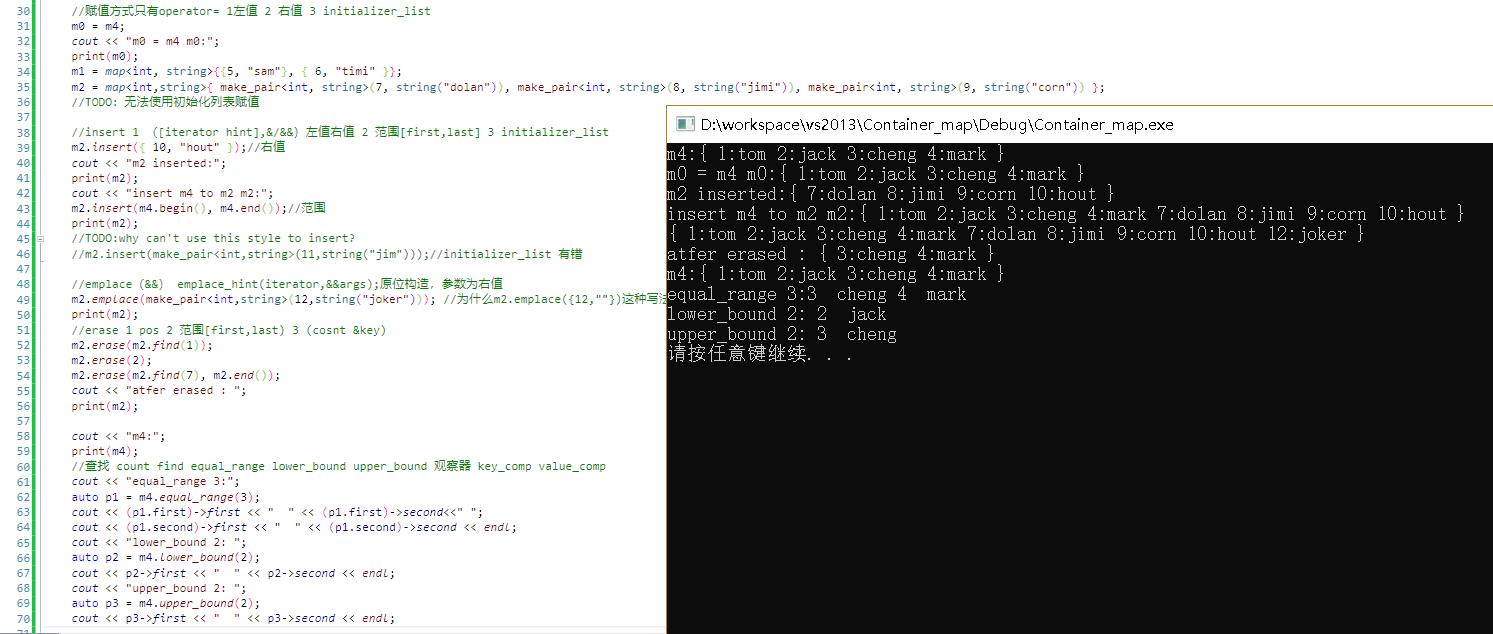
赋值方式只有operator= 1左值 2 右值 3 initializer\_list

insert 1 （[iterator hint],&/&&）左值右值 2 范围[first,last] 3 initializer\_list

emplace（&&） emplace\_hint(iterator,&&args);原位构造，参数为右值

erase 1 pos 2 范围[first,last) 3 (cosnt &key)

查找 count find equal\_range lower\_range upper\_range 观察器 key\_comp value\_comp



**C++类**：BV1zT4y1U&HM(如何写好一个类)

内容：类的封装、构造、析构、拷贝构造、复制赋值、重载操作符、类继承、虚函数、类型转换。

封装、类继承

封装：隐藏类的实现细节，给用户提供公共的接口。

1. 类访问控制
2. public:任何地方都可以访问。
3. Private:只有类和友员函数可以访问。
4. Protected:类和派生类内可以访问，但派生类对象不可以。
5. 继承访问控制基本都是public继承即访问级别不变：
6. public继承：派生类保持访问级别不变。
7. Private继承：基类所有成员变为private.
8. Protected继承：public、protected->protected. private维持不变。

总结就是：对象只能访问public权限，private只在类一个层级（基类派生都不可）。

Protected是private可继承的方式，需要继承但是又不是接口的成员声明为protected。



由输出还可以看出派生构造析构顺序，构造先基类后子类，析构先子类后基类。

构造、析构、拷贝构造、复制赋值

特殊的**6个默认成员函数**：

Class X {

X(); //默认构造函数

~X(); //默认析构函数

X(const X & x);//默认拷贝构造函数

X& operator=(const X& x); //默认拷贝赋值函数

X(X&&); //默认移动构造函数

X& operator=(X&&); //默认移动赋值函数

}；

这些函数存在三种情况：（1）未声明（2）编译器隐式声明：defaulted /deleted （3）用户显式声明：defaulted /deleted / user-defined

User-defined**用户声明的四种方式**：

Class X{

X(){}； //声明并定义

X(); //只声明不定义

X()=default; //让编译器生成默认

X()=delete; //阻止编译器生成（成员函数存在，但不可用）

}

未声明和删除的区别：删除会参与重载决策。

假如有变参数构造函数

Template<typename ...Args>

X(Args...args);

未定义默认则 可以声明对象 X x;调用了X::X<>();

如果同时加上定义:

X()=default; X x;调用X::X();

如果定义成：

X()=delete; X x;的编译错误。

**什么情况下编译器会默认生成这些函数**：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **用户声明** | 默认构造 | 析构 | 拷贝构造 | 拷贝赋值 | 移动构造 | 移动赋值 |
| 不声明 | 默认 | 默认 | 默认 | 默认 | 默认 | 默认 |
| 任意构造 | 未声明 | 默认 | 默认 | 默认 | 默认 | 默认 |
| 默认构造 |  | 默认 | 默认 | 默认 | 默认 | 默认 |
| 析构 | 默认 |  | 默认（危） | 默认（危） | 未声明 | 未声明 |
| 拷贝构造 | 未声明 | 默认 |  | 默认（危） | 未声明 | 未声明 |
| 拷贝赋值 | 默认 | 默认 | 默认（危） |  | 未声明 | 未声明 |
| 移动构造 | 未声明 | 默认 | 删除 | 删除 |  | 未声明 |
| 移动赋值 | 默认 | 默认 | 删除 | 删除 | 未声明 |  |

**如何正确写一个类**：

（1）6中函数一个都不写，让编译器自动生成。

（2）6个函数都写。

（3）只可移动不可赋值。有移动函数声明，默认拷贝函数为删除。

{

X();

~();

X(X&&);

X& operator=(X&&);

}

（4）只可拷贝不可移动。不声明移动，默认移动函数为未声明。

{

X();

~X();

X(const X&);

X& operator=(const X&);

}

（5）不可拷贝不可移动。Delete拷贝就行，默认移动为未声明。

{

X(const&)=delete;

X& operator=(const&)=delete;

}

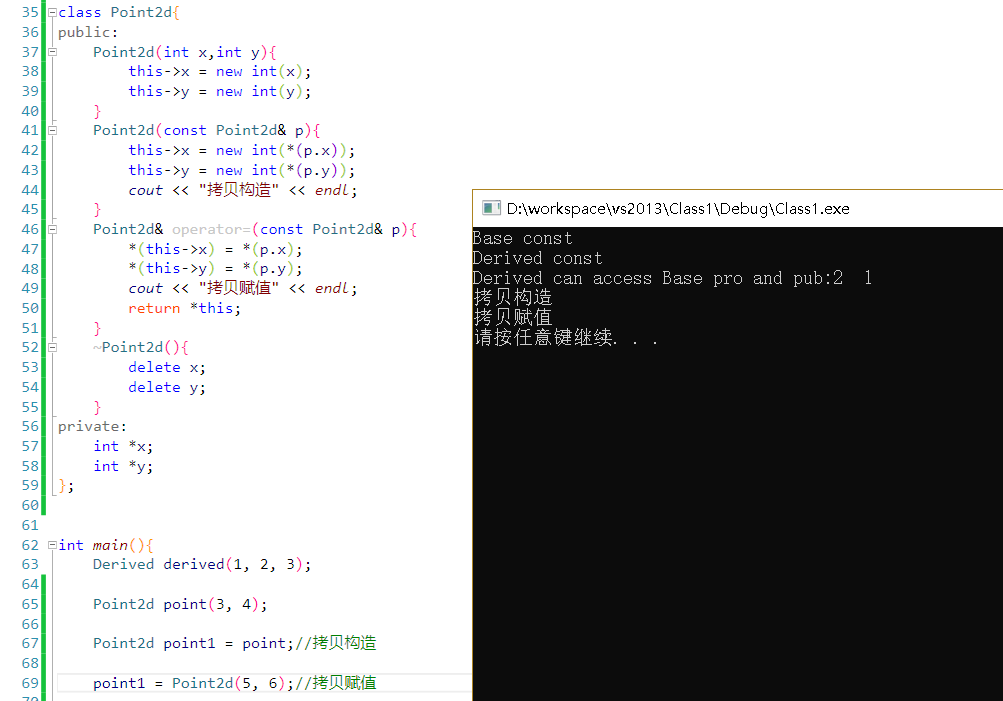
（6）可继承的基类必须有虚析构函数。

1. 对于是否写自己的拷贝和移动如果需要定义

//记录一个临时看到的知识点，

Const作用的全局变量和static作用的变量一样只有local的作用域；

不同之处在于可以通过extern 引入到其他文件中。



虚函数：

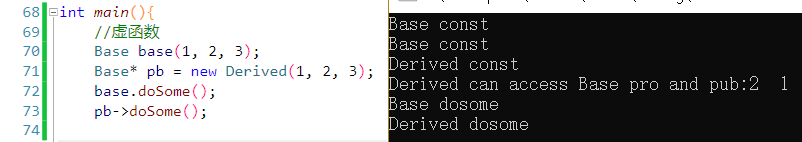
1.虚函数是用于实现运行时多态的机制。核心是可以通过基类对象指向派生类对象并使用派生类对象的函数。关键字为virtual。

2.纯虚函数没有实现，c++中纯虚函数用 = 0 来标识。纯虚函数的类是抽象类，只可以被继承，不能声明对象，常用来作为定义接口。

3.虚析构函数在被继承的基类中一定要声明，防止多态删除对象不完全的现象。

4.构造函数不能是虚函数。

5.构造函数和析构函数中的虚函数：类的虚函数在构造和析构中看做普通函数。不能在构造和析构中让自己“多态”。

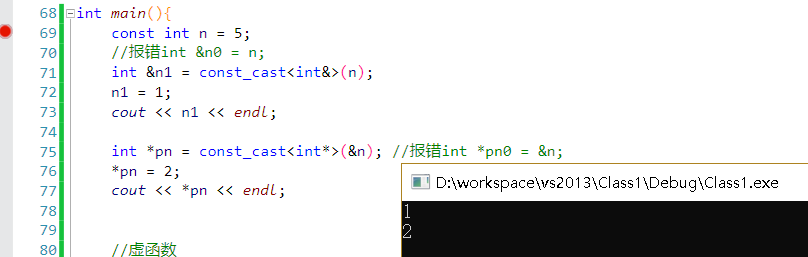


类型转换：

强制类型转换是有风险的。例如：整型-》指针 基类指针-》派生类指针 一种函数指针-》另一种函数指针 常量指针-》非常量指针。

C++引入了四种类型转换运算符：

Const\_cast 去除cosnt和volatile属性，唯一可以的运算符。不能转换const对象到非const对象。但是可以转换到1**引用**和2**指针**3**this指针**的cosnt 4 **volatile**。



Static\_cast 编译期用来代替隐式的类型转换进行明确。

可用于低风险的转换。

1. 整型和浮点型
2. 字符和整型
3. 转换运算符 重载类型转换运算符的显示转换（可声明为explicit不可隐式转换）。
4. 空指针void\* 转换为任何目标类型的指针

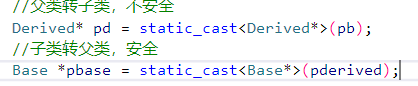
不可以转换：

1. 不同类型的指针。
2. 不同引用
3. 整型和指针。

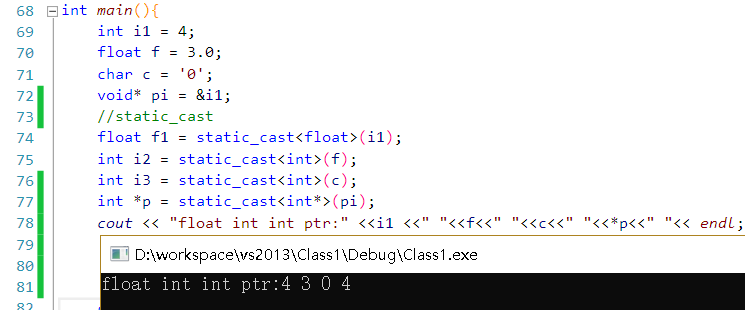
**转换父类和子类的指针**：

1. 父类转子类（不安全，没有运行时检查）；

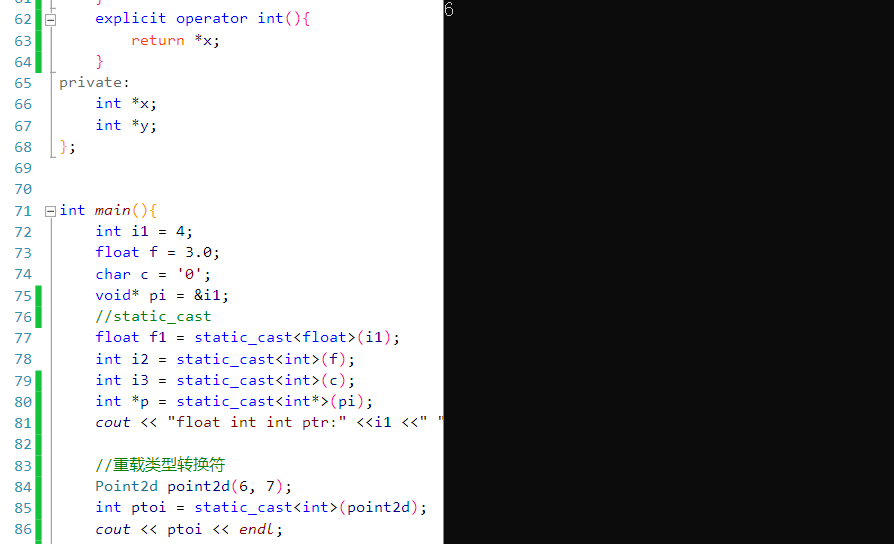
（2）子类转父类（安全）



低风险转换：



重载类型转换符：

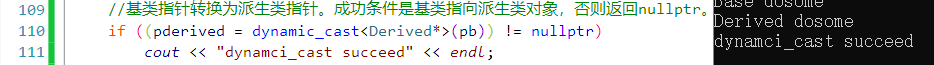


Dynamic\_cast 用于拥有虚函数的基类和派生类之间的指针和引用的转换。

1. 基类有虚函数。Dynamic\_cast 是进行运行类型检查，需要RTTI信息，而RTTI信息存储在虚函数表中，定义虚函数才存在。
2. 运行时检查，转换不成功返回空指针。
3. 非必要不使用，性能开销。

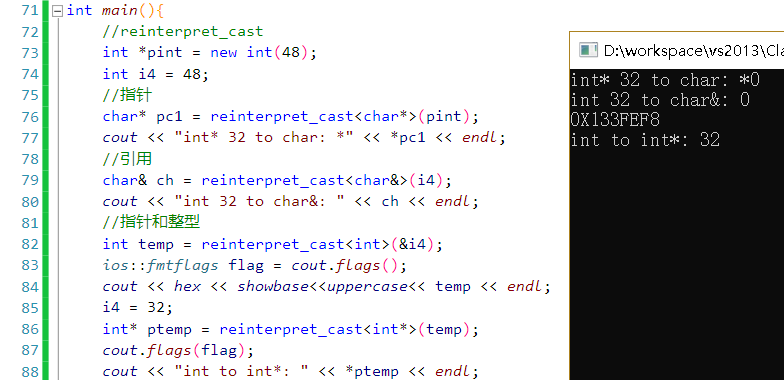
常见的转换方式：

1. 基类指针、引用转换为派生类指针。
2. 派生类指针、引用转换为基类指针（可以用dynamic\_cast,但是建议用static\_cast）;



Reinterpret\_cast 用于不同类型之间的转换，编译期强制诸字节复制。

1. 不同类型指针
2. 不同类型引用
3. 指针和能够容纳指针的整数类型
4. 父类子类指针（不检查，仅复制值）



C++：引用与指针，多维数组，命名空间，异常

1. 引用和指针。
2. 变量的指针就是变量的内存地址，变量内容为其他变量地址的变量是指针变量，简称指针。
3. 指针和数组。引用数组：1.下标法 a[i] 2.指针法 \*(a+3);
4. 指针作为函数参数。编译器将形参数组名看做指针处理。
5. 字符指针指向字符串。
6. 函数指针。

定义函数指针的语法： int （\*p）(int); 函数返回值 （\*变量名）（形参列表）；

可以用typedef定义指针类型：typedef int (\*fun\_type)(int);

1. 指针数组和数组指针。

Int\* p[4];//指针数组 int (\*p)[4];//数组指针

1. 多维指针。

Char\*\* p = char\* name[];字符串数组

Int \*\*arr =int \* arr[];二维数组

1. 指针和const

Const int \*p;//常量指针 int int const \* p;//常指针

1. 运算。P+-i 指的是P的指向移动i个元素。

指针向减，相差的元素个数。

指针比较。

1. 引用
2. 定义：变量的别名。
3. 声明必须初始化。
4. 不能更改指向。
5. No引用数组。
6. No引用的引用。
7. 多维数组

定义：type name[size n][size n-1]..[size1];

初始化：

（1）按行int [2][2] = {{1,2}，{3，4}}

（2）全部 int [2][2] ={1,2,3,4};全部赋值一维可以不指定 int[2][];

（3）部分 int [2][2] ={{1},{3});其他为零。

（4）默认为初始值 {}；不带{}是随机值。

1. 命名空间

命名空间是c++引入的用户命名作用域，解决程序中同名冲突。存放全局的符号。

1. namespace TV = television；别名
2. Using std::endl; 命名空间成员
3. Using namespace 空间名 使用所有名称。
4. 无名namespace 防止命名被其他文件引入。
5. Std标准库命名空间。
6. 异常

异常处理指的是对程序运行时出现的差错和例外情况处理。

1.c++异常处理由3部分组成：try throw catch;

Try 检查

Cathch 捕获异常

Tyr{

检查语句；

}

//可以有多个，catch只检查类型，不检查值。

Catch(...)捕获任何类型的异常。

Catch(异常类型 [变量]){

异常处理语句；

Throw；只写这一句，不处理，继续向上层抛出。

//throw如果没有找到catch，最终系统会使程序终止运行。

}

Throw 抛出异常；if（条件）throw 表达式；

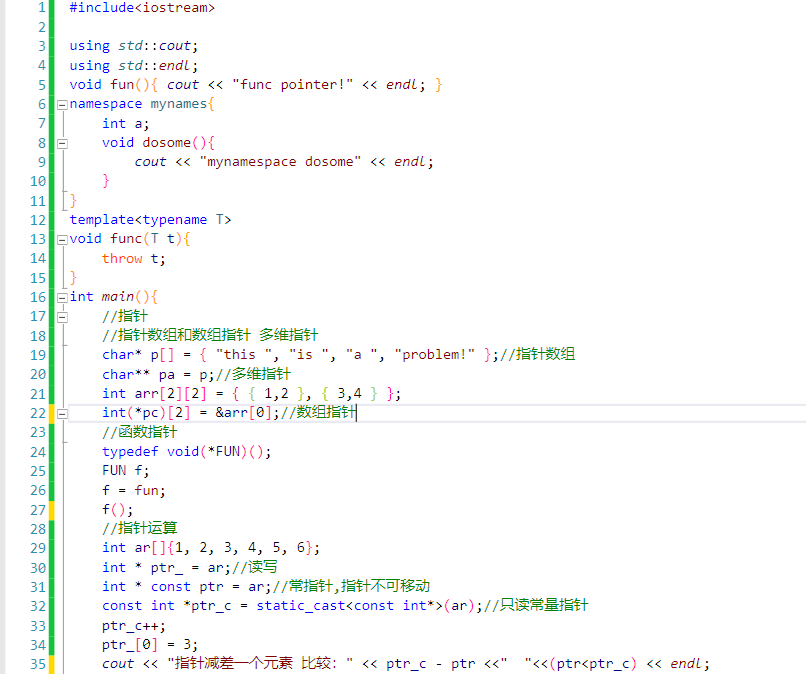
2.函数声明中指定异常处理情况。

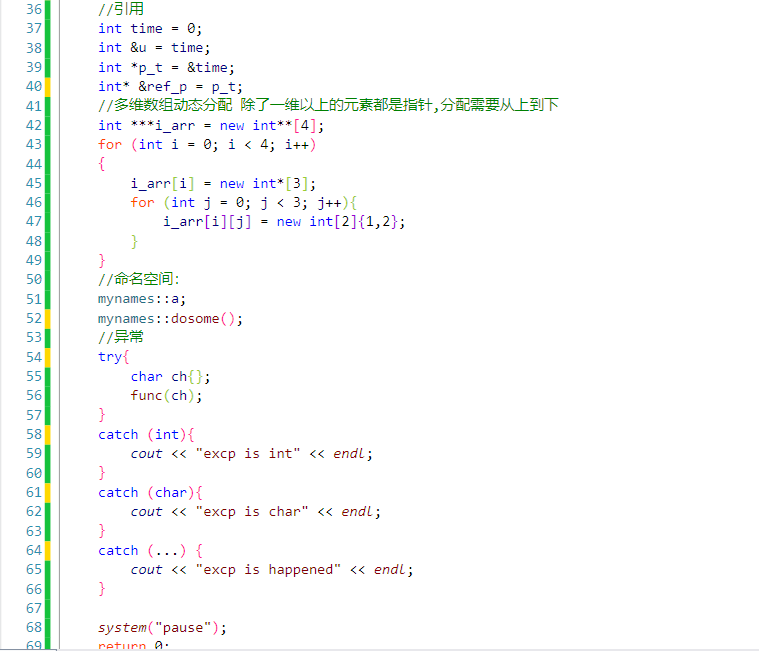
Int() throw(double);只抛出double异常

Int()throw(int,double,char)抛出三种

Int()throw()不抛出异常，如果有导致异常终止。

3.异常处理中处理析构函数。局部对象在catch被捕获后会被析构。





|  |  |
| --- | --- |
| 7月19日 | C++：函数模板与偏特化，类模板，类成员函数模板 |

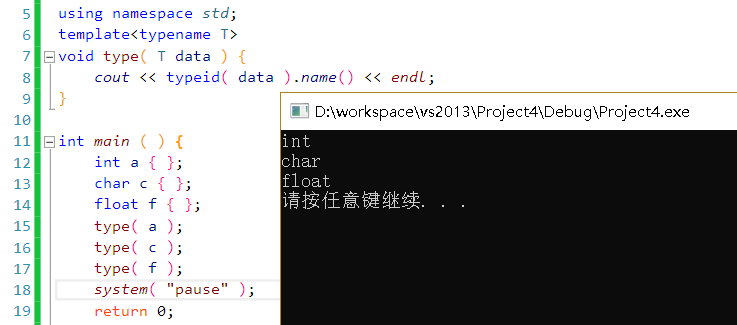
函数模板、类模板

1. 函数模板：

函数模板就是类型和形参不具体指定，用虚拟的类型和变量代替，建立通用的函数称为函数模板。

声明定义：template<typename 类型名,[...]> 普通函数定义体;

试用于函数体相同而类型不同的函数。



1. 类模板

类的结构和功能相同，仅仅是数据类型不同的通用类的定义。

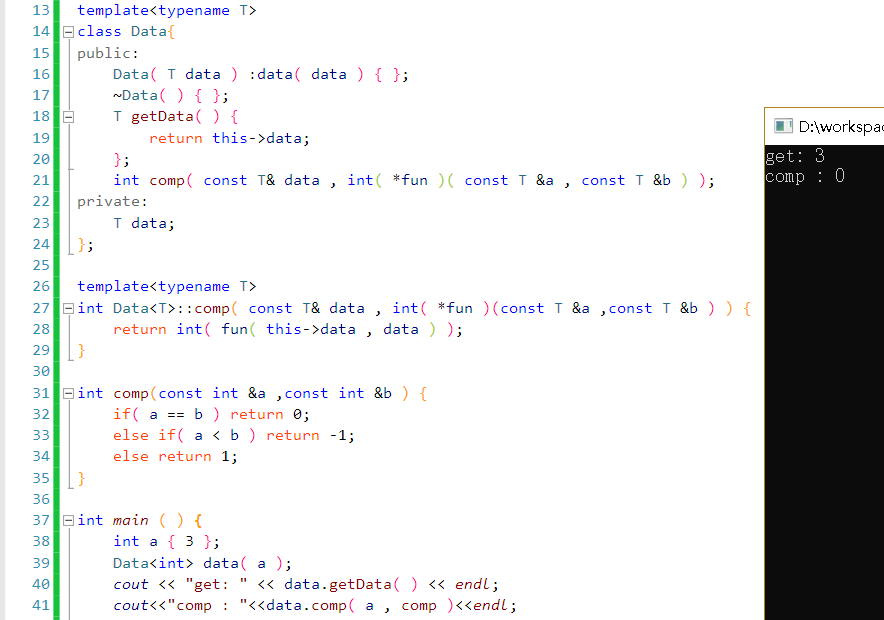
声明定义：template<typename T,[...]> 普通类定义；

对象定义：类模板名<类型列表> 对象名(参数列表);

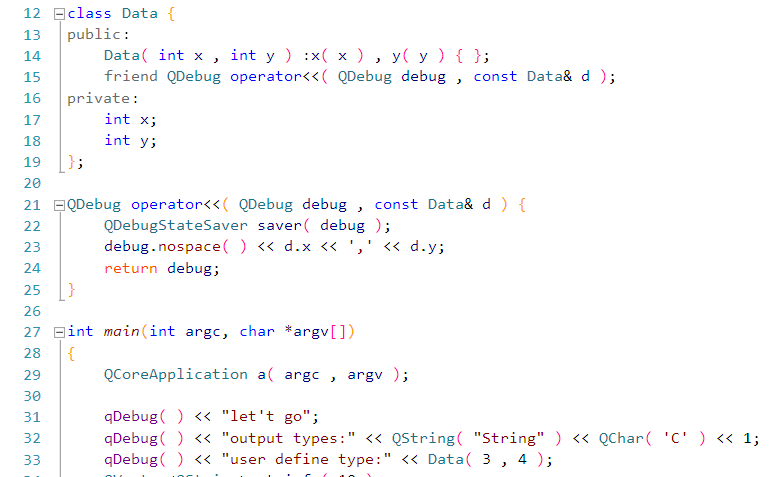
成员函数如果在类外定义：应写成模板形式。

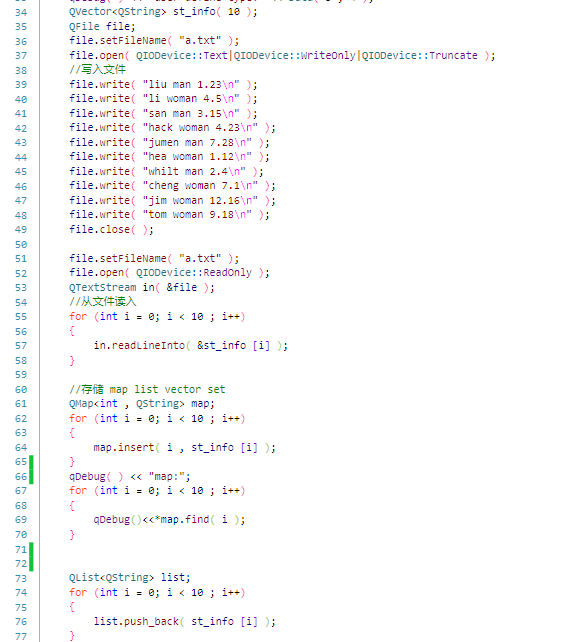
Template<typename 虚拟类型参数>

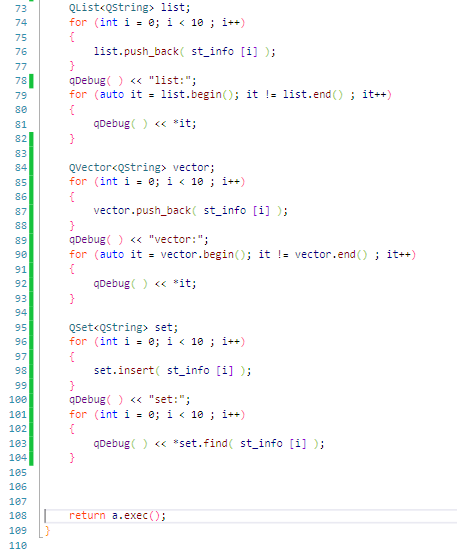
函数类型 类模板名<虚拟类型参数>::成员函数名(函数形参表）{。。。}；



|  |  |  |
| --- | --- | --- |
| 7月20日 | Qt：熟悉开发环境、QDebug、QString、QMap、QList、QVector、QSet | 输入10个学生的姓名、性别、生日信息，每个学生的信息封装到一个QString中，分别使用QMap、QList、QVector、QSet存储，然后遍历输出 |
| Qt：QtDesigner | 分别使用QtDesigner和不使用QtDesigner搭建计算器界面，实现简单的按键功能 |
| Qt：画布、鼠标和键盘事件 | 使用鼠标和画布实现画板功能，通过键盘按键可缩放、平移、旋转画板 |



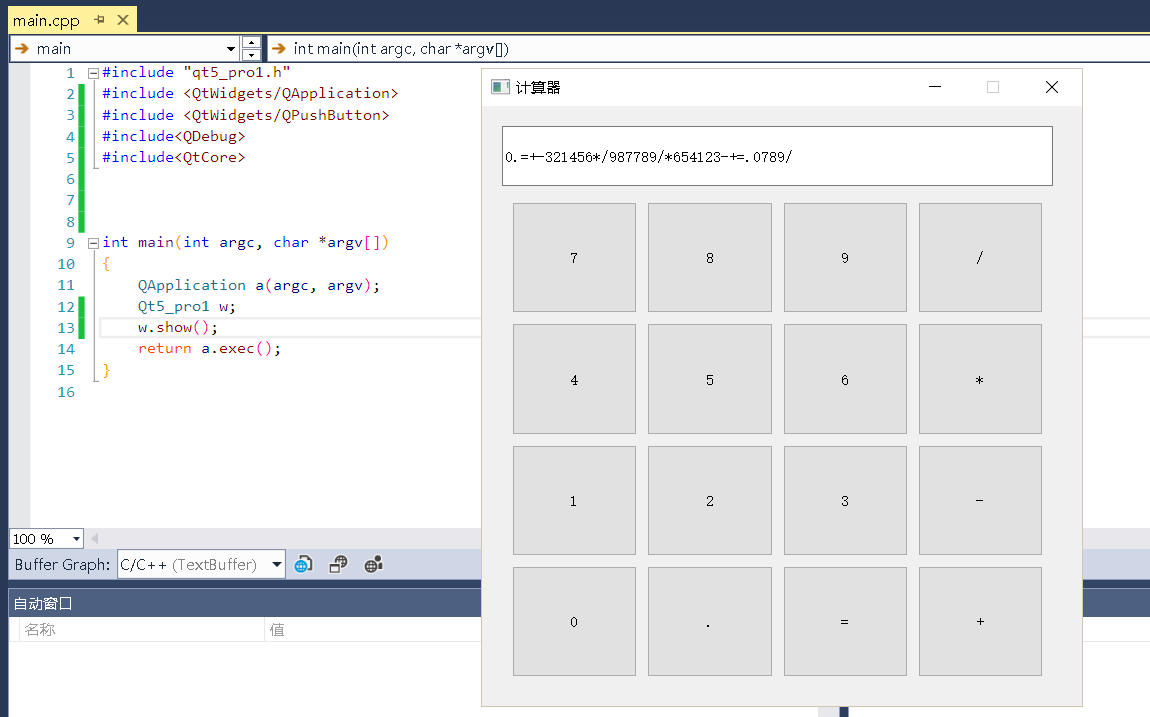




Qt designer计算器：

Designer提供的组件：

根据ui文件编译器会将设计文件转化成一个类，包含声明的头文件和实现的源文件。在类的private成员中定义了这个ui类的对象，通过这个ui对象设置父对象使ui对象显示出来，通过ui对象可以访问设计器中的组件，给按钮设置槽函数。Connect(sender,SIGNAL(),this,SLOT());



手工计算器：

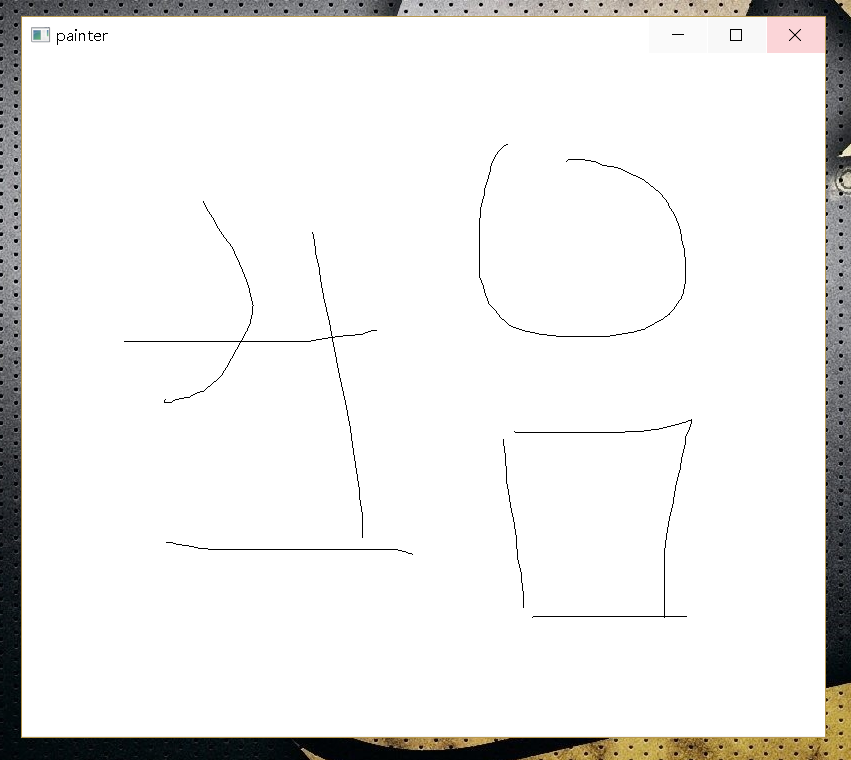
不同之处在于不需要在构造函数中添加父类的指针不需要设置ui，对象定义时指定parent把对象加入父类的对象树中，通过add\*方法加入的会自动加入对象树。



画布：

Qt事件用来处理物理的一些事件，例如鼠标、键盘、绘画、定时等。使用方式是通过重载protected的函数来实现，包括事件函数和事件过滤函数两种方式。

画布:通过事件获取鼠标和键盘按钮，实现鼠标拖动平移，按键缩放和旋转。



|  |  |  |
| --- | --- | --- |
| 7月21日 | Qt：QMainWindow | 使用QMainWindow搭建一个记事本编辑工具，拥有标题栏、菜单栏、工具栏、文本编辑窗口、状态栏 |
| Qt：树、表格、列表组件 | 分别使用QTreeWidget、QTableWidget、QListWidget实现学生信息管理系统 |

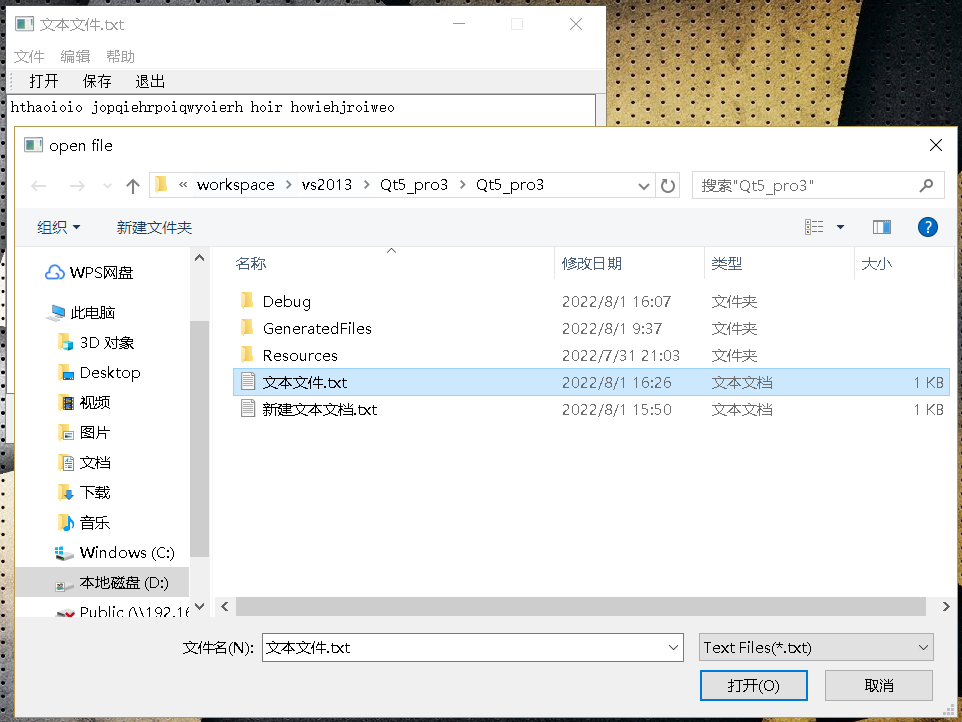
//记事本：

QMainWindow继承QWidget,用于创建通用应用程序窗口。结构包含：菜单栏（menu bar）、工具栏（tool bar）、停靠窗口（dock widgets）、状态栏（status bar）和中央窗口（central widget）

工作方式：菜单QMenu对象添加到menuBar()-addMenu，每个菜单添加动作QAction对象,再给动作对象绑定槽函数。

文件对话框：QFileDialog::getOpenFilaName 和QFileDialog::getSaveFilaName可以打开文件打开和保存（另存）对话框。

文本中的编辑操作：通过QTextEdit对象提供的槽函数。





Qt：树、表格、列表组件

//树：

QTreeWdiget（树结构组件），每个节点都是一个QTreeWidgetItem对象。

AddTopLevelItem()用于添加顶层节点，也可以在对象创建时设置parent为tree widget即可。

CurtentItem返回当前设置或选中节点。

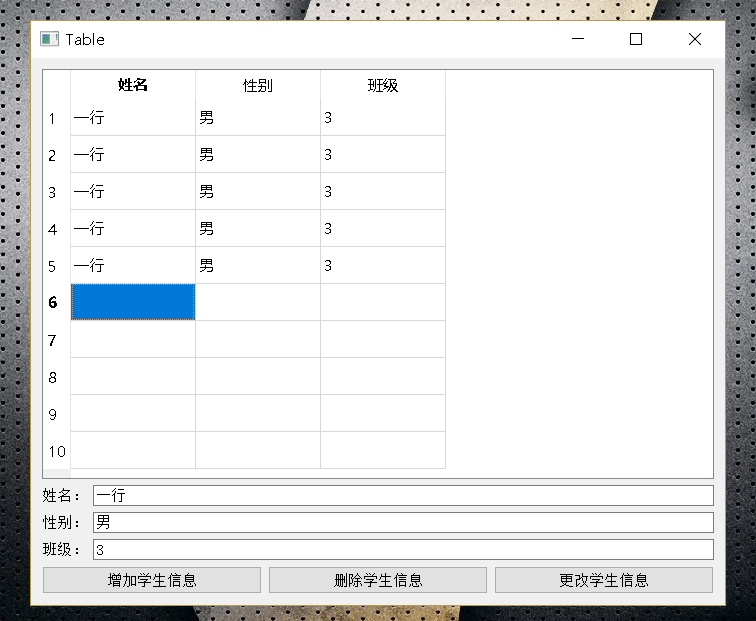
TakeTopLevelItem删除顶层节点，需要获取int index，indexofTopiiem(treewdigetitem\*)可以查找，top(LevelItem(int) 返回treeWidgetItem\*。

QtreeWidgetitem :

Addchild()添加子节点，parent()返回父节点指针，顶层为空nullptr(可以依次判断是否是顶层节点) chlid(int) 返回子节点指针。



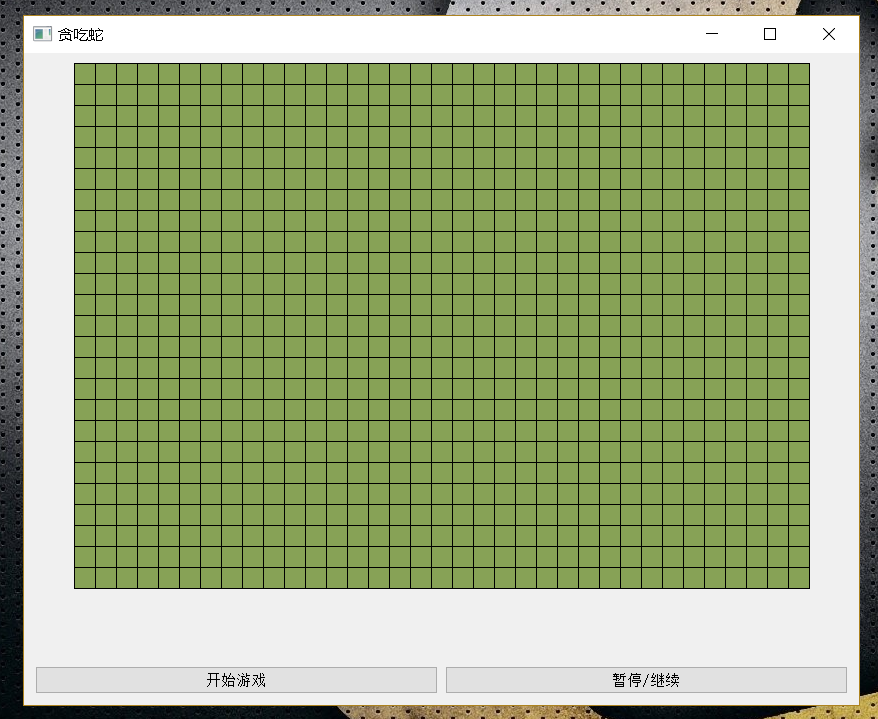
表格：



列表：



|  |  |  |
| --- | --- | --- |
| 7月22日 | Qt：综合测试 | 使用QMainWindow搭建贪吃蛇游戏，拥有标题栏、菜单栏、工具栏、贪吃蛇画面、状态栏，能够正常运行游戏 |

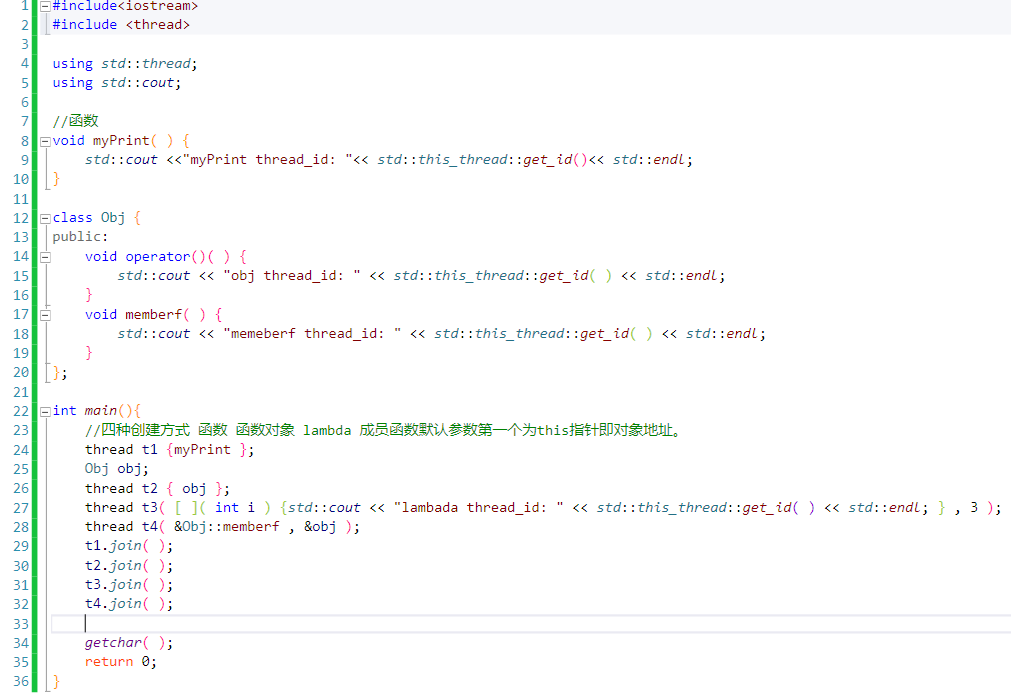


|  |  |  |
| --- | --- | --- |
| 7月25日 | 多线程：创建线程，创建、等待、发送信号量，创建、加锁、解锁互斥锁，共享内存 | 输出各知识点演练代码 |

线程创建：可调用对象：函数、函数指针、lambda表达式、bind对象、 函数对象

首个参数为函数地址或对象，其余为函数参数可变参数。

成员函数需要使用域名，首个参数为this指针。



条件变量：std::condition\_variable 对象的wait被调用后，当前线程被阻塞，直到其他线程调用notify\_后参与锁的竞争，获得锁后，对锁进行锁定，执行当前线程。

Wait 的锁对象通常为unique\_lock<mutex>

Wait(mtx) 的前提是当前线程已经获得（unique\_lock过）mtx，当wait被阻塞时，自动调用mtx.unlock()放弃锁。当被notify\_后自动调用mtx.lock加锁。

Wait(mtx,predicate)predicate返回值为bool，当pred返回false时才会阻塞，当被notify\_后只有pred为true才会被解除阻塞；等同于while(!pred())wait(mtx);

Wait\_for 和wait\_untill用于主动阻塞一段时间。

Notify\_one唤醒一个不确定的条件

Notify\_all唤醒所有等待条件



互斥锁：

四种：

Mutex 读写锁

Recursive\_mutex 递归锁

Time\_mutex 定时锁

Recursive\_timed\_mutex 定时递归锁

Lock：lock\_guard unique\_lock

其他：once\_flag adopt\_lock\_t defer\_lock\_t try\_lock\_t

函数：

try\_lock 尝试多个锁上锁

Lock 对多个锁上锁

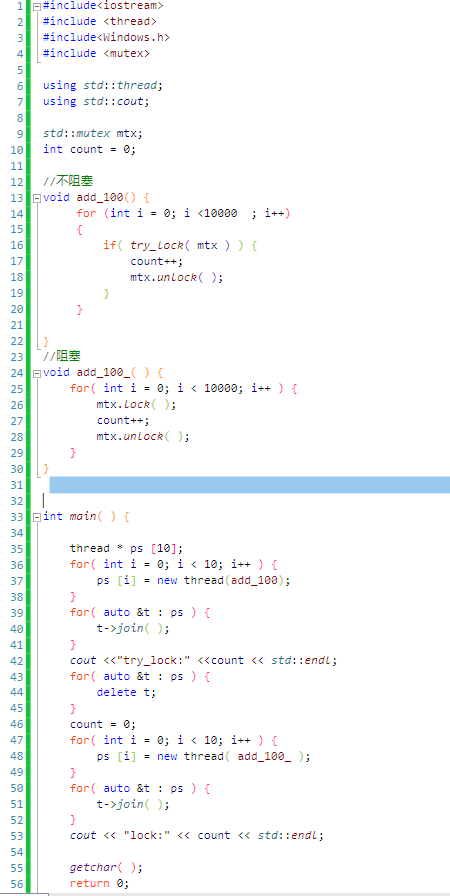
Call\_once 如果多个线程需要同时调用某个函数，call\_once保证只调用一次。

Metux默认是unlock的。

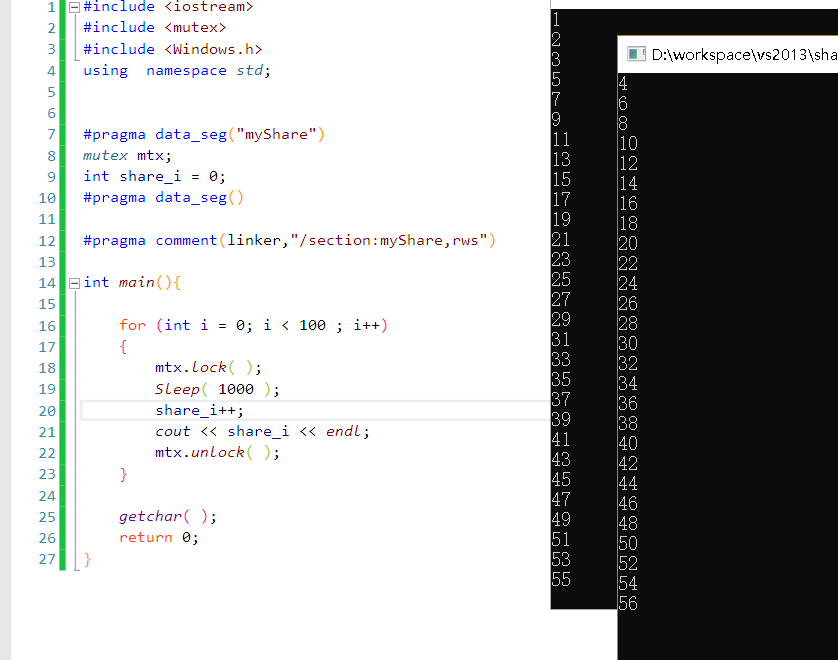
Lock 上锁，1.成功上锁2失败阻塞3.锁被调用线程上锁会死锁（deadlocak）.

Unlock 解锁

Try\_lock尝试上锁，失败不会阻塞，返回false。3.同上



共享内存：多进程通讯方式。



|  |  |  |
| --- | --- | --- |
| 7月26日 | 文件系统：格式化文件读写，块文件读写，文件夹操作与遍历，动态加卸载动态库 | 输出各知识点演练代码 |

|  |  |  |
| --- | --- | --- |
| 7月27日 | UDP编程：动态获取本机地址，创建socket，绑定地址和端口，配置地址复用和缓冲区及其它参数，加入组播，发送和接收数据 | 输出各知识点演练代码 |

1. WSAStartup函数：

Int WSAStartup( WORD wVersionRequested, LPWSADATA lpWSAData);

使用win socket前需要调用，第一个参数指明socket版本，高位副版本，低位主版本；第二个参数返回版本信息。成功返回0；

1. WSACleanup函数：

Int WSACleanup(void);

释放socket库的系统资源。

1. 检索 域名 通讯服务 协议的信息的函数有：

Gethostbyaddr gethostbyname gethostname getprotobyname getprotobynumber getserverbyname getserverbyport

1. gethostname()

Int PASCAL FAR gethostname (char FAR \*name , int namelen);

Name:缓冲区指针；namelen:缓冲区大小；

返回值：成功返回0；否则返回错误代码；

1. gethostbyname();

Struct hostent FAR\* PASCAL FAR gethostbyname(const char FAR \*name);

Name:调用gethostname()得到的主机名。

返回值：返回指向hostent结构指针。

1. MAKEWORD(BYTE blow, BYTE bhigh);

Blow：主版本; bhigh: 副版本

1. htons ntohs

#include winsocket2.h

u\_short htons(u\_short hostshort);

组播：

组播地址：

224.0.0.0-239.255.255.255所有

224.0.0.0-224.0.0.255特殊用处预留，不能被路由。

224.0.0.1同一网段所有主机

224.0.0.2同一网段所有组播路由器

224.0.1.0-224.0.1.255公用组播地址，可用于Internet

224.0.2.0-238.255.255.255公用组播地址（临时），全网有效

239.0.0.0-239.255.255.255本地组播地址

Setsockopt函数：

SOL\_SOCKET层

1. SO\_RCVTIMEO SO\_SNDTIMEO收发超时
2. SO\_RCVBUF SO\_SNDBUF收发缓冲区，udp没有发送缓冲
3. SO\_BROADCAST广播

|  |  |  |
| --- | --- | --- |
| 7月28日 | DDS通信：创建域参与者，创建主题，创建数据读者、写者，收发主题数据 | 输出各知识点演练代码 |

ACI\_DDS流程:

1. DDS\_ParticipantFactory\_init()

日志和任务列表初始化

1. DDS\_DomainParticipantFactory\_create\_participant

创建域参与者

1. DD\_DomainParticipant\_create\_topic

创建主题数据

1. 发送 DDS\_Topic\_create\_datawriter

DDS\_StringDataWriter\_write

1. 接收 DDS\_Topic\_create\_datareader

ReceiveData(阅读器回调函数)

(1 DDS\_StringDataReader\_take 将reader中的数据读入到DataSeq中。

(2 DDS\_StringSeq\_get 获取DataSeq队列中的数据。

(3DDS\_StringDataReader\_return\_loan资源释放

\*\*其中1.域参与者、主题、写入器、阅读器属于实体需要设置Qos策略。

Qos为NULL设置默认Qos，一般写入器和阅读器需要设置可靠性为BEST\_EFFORT\_RELIABILITY\_QOS 或 RELIABLE\_RELIABILITY\_QOS，可靠传输和不可靠传输。

|  |  |  |
| --- | --- | --- |
| 7月29日 | 综合测试 | 创建8个进程，随机收发主题数据，另外创建1个进程，监控前8个进程收发数据日志，并记录到文件中 |

|  |  |  |
| --- | --- | --- |
| 8月1日 | 算法分析：时间复杂度与空间复杂度，贪心、分治、动态规划算法思想 | 输出常见排序算法时间复杂度与空间复杂度，贪心、分治、动态规划算法思想各实现一个例子 |

算法时间复杂度：算法的时间复杂度是指执行算法需要的计算工作量。一个算法中语句执行的次数称为语句频度或时间频度，记为T(n). n称为问题的规模，当n不断变化的时，时间频度也不断变化。一般情况，算法的基本操作执行次数是问题规模n的一个函数，用T（n表示，若有f(n),存在一个正常数使得c\*f(n) >= T(n) 恒成立，记T(n) = O(f(n)),称O（f((n)）为算法的渐近时间复杂度，简称时间复杂度。

|  |  |
| --- | --- |
| 复杂度 | 术语 |
| O( 1 ) | 常数复杂度 |
| O( log n ) | 对数复杂度 |
| O( n ) | 线性复杂度 |
| O(n log n) | N log N复杂度 |
| O( n ^ a ) | 多项式复杂度 |
| O(a ^ n) a > 1 | 指数复杂度 |
| O( n! ) | 阶乘复杂度 |

空间复杂度：与时间复杂度（Space Complexity）类似，指算法执行时所需的存储空间的度量，记作：S(n) = O(f(n));

包括部分：程序内存、输入数据内存、执行过程内存；

稳定性：简单来说就是所有相等的元素经过排序后，仍然保持它们在排序前的相对次序我们就说这种排序是稳定的。反之就是不稳定的。

常见排序算法复杂度： 基于比较的排序算法（下限n log n的限制）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 算法 | 平均时间复杂度 | 最坏时间复杂度 | 最好时间复杂度 | 空间复杂度 | 稳定性 |
| 插入 | O(N^2) | O(N^2) | O(N) | O(1) | 稳定 |
| 希尔 | O(N^1.5) | O(N^2) | O(N) | O(1) | 不稳定 |
| 选择 | O(N^2) | O(N^2) | O(N^2) | O(1) | 不稳定 |
| 堆 | O(N log N) | O(N log N) | O(N log N) | O(1) | 不稳定 |
| 冒泡 | O(N ^ 2) | O(N ^ 2) | O( N ) | O(1) | 稳定 |
| 快速 | O(N log N) | O(N ^ 2) | O(N log N) | O(N log N) | 不稳定 |
| 归并 | O(N log N) | O(N log N) | O(N log N) | O(N) | 稳定 |

非比较排序算法：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 计数 | O(N + K) | O(N + K) | O(N + K) | O(N + K) | 稳定 |
| 桶 | O(N + K) | O(N ^ 2) | O(N ) | O(N + K) | 不稳定 |
| 基数 | O(N \* K) | O(N \* K) | O(N \* K) | O(N + K) | 稳定 |

|  |  |  |
| --- | --- | --- |
| 8月2日 | 数据结构：列表，队列，栈 | 手写代码实现列表、队列、栈 |

|  |  |  |
| --- | --- | --- |
| 8月3日 | 基础算法：二分查找算法，冒泡排序，选择排序，插入排序，二分插入排序，希尔排序 | 输出各算法实现代码，分析代码时间复杂度和空间复杂度 |

二分查找算法

时间复杂度：每一次迭代查找规模减半，查找次数为log2 N，所以复杂度为log N.

空间复杂度：O(1)

冒泡排序

时间复杂度：需要执行n-1次，执行次数为（n-1）+(n-2)+...+1 = n\*(n-1)/2；为平方 复杂度O(n^2)。最好情况需要优化，即每一次遍历时判断是否发生了swap，如果第一 次遍历就没有swap说明序列是有序的，直接return。执行了n-1次，复杂度为O(n)。

空间复杂度: O(1)

选择排序

时间复杂度：执行n-1次遍历，执行次数为（n+(n-1)+...+2 = (n+2)(n-1)/2;易得为平方复 杂度O(n^2)。

空间复杂度:O(1)

插入排序

时间复杂度：执行n-1次遍历，执行次数为1+2+...+(n-1) = n\*(n-1)/2,为平方复杂度O(n^2)。最好时间复杂度添加判断每次是否都大于前一个已排序序列的最后一个元 素，即当前待插入元素的前一个元素，当待排序序列有序时，执行n-1次，复杂度O(n)。

空间复杂度:O(1)

二分插入排序

时间复杂度：执行n-1次遍历，执行次数为依然为1+2+...+(n-1) = n\*(n-1)/2,为平方复杂度O(n^2)，因为二分查找只是加快了查找的速度，但是swap的数量依然不变。

空间复杂度:O(1)

希尔排序(插入排序和二分排序的改进）

时间复杂度：希尔排序的时间复杂度和数据长度的递减序列相关，复杂度在O(1.3-2)之 间。

空间复杂度:O(1)

|  |  |  |
| --- | --- | --- |
| 8月4日 | 基础算法：快速排序，二路归并排序，堆排序，桶排序，计数排序 | 输出快速排序实现代码，分析代码时间复杂度和空间复杂度，了解其它排序算法思想 |

快速排序算法：

快速排序算法的思想是分治法，首先将序列划分成三个部分，中间为一个元素，左边元 素均小于等于中间元素，右边元素均大于等于中间元素。递归左边的部分和右边的部分。

时间复杂度：假设每次都交换所有的元素，即每次递归首元素都是最小元素，递归深度 为n-2,执行次数为：(n-1)+(n-2)+...+1 = n\*(n-2) /2 ，为平方复杂度O(n^2)。

最好的时间复杂度为：每次都平均分割，递归深度为log N, 每一层的递归的次数为2^k 个，k为层数，即1,2,4,8,16这样的规律，每层的执行次数和都相同为n。

故执行次数 = N log N 。

空间复杂度：每一层都需要将除去中间元素的其余元素拷贝到临时空间中，即每层需要 的内存为n-1、n-2 、n-4、n-8...... ，忽略常数，执行次数为N log N。故空间复杂度为 N log N。

二路归并排序算法：

思想，归并两个已排序序列叫归并，如果一个序列的两部分已有序便可以归并成有序的 一个序列。采用递归思想，如果要归并序列左右部分传递给下一层递归调用，当序列长 度为一式返回上一层进行归并。

堆排序：堆排序的思想是维护一个最小或者最大堆，依次从堆顶取出元素，维护堆，再次取 出堆顶元素，依此类推，直到堆为空。

计数排序：是一个非基于比较的排序算法。他的优势是当对于范围K的整数排序时，时间复 杂度为O(n+k),一定范围内快于任何比较排序算法，当O(K) > O(n\* log n)时不如比较排序 算法。算法的思路是对于给定序列中的每一个元素遍历，将它映射到新数组的下标中， 新数组下标对应值++。最后对应下标值的个数即为值的个数。

桶排序：用到了递归和分治的思想。确定待排序序列的范围，定义适合的桶子数量，遍历每 个序列，将序列/除以桶子的长度加上起始位置放到一个桶子里，对桶子里的数递归进 行，桶子里元素为一个或零个返回。以一个全局计数器index= 0标志以排序个数，递归 返回时为1将元素赋值给序列的 arr[index]中，index++；

基数排序：可以用到计数排序作为底层的算法。上层的思路是从数据的地位权开始遍历数据 排序，直到最高位，排序完成。

|  |  |  |
| --- | --- | --- |
| 8月5日 | 版本管理：学习使用GIT进行项目版本管理 | 输出GIT学习笔记 |

**版本控制**：版本控制是一种记录文件内容变化，以便将来查阅特定版本修订情况的系统。

1. 本地版本控制： 如rcs，它通过在本地计算机上保存不同版本的补丁记录修订前后的变化。
2. 集中化的版本控制： 如CVS、subversion、perforce等。由一个集中管理的服务器，保存所有的文件修订版本，协同者连接服务器，取出最新的文件或者是提交更新。
3. 分布式版本控制： 如git、mercurial、bazaar、darcs等。客户端不仅仅只是提取最新的文件快照，而是完整的把代码仓库镜像拷贝下来，在本地提交更新，最后合并提交到中心版本上，适合多人协同开发的模式。

**启动git**：

1. Git Bash: unix风格命令行，推荐使用。
2. Git CMD: windows风格命令行。
3. Git GUI: 图形界面的Git。

**常用Linux命令**：

1. cd change directory
2. cd .. change to father directory
3. pwd print workdirectory
4. Ls(ll) list files ll是ls -l别名-l参数列出详细信息
5. Touch 新建文件
6. Rm 删除文件
7. Mkdir 新建目录
8. Rm -r 删除目录
9. Mv 移动文件
10. Reset 重新初始化终端/清屏
11. Clear 清屏
12. Help 帮助
13. Exit 退出
14. # 注释

**Git 配置**：

1. 查看配置： git config -l

查看系统配置： git config --system --list

查看当前用户[global]配置：git config --global --list

相关的配置文件：

1. git\mingw64\etc\gitconfig文件 :安装目录下的gitconfig --system系统级
2. C:\users\administrator\.gitconfig 只试用于当前登录用户的配置 --global 全局

**设置用户名和邮箱**（用户标识，必要）

Git config --global user.name “name” #名称

Git config --global user.email [234@qq.com](mailto:234@qq.com) #邮箱

**Git基本理论:**

Git在本地有三个区域：工作目录（working Directory）、暂存区（Stage/Index）、版本库（Repository/Git Directory）.如果加上远程的仓库（remote Directory）就可以分成四个工作区域。

Workspace: 工作区，本地放代码的目录。

Index/Stage: 暂存区，保存临时改动。

Repository: 本地仓库，版本库，包含所有提交的版本数据，其中HEAD指向最新仓库版本。

Remote: 远程仓库，托管代码的Git服务器。

Git add/git checkout 工作区和暂存区提交和撤销

Git commit/git reset 暂存区和版本库提交和撤销

Git push/git pull 本地仓库和远程仓库的推送和拉取

**一般流程**：

1. 在工作目录中添加、修改文件。
2. 将需要进行版本管理的文件放入暂存区。
3. 将暂存区中的文件提交到本地git仓库。

因此，**本地文件有四种状态：Modified已修改 Staged已暂存 Committed已提交 Untracked未跟踪**

**创建本地仓库：**

1. 新建：git init 默认以当前目录为工作目录，在目录下生成.git目录
2. 克隆远程仓库：git clone [url]

**文件操作**：

Untracked文件：不参与版本控制，git add 变为Staged.

Unmodified文件：文件和版本库最新一致，git rm移除版本库，成为untracked文件，修改：modified

modified文件：git add 变成staged, git checkout 丢弃修改，返回unmodified.

staged文件：git commit 变成unmodified,git reset HEAD filename,变成Modified

Git status [filename] 查看具体或所有文件的状态。

**忽略文件**：

某些数据库文件、临时文件、项目文件、设计文件我们不想纳入版本控制中，可以通过在主目录下添加.gitignore文件。

规则：

1. 空行 #开始的行被忽略
2. 通配符 （\*）任意多个字符 （?）一个字符 [abc]字符范围 （{string1,string2}）可选字符串
3. （！）名称前 不被忽略
4. （/）名称前 当前目录下忽略，子目录不忽略
5. （/）名称后 子目录而非文件

**Git分支操作**：

Git branch 列出分支

Git branch 列出远程分支

Git branch [branch name] 新建分支

Git checkout -b [branch] 新建分支并切换到该分支

Git merge [branch] 合并指定分支到当前分支

Git branch -d [branch name] 删除分支

Git push origin --delete [branch name] 删除远程分支

Git branch -dr [remote/branch] 删除远程分支

//更多命令和学习可以查看gitee主页下面的学习git链接了解。

|  |  |  |
| --- | --- | --- |
| 8月29日 | DDS1.4：学习至2.2.2.1章基础设施模块 | 能够熟练绘画DCPS概念模型图，能够简单绘画基础设施模块图，能够简单叙述基础设施模块内容，输出学习笔记 |

1.1引言

定义：DDS规范定义了以数据为中心的发布订阅模型（DCPS）通信模型，包含应用接口和通信语义（行为和服务质量）。

目的：实现在正确的时间将正确的信息传递到正确的目的，且保证高效和鲁棒性。

为了使用资源的过程的高性能、可预测性和高效性，接口设计遵循：

1. 中间件提前分配资源，减少动态分配的资源。
2. 避免使用无限或不可预测的资源实现的功能。
3. 减少数据拷贝。

最大程度的基于类型的接口（考虑实际的数据类型进行设计的接口）。

服务质量（Oquality of Service,,Qos）是用来指定服务行为的一般概念。Qos由一系列Qos策略组成。

发布端和订阅端明确划分，独立。

1.2宗旨

以数据为中心的发布订阅模型的底层是数据模型。该数据模型定义了“全局数据空间”并指定发布者和订阅者如何访问此数据空间。该数据模型可以像一组不相关的数据结构一样简单，每个数据结构都由一个主题和一个数据类型来标识。主题提供了一个唯一标志全局数据空间内某些数据项的标识符。数据类型提供结构信息，告知中间件如何操作这些数据。

2.1概要

DCPS模型定义了应用程序用于发布或订阅数据对象数值的功能。模型允许：

1. 发布程序识别其计划发布的数据对象，并为这些对象赋值。
2. 订阅程序识别其感兴趣的数据对象，并访问这些对象的值。
3. 应用程序定义主题，将数据类型信息添加到该主题上，允许应用程序创建发布者和订阅者实体，并将Qos策略添加到这些实体上，最终可以操作这些实体。

2.2平台无关模型（PIM）

2.2.1

类的定义用UML和表格记录格式。PIM级别定义了返回码。

概述：

信息流借助以下实体：

发送方：发布者（publisher）、数据写入者（DataWriter）

接收方：订阅者（Subscriber）、数据读取者（DataReader）

主题：（Topic）订阅必须明确的引用发布，发布必须以这种方式为订阅所知。

主题的目的：关联名称（数据域中唯一），数据类型和与数据本身相关的Qos。

除主题Qos外，主题关联的数据写入者的Qos和与数据写入者关联的发布者的Qos控制发布者这一方的行为。数据读取者和订阅者Qos控制订阅者这一方的行为。

整体概念模型：

所有主要通信对象都遵循以下统一模式：

1. 支持Qos
2. 接受监听器（Listener）；监听器作为中间件提供通用机制，通知相关异步事件。监听器与状态条件的变化有关。状态条件在2.2.4小节。
3. 接受状态条件（StatusCondition）（以及数据读取者的一组读取条件 对象）；条件为中间件和应用程序之间的设备通信方式提供支持（基于等待而不是基于通知）。

所有的DCPS实体都属于域参与者（DomainParticipant）。域参与者表示域中程序的本地成员身份。数据域是个分布式概念，使得域中程序能够相互通信。同一数据域的发布者和订阅者才可通信。

数据域实体（DomainEntity）是个中间对象，唯一目的是声明域参与者不能包含其他域参与成员。

2.2.2平台无关模型（PIM）的描述

五大模块：

1. 域模块2.发布模块3.订阅模块4.主题模块5.基础设施模块

基础设施模块由以下类组成：

Entity

DomainEntity

QoSPolicy

Listener

Status

WaitSet

Condition

GuardCondition

StatusCondition