计学组资料

作者

目录

1 GDB 部分 (如何减少与修复 bug)

在编程的过程中，最让人泄气的事情就是：你兴致冲冲地想到一个绝佳的点子，匆匆地在你的电脑上实现，然后紧张地按下运行，可终端那黑糊糊的屏幕上所显示的东西与你所预期的竟然完全不同！甚至，你已经在本地测试了好几百次，可当你再一次将答案提交上 OJ 时，看到的依然是冷冰冰的：Wrong Answer。

这样的经历曾多次（现在也时不时）打消我对学习的热情。希望我所写的这篇小短文能够帮助大家减少这样的糟糕体验，以及在发现结果与预期不符时，也不至于束手无策。

1.1 减少错误的好习惯

1.1.1 写伪代码

正如吹牛之前要打草稿一样，写代码之前也应该要先打草稿，也就是写伪代码。以前，我也不喜欢写伪代码，觉得这样浪费我的时间。但现在，我几乎在写每个程序之前都会写伪代码。一方面是因为写伪代码可以显著降低 bug 出现的频率；另一方面，写伪代码本身就是一个充满趣味的过程，比在电脑上写程序还有意思。因为在写伪代码的时候，可以更多地关注程序的本质，而暂时撇去一些细节，大家试着写一些伪代码就明白了。我一般分两个阶段写伪代码：

- 阶段一：用白话写出大概要干什么。语言不一定很详细，但脑袋中一定要有大概的图像，就像在放视频一样。下面举两个例子：

> 例子 1 冒泡排序
> 从头到尾两两比较数组中的元素，把较大的元素弄到后面去，第 i 次循环后，后 i 个元素一定是最大的 i 个元素，直到排序完成。
> 例子 2 链表的删除
> 第一步：找到我要删除的元素的前一个元素第二步：把下一个元素删除，然后连接下下个元素

- 阶段二：用类似 python 这样的语言写出伪代码。这一步中的语言并不重要，可以说怎么习惯怎么来。我个人喜欢算法导论一书中的伪代码写法，大家可以参考。以冒泡排序为例：

```
        bubbleSort(A, n):
        for (i = 0; i < n-1; i++):
            for (j = 0; j < n-i-1; j++):
                if (A[j] > A[j+1]):
                    swap(A[j], A[j+1])
```

这一步可以说是整个写代码过程中最为重要的一步。有两个要点：

- 合理地使用抽象：有的时候，有的部分很简单但又很麻烦，这时我们可以假装有一个趁手的函数。例如上面例子中的 swap（虽然 stl 中有这个函数）。这个技巧可以让人关注程序的主要逻辑。
- 关注循环/递归：这是代码中最容易出错的部分，详见下一小节。

伪代码的阶段数取决于你最终程序的复杂程度。如果我要完整实现一个链表，那么我会多写一层伪代码，用于写出链表的 ADI（抽象数据结构，即它的所有接口，比如插入、删除等等）。如果我要做一个大型项目，那么伪代码的阶段数会更多。

1.1.2 避免常见的错误

- 循环和递归是最常见的错误来源。这里的循环并非指的是打印 100 次”hello world” 这样的简单循环，而是像冒泡排序中的循环那样富有变化的循环 (通常是多重循环)。

如何证明一个循环或者递归会符合预期地运行？本质的方法是数学归纳法。在《算法导论》中以插入排序为例进行过完善的分析，推荐大家参考。我在这里也以冒泡排序为例：

在冒泡排序中，主要维持了两个性质：
- i 代表了数组末尾已排好的元素的个数
- 索引为 j 的元素始终是它及之前的元素中最大的。
如果这两个性质能一直保持，且停止在了正确的地方，我们的冒泡排序就是正确的。下面证明这两个性质一直被保持：
内层循环（目标：找出前 n-i 个元素中最大的元素）：
- 初始：j=0 时，索引为 j 的元素是它及之前的元素中最大的那个。
- 中间：索引为 j 的元素始终是它及之前的元素中最大的。
- 终止：循环结束时，j=n-i-1，索引为 j 的元素本身及之前共 n-i 个元素，且其为这些元素中最大的，故内层循环正确。（可带入 i=0 检查是否越界）
外层循环
- 初始：i==0 时，数组末尾共 0 个元素是排好序的。
- 中间：由于内层循环正确，所以第 i==k 时，末尾的 k 个元素是按从小到大的顺序排好序的。
- 终止：循环结束时，后 n-1 个元素是数组中最大的 n-1 个元素，且已排好序，故整个数组已排好序。

在这一步中，最应该关注细节，避免循环边界有问题。

• 边界条件往往也是容易出错的地方。例如，在链表问题中，对头、尾节点有时需要特殊讨论。其它的边界条件有：无法处理“空的情况”，数据过大溢出等。

• 我们也常常犯一些低级的错误：比如漏写语句（该 ++ 的地方没有 ++）、参数顺序出错、大于小于号写反、未考虑边界情况、想的是 i 写的是 j 等等。对于这些问题，应有一套应对的方法。比如，带入边界情况，试试看程序是否会出错。在参加程设考试的时候，我列了一个自己会犯的常用错误的 checklist，当出 bug 的时候一一排查，这帮助我节省了很多时间。

1.2 GDB 的基本使用

GDB 是 GNU 的调试器，可以用于调试包括 C 和 C++ 在内的语言。（GNU 是一个自由软件操作系统，与软件开源运动密切相关，gcc、make 等也是这一运动的产物）。在处理 Segmentation fault（段错误），内存泄露

（valgrind 工具在探测内存泄露时非常有用）等较为复杂的错误时，用 GDB 调试往往能较快找到问题。而且，通过 GDB 观察程序有助于程序思维的锻炼，在学习汇编和计算机系统时也时不时会用到 GDB。对于 GDB 的使用，网上教程比较丰富，大家可以先查找简明教程入个门 (基本的 r，s，n，b)，然后查阅 GDB 的官方文档获得更深入的认识。以下梳理我使用 GDB 时的常用流程及命令：

1.2.1 用 gdb 执行程序的技巧

- r 1 2 3 , 则 1 2 3 就作为输入被传进程序中了。
- 更进一步，可以利用重定向，将需要测试的数据写入一个 txt 文件，然后输入命令 r < a.txt , 这样文件中的内容就作为输入被传进程序中了

1.2.2 用 GDB 查看信息的技巧

- 当程序出现段错误时，我一般会先 r，然后当程序终止时输入命令 bt(backtrace) 打印当前环境中所有栈的信息，这适合于有多个函数或有递归的情况。当栈的信息出现后，info stack 会显示当前栈的信息，使用 frame n（n 是对应栈的编号）up down 指令可以移动到指定的栈中去。(参考C 语言中文网，GDB 官方文档) • p，x, info, display 是几个常用的打印信息的命令。x 后往往跟的是一个指针或者地址，执行后会打印出对应地址的变量信息。p 后面跟参数可以用比较友好的形式展现一些信息，例如 print (char *) 0xbfff890 将会显示对应地址中存储的 C 字符串，非常有用 (参考 CSAPP 中提供的gdb-note)。info 主要是打印断点信息、栈帧、寄存器、程序本身的，例如 info b 打印出所有断点的信息。display 的用法比较简单，不多赘述，跟 undisplay 搭配使用。

1.2.3 GDB 断点的技巧

用 GDB 打断点与取消断点比较简单，下面介绍几个高效利用断点的技巧：

- continue 语句：程序继续执行，直到遇到下一个断点。
- until n：程序继续执行，直到断点 n
- rc（reverse continue）语句：返回到上一次停下的位置，适用于执行过了头的情况。

• 条件断点：condition bnum expr 用于为已有断点设置条件。例如：condition 1 i == 5 意为：当程序执行到断点 1，且 i 的值为 5 时程序终止。condition bnum 去除条件断点的条件。

• 观察断点：watch var，当 var 的值改变时程序终止。

总的来说，GDB 的使用还是需要大家熟能生巧，在实践中总结最适合自己的调试方法。

2 输入输出 & STL（标准模板库）& 奇怪的技巧

该部分介绍的主要是一些程设课堂上不怎么讲，但是做题中可能会用到的知识点，相对而言不是那么的基础，略有拔高。这些知识点、技巧通常需要查阅大量资料并进行总结，在此我利用亿点点 编程经验总结出来。**（由于是复习资料，简洁明了、通俗易懂是重点，所以表达较为口语化）**。

提示：对于课内考试而言，该部分知识无需完全掌握，部分内容了解即可。但是对于算法竞赛、职业技能非常重要。

2.1 输入输出

主要介绍标准输入输出（在控制台界面的输入输出 cin, cout, getline, putchar, getchar, puts, gets, scanf, printf）和文件输入输出（文件重定向 freopen, 文件指针 FILE*, 文件流 fstream）。

2.1.1 cin

C++ 里封装好的标准 IO(Input & Output) 流。头文件 <iostream>, 命名空间 std, 基本使用方法略。

- cin 是一种较为智能的读入方式：自动判断数据类型；自动跳过回车、空格等无关字符 **（重点：不会读入键入的所有字符）；读入整数时，自动跳过除 +、-、0、1、2、3、4、5、6、7、8、9 外的所有字符**。
- **重载 >> 运算符使得能够直接通过 cin 读入自定义数据类型（大概率考）**

  定义一个矩形类，利用 cin 读入矩形的长宽。

```
1 class Rect{
```

```
    private:
            int wid,len;
        public:
                Rect(int w=0, int l=0){
                        wid=w,len=l;
                }
          friend istream & operator >> (istream &in, Rect &
              x){
        /*注意: 1.重载运算符时使用友元friend
           2.返回值为输入流的引用
           3.istream &in为输入流, 必须引用;
           4. Rect &x是读入的变量, 需要赋值, 必须引用*/
                in>>x.wid>>x.len;
                return in; //必须返回输入流
        }
};
```

在主函数中定义 Rect a, 调用 cin>>a 即可直接读入矩形长宽。

- 判断文件末尾（eof，end of file）

  方法 1:

  while(cin>>x); //注意分号

  cin 输入流有返回值，当读到文件末尾时，自动返回 false；因此可以直接读入 x，直至文件末尾。备注：在命令行界面中，输入完数据后，先 enter，再 Ctrl+Z，再 enter，可以代替文末。

  方法 2:

  cin.eof() 函数，读到文末返回 1，否则返回 0；

  例:

```
        while(!cin.eof())
                cin>>x;
```

可以一直读到 eof。

2.1.2 cout

<ifstream> 中的标准输出流，较为智能（例如忽略小数点后无意义的 0），但不如 printf 灵活。与 cin 的通病：速度极慢，读入 10^5 以上数据量时使用 scanf/printf/快速读入。

- cout 格式化输出需要包含头文件 <iomanip>。

  setw(x)：设定输出宽度为 x 个字符。不足的补空格，默认右对齐。超过 x 个字符不补空格，直接输出。

  cout<<setw(3)<<12«endl;

  输出：(先有一个空格)12

  setprecision(x): 设定浮点数输出小数点后 x 位，自动补零。

  cout<<setprecision(4)<<3.14<<endl;

  输出：3.1400

  fixed：操作符，表示后面的变量均按照当前的格式输出。

  cout<<fixed<<setprecision(4)<<3.14<<’ ‘<<2.71<<endl; //均输出四位小数

  输出：3.1400 2.7100

  hex: 十六进制，oct: 八进制，bin: 二进制

  cout<<hex<<15<<endl;

  输出：F (十六进制中 F 为 16)

- **重载 << 运算符，从而直接输出自定义类型（重要）**

  在前面定义的矩形类的 public 成员中加入以下内容：

```
friend ostream & operator << (ostream &out, const Rect &x){
/*注意：1.重载运算符时使用友元friend
       2.返回值输出流的引用（因为输出流会发生改变/刷新）
          3.ostream &out为输出流，必须引用；
*/
                          out<<"width="<<x.wid<<",length="<<x
                              .len<<endl;
                          return out;
                }
```

定义 Rect a, 调用 cout«a 即可直接输出长宽。

2.1.3 getline

读入一整行的 string 字符串（可以包含空格），以回车结束。
用法:

```
    string s;
    getline(cin, s);//cin是标准输入流, 读入的字符串赋值给s;
判断文末的方法:
        while(getline(cin,s)); //注意分号
```

读到 eof 时自动跳出循环

2.1.4 putchar/getchar

- 属于 C 语言的标准输入输出而非 C++。包含在头文件 <cstdio> 中。
- getchar() 读入一个字符并返回值。与 cin.get() 类似，可以读入任何字符，含空格、tab、回车。

  例: char c=getchar();
- putchar(x) 输出一个字符 x。x 必须为 char 类型。
- 利用 getchar() 实现快速读入整数（记忆即可）

```
    template<typename qRead> //模板, 适用int,short,long
        long类型
    inline void qr(qRead &s){
        char c=getchar();
        s=0;
        qRead f=1;
        for(;c<'0'||c>'9';c=getchar()){
            if(c=='-')
                f=-1; //判断负号
        }
        for(;c>='0'&&c<='9';c=getchar())
            s=s*10+(c-'0');
        s*=f;
    }
```

2.1.5 scaf(可以便捷的实现混合类型、复杂格式、指定特殊格式的输入)

(1) 属于 C 语言的标准输入输出而非 C++。包含在头文件 <cstdio> 中。

(2) 用法 scanf(" 格式说明串", & 变量 1, & 变量 2, ……);

变量必须加上取地址符 &，必须与格式说明符一一对应 (个数对应、变量类型也对应)。

格式说明符：

- %d：读取一个 int 整数。
- %f：读取一个 float 浮点数。
- %c：读取一个 char 字符。
- %s：读取一个字符串（char* 类型，直到遇到空白字符）。
- %x 或%X：读取一个十六进制整数（并以十进制形式赋值给变量）。
- %o：读取一个八进制整数。
- %u：读取一个无符号整数 (unsigned int)。
- %ld、%lf 等：用于读取长整型 (long)、长浮点型 (double) 等数据类型的值
- %lld：用于读入 long long 类型。
- `\n`回车, `\t`制表符, `\b`退格
- 一个 \ 表示转义符；要输入 \，在格式控制串中写两个右斜杠，即 \\

字段宽度和精度：可以在格式说明符前指定一个整数，表示要读取的最大字符数（对于字符串）或数字的最大宽度（对于整数和浮点数）。例如，%5d 表示最多读取 5 位数字的整数。

对于浮点数，可以使用. 后跟一个整数来指定精度，即小数点后的位数。例如，%6.2f 表示读取一个总共最多 6 位字符（包括小数点和小数部分）的浮点数，其中小数部分有 2 位。

忽略空白字符：

- scanf 在读取输入时会自动忽略任何前导的空白字符（空格、制表符、换行符）。

(3) 例:

```
int a; double b; char c;
scanf("%d %lf\n%c",&a,&b,&c);
读入整数a，双精度浮点数b，字符c，以空格、回车分隔

int a,b,c;
scanf("%d-%d=%d",&a,&b,&c);
```

可以读入一个减法运算式，获取被减数 a，减数 b，差 c，且 a,b,c, 均为整数，并且确保分隔符为 - 和 =，从而比 cin 方便。

使用%s 读取字符串时，scanf 会在遇到第一个空白字符时停止读取。如果需要读取包含空格的字符串，可以使用%[^\n] 这样的格式说明符，它表示读取直到遇到换行符为止的所有字符（注意，换行符本身不会被读取到字符串中）。

```
char s[100];
scanf("\%s",s);
```

字符数组名称就是指针指向的地址，无需加上 &

2.1.6 printf（可以便捷的实现混合类型、复杂格式、指定特殊格式的输出）

(1) 属于 C 语言的标准输入输出而非 C++。包含在头文件 <cstdio> 中。

(2) 用法 printf(" 格式说明串", 变量 1, 变量 2, ……);
变量必须与格式说明符一一对应（个数对应、变量类型也对应）。
格式说明符：

- %d：输出一个 int 整数。
- %f：输出一个 float 浮点数。
- %c：输出一个 char 字符。
- %s：输出一个字符串（char* 类型，直到遇到'\0'）。

- %x 或%X：输出一个十六进制整数。
- %o：输出一个八进制整数。
- %u：输出一个无符号整数 (unsigned int)。
- %ld、%lf 等：用于输出长整型 (long)、长/双精度浮点型 (double) 等数据类型的值
- %lld，用于输出 long long 类型。
- \n回车, \t制表符, \b退格 \a响铃（每台电脑的铃声不一样）
- 一个 \ 表示转义符；要输入 \ ，在格式控制串中写两个右斜杠，即 \\

字段宽度和精度：

可以在格式说明符前指定一个整数，表示要输出的最大字符数（对于字符串）或数字的最大宽度（对于整数和浮点数）。例如，%5d 表示以右对齐方式输出一个 5 位整数，位数不足的在数字左边用空格补齐。

对于浮点数，可以使用. 后跟一个整数来指定精度，即小数点后的位数。例如，%6.2f 表示输出一个总共最多 6 位字符（包括小数点和小数部分）的浮点数，其中小数部分有 2 位。%.1lf 表示输出一个双精度浮点数（double 类型），保留一位小数。

(3) 例：

```
int a; double b; char c; int d;
printf("%5d %.1lf\n%c\t%x",a,b,c,d);
```

输出十进制整数 a（宽度为 5 位），双精度浮点数 b（保留一位小数），单字符 c，十六进制整数 d，以空格、回车、tab 分隔。

```
int a,b,c;
printf("%d-%d=%d",a,b,c);
```

可以输出一个减法运算式，被减数 a，减数 b，差 c，且分隔符为 - 和 =，从而比 cout 方便。

输出一个 C 语言格式字符串。默认\0 为结束符。

```
char s[100];
printf("%s",s);
```

输出字符串时，需要传递字符串元素的首地址，因此直接传入字符串变量名即可。

无变量，直接输出字符串也可行，例如：printf(“hello world!\n”);

2.1.7 gets/puts

均是 C 语言的标准 I/O，头文件 <cstdio> 或 <stdio.h>

```
char *s = new char[str_len];
gets(s); // 读入一行字符串，直至\n结束；
puts(s); // 输出一行字符串，直至\0结束，最后自动换行
```

备注：对于 C++ 中的 string s，把 s 转换为 C 格式字符串：s.c_str() 返回一个 C 格式字符串。

2.1.8 freopen

属于 C 语言的文件输出输入，与标准输出输出（standard I/O）相对，头文件 <cstdio>；

用法：freopen(” 文件名”，打开方式，重定向到);

文件名: 需要完整路径，不写完整路径则默认 exe 程序所在文件夹。

打开方式，主要介绍三种：r 只读，w 只写（自动创建一个输出文件，如果已有，则直接覆盖），a 追加（在已有输出文件后面接着写）。

重定向到：标准输入 stdin，标准输出 stdout;

```
freopen("test.in","r",stdin);
freopen("test.out","w",stdout);
```

// 信息学竞赛中常用写法，放在 main() 函数开头处。表示将标准输入重定向到输入文件 test.in 中，标准输出重定向到输出文件 test.out 中。这样就不用修改程序中的 cin,cout,scanf,printf 等标准输入输出，直接以只读方式从 test.in 中读入数据，并输出到 test.out 中。

2.1.9 文件指针 FILE*

包含在头文件 <cstdio> 中。

FILE *fp=fopen(” 文件名”, 打开方式);

fopen() 用于打开一个文件，返回文件指针，赋值为 fp。

文件名、打开方式：要求同 freopen();

如果以只读或追加方式打开文件，文件不存在时，fopen() 会返回 NULL。

使用方式：

```
fscanf(文件指针, "格式说明串", &变量1, &变量2, …);
fprintf(文件指针, "格式说明串", 变量1, 变量2, …);
```

除了多了一个文件指针，其余使用方式与 scanf/printf 类似。

最后记得关闭文件：fclose(fp);

2.1.10 文件流 fstream

属于 C++ 的文件输入输出，头文件 <fstream> ，命名空间 std（写上 using namespace std 即可）；

istream fin(" 文件名");

定义一个输入文件流，并打开文件。之后使用方法与 cin 类似，把 cin 替换为 fin 即可。例如：

```
int x,y;
fin>>x>>y;
```

ostream fout(" 文件名");

定义一个输出文件流，并打开文件。之后使用方法与 cout 类似，把 cout 替换为 fout 即可。例如：

```
int x,y;
fout<<x<<y;
```

最后记得关闭文件：fin.close(); fout.close();

2.2 STL，Standard Template Library，标准模板库

封装了一些函数（算法，一般在头文件 <algorithm> 中）和容器（数据结构，一般有专门的头文件），属于 C++ 的专有内容，记得加上 using namespace std;

2.2.1 迭代器

C++ 版本指针，一般每种容器自带，无需额外头文件。

以数组 (vector) 为例：

```
    vector<int> vt;
    vector<int>:: iterator it=vt.begin(); //迭代器it，初始位置在开头
    for(; it!=vt.end(); it++){ //判断终止位置，必须用!=，不能用<=; it++移到下一位置
        cout<< (*it) << endl; //取值符号*
    }
```

对于数组: 指针就可以当作迭代器

```
    int a[100];
    a+1：指向a[1]的迭代器;

又如：vector v; vector::iterator it1,it2;
```

- 指向容器第一个元素的迭代器：v.begin()
- 指向容器最后一个元素的下一个位置的迭代器：v.end()
- 访问迭代器所指向的元素：int x=*it1
- 移动迭代器至后一个元素或前一个元素：it1++, ++it1, it1–, –it1
- 移动迭代器至后 x 个元素或前 x 个元素：it + x, it –x
- 只有 vector, string, deque 迭代器支持比较两个迭代器的位置关系：it1<=it2, it1>it2, it1==it2 ...

2.2.2 sort

平均 O(nlogn) 的快速排序，不稳定，头文件 <algorithm>

稳定排序 stable_sort();

sort(begin, end, cmp);

- begin 为排序数组的第一个元素的地址
- 数组名 arr 即为一个指针常量，可以得到数组第一个元素（arr[0]）的地址
- end 为排序数组的最后一个元素的下一个元素的地址
- 换而言之，sort 的数组地址的传入是左闭右开的区间
- 例如，sort(arr,arr+n) 是对 arr[0], arr[1], ..., arr[n-1] 进行排序
- sort(arr+1,arr+n+1) 是对 arr[1], arr[2], ..., arr[n] 进行排序
- cmp 是 sort 函数的排序规则 (比大小的函数)，可以不写，默认是从小到大排序的

• 如果需要从大到小可以换成 greater<T>()，T 为排序数组的变量类型

• 如果我们需要按照其他的排序规则，我们可以自定义一个 bool 类型的比较函数 cmp() 来传入

cmp 函数传入两个量 T x, T y，其中 T 表示排序数组的变量类型；同时 cmp 函数返回一个 bool 类型

• cmp 函数表示 sort 内部快速排序中的询问，当 cmp 返回 1 时表示 x 和 y 无需交换

• 对于最终的排序结果，对于任意两个在排序范围内的正整数 i,j (i < j)，都满足 cmp(arr[i], arr[j]) == 1 or arr[i] == arr[j]

• cmp 函数也可以理解为排序后的结果，按照 cmp 的顺序进行排列

一个例子：从大到小排序

```
bool cmp(int x, int y) { return x > y; }
int n = 5;
int arr[M] = {1, 5, 4, 2, 3};
sort(arr, arr + n, cmp);
// sort(arr, arr + n, greater());
```

2.2.3 lower_bound(begin,end,value,cmp())

找到在迭代器 begin(含) 至 end(不含) 之间第一个 >=value 的数的地址，cmp() 为比较规则，类似 sort()，不写 cmp 默认比较方式为 <=。

```
int a[100],t;
int p=lower_bound(a,a+100,t)-a; //-a目的是将地址转化为下
    标;
```

找不到则返回最后元素的下一个位置；若要找第一个 <=value 的数，用 p=upper_bound() 找第一个 >value 的，然后 p–（位置前移一个）。

upper_bound(begin,end,value,cmp())

找到在迭代器 begin(含) 至 end(不含) 之间第一个 >value 的数的地址，cmp() 为比较规则，类似 sort()，不写 cmp 默认比较方式为 <= 。

```
int a[100],t;
int p=upper_bound(a,a+100,t)-a; //-a目的是将地址转化为下
    标;
```

2.2.4 栈，头文件 <stack>

	通过数组实现	通过STL实现

具体实现：
1.定义
2.插入
3.删除
4.查询栈顶
5.获取元素个数
6.判断栈空
	`int Stack[N],top=0;`
	`Stack[++top]=x;`
	`top--;`
	`int ans=Stack[top];`
	`int Size=top;`
	`if(top==0) puts("Empty");`
		`stack<int> Stack;`
		`Stack.push(x);`
		`Stack.pop();`
		`int ans=Stack.top();`
		`int Size=Stack.size();`
		`if(Stack.empty()) puts("Empty");`

2.2.5 队列，头文件 <queue>

	通过数组实现	通过STL实现

具体实现：
1.定义
2.入队
3.出队
4.获取队首
5.获取元素个数
6.判断队空
	`int Queue[N],head=1,tail=0;`
	`Queue[++tail]=x;`
	`head++;`
	`int ans=Queue[head];`
	`int Size=tail-head+1;`
	`if(tail<head) puts("Empty");`
		`stack<int> Stack;`
		`Stack.push(x);`
		`Stack.pop();`
		`int y = Stack.top();`
		`int Size = Stack.size();`
		`if (Stack.empty()) puts("Empty");`

2.2.6 链表，头文件 <list>

定义：
插入元素：
删除元素：
遍历链表：

```
list<int> List;
List.insert(it, x); // it 为迭代器
List.erase(it);
```

```
for (list<int>::iterator it = List.begin(); it != List.end(); it++)
    cout << (*it) << endl;
```

2.2.7 集合 <set>, 可重集合 <multiset>

set：无重复元素的集合

定义：
插入元素：
删除元素：
查找元素：
求前驱：
求后继：
统计元素个数：
清空：
元素总个数：
遍历：

```
#include <set>
set<int> S;
S.insert(x);
S.erase(x);
S.find(x);
int pre = *(--S.lower_bound(x));
int suc = *(S.upper_bound(x));
int num = S.count(x); // 有 1 无 0
S.clear();
int Size = S.size();
for (auto x : S) cout << x << endl;
```

multiset：有重复元素的集合

定义：
删除一个元素x：
删除全部元素x：

```
multiset<int> S;
S.erase(S.find(x));
S.erase(x);
```

2.2.8 映射 <map>

- 映射、关联数组
- 以键值对的方式存储, 键值对：pair p<t1,t2>
- p.first 称为键,p.second 称为值
- 不允许键重复，并根据键自动排, 可以使用类似数组的方式去操作
- 类似 Python 的字典
- 内部采用红黑树维护
- 键值对可以理解为一一映射的关系
- 可以将 map 看一个数组，数组下标为键，数组存储的值为值
- 以键值对类型为 pair 创建一个关联数组，创建时关联数组为空：map mp;
- 向关联数组中插入键值对 (14, 0.1)：mp[14] = 0.1; 如果之前插入过键 14，则将之前的键值对替换为 (14, 0.1)
- 查询关联数组键 114514 对应的值：mp[114514]
- 将关联数组 114514 对应的值自增一：mp[114514]++;
- 查找键所对应的迭代器(不存在则返回 mp.end())：auto it=mp.find(114514)

- 删除迭代器所指向的键值对、删除键所对应的键值对：mp.erase(it), mp.erase(14)
- 清空关联数组：mp.clear();
- 访问、插入、删除、查询关联数组中的键值对的复杂度均为 O(logn)，n 为关联数组大小

map：键值无重复的映射 `#include <map>`

定义：
通过键值查找：
获取映射值：
插入键值对：

删除键值对：
遍历：

```cpp
map<int, int> mp;
map<int, int>::iterator it = mp.find(key);
cout << mp[key] << endl;
mp[key] = val; // 方法一
mp.insert(make_pair(key, val)); // 方法二
mp.erase(key);
for (auto x : mp)
    cout << x.first << " " << x.second << endl;
```

2.2.9 优先队列/堆 priority_queue，头文件 <queue>

定义大根堆：
定义小根堆：
插入元素：
删除堆顶：
获取堆顶：

```cpp
priority_queue<int> q;
priority_queue<int,vector<int>,greater<int> >q;
q.push(x);
q.pop();
q.top();
```

2.2.10 vector 动态数组（长度不限的数组，采用分块存储）

头文件 <vector>

- 动态数组，适用于预先不知道开多大的数组，或者想要轻快简便传参的情况
- 可以快速延长数组
- 以 int 类型创建一个动态数组，创建时动态数组为空：vector v;
- 以 int 类型创建一个动态数组，并预分配 10 的大小：vector v2(10);
- 往动态数组尾部加入一个元素 x ： v.push_back(x)
- 在动态数组尾部删除一个元素：v.pop_back()
- 查询动态数组的大小：v.size()

- 访问动态数组下标为 i 的元素（默认下标从 0 开始）：v[i];
- 将动态数组清空：v.clear()
- 上述操作的时间复杂度均为 O(1)

访问动态数组中的全部元素：

```
for (int i = 0; i < vc.size(); ++i)
    v[i];
for (vector::iterator it = v.begin(); it != v.end(); ++it)
    *it;
for (auto it = v.begin(); it != v.end(); ++it) // 需要 C++11 以上
    *it;
for (auto i:v) // 需要 C++11 以上
    i;
```

2.3 编程技巧

编程技巧是日积月累的，我一时半会回忆不出那么多，这里先写一点点：

2.3.1 位运算

按二进制位进行运算，效率高，可用于状态压缩 dp。

按位与 & (and) 与操作：

- 1&1 = 1
- 1&0 = 0
- 0&1 = 0
- 0&0 = 0

按位与：对每一个二进制位进行与操作例：

21 & 19 = 17 • 21 = (10101)2 • 19 = (10011)2 • 17 = (10001)2

按位或 | (or) 或操作：

- 1|1 = 1
- 1|0 = 1
- 0|1 = 1
- 0|0 = 0

按位或：对每一个二进制位进行或操作例：

21 | 19 = 23 • 21 = (10101)2 • 19 = (10011)2 • 23 = (10111)2

按位异或 $\oplus$ (xor) 异或操作：

- $1 \oplus 1 = 0$
- $1 \oplus 0 = 1$
- $0 \oplus 1 = 1$
- $0 \oplus 0 = 0$
- 相同为 0，相异为 1

按位异或：对每一个二进制位进行异或操作

例: $21 \oplus 19 = 6$ • $21 = (10101)2$ • $19 = (10011)2$ • $6 = (00110)2$

C++ 可以使用 ^ 表示异或

异或的性质：

- $0 \oplus a = a$
- $a \oplus a = 0$
- $a \oplus b = b \oplus a$
- $(a \oplus b) \oplus c = a \oplus (b \oplus c)$

按位非 ~ (not) 非操作：

- $\sim 1 = 0$
- $\sim 0 = 1$

按位非：对每一个二进制位进行非操作例:

(unsigned int) $\sim 13 = 4294967282$

- $13 = (00000000000000000000000000001101)_2$
- $4294967282 = (11111111111111111111111111110010)_2$

对于 int, long long 类型，采用补码存储，符号位也会被取反

2.3.2 宏定义

#define A B

即定义 A 为 B，编译器自动把程序中的 A 替换为 B，A、B 可以为任意语句。例如树状数组中常用

#define lowbit(x) x&(-x)

如果想要给变量类型名一个简短的名字，可以用 typedef:

typedef unsigned long long ull; // 短名字为 ull

2.3.3 骗分技巧（重要!!! 慎用!!!）

（1）如果你的程序运行太慢，可以自己开 O_2 优化加速，在程序开头加入预处理命令即可：#pragma G++ optimize(O_2)

（2）面向输出结果编程

如果你知道部分测试用例或可以人肉计算出题目的答案，直接输出即可。

（3）打表

例如求 10^6 以内所有质数的问题，而你不会高效的线性筛法，你可以先写一个判断 x 是否质数的函数 isprime(x)，枚举 x 为 1 10^6，是质数时就输出 x，然后把用大量时间运行出来的所有质数拷贝到需要提交的程序中，直接输出即可。

你提交的程序差不多是这样：

int prime[]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47……}

（4）修改老师编写好的模板

面向对象编程考试题中老师通常会写好类的定义。

如果你懒得写一些 public 函数，想要直接读取 private 成员变量，可以直接将老师写好的 private 改为 public，再改一下缩进，大概率看不出来。

（5）如果你不会写某种数据结构（例如链表），可以尝试把 STL 中封装好的 list 类翻出来看（找头文件即可），不过大概率看不懂 STL 中的写法。

3 面向对象的程序设计

3.1 类继承

还是以游戏飞机大战为例。对于我们的战机、敌机、飞行的子弹这三种对象（三个类），他们都是飞行的物体，能向前或向后运动。因此对于这三个类，我们都需要编写对应的运动函数以及碰撞检测函数，但是这些函数本质上都是相同的，编写三遍不符合代码重用原则，也容易出错。

由此我们想到：考虑到它们三个都是飞行的物体，能不能创建一个描述飞行的物体的基类，然后以此为基础，添加三者自己特有的功能，创建出三个派生类。这样三者共有的运动函数以及碰撞检测函数均只用编写一次，同时如果将来还要加入其他运动的物体（比如炸弹），也只需要在此基础上添加对应功能即可。

上述方法便是类继承，它可以让我们在已有类的基础上添加新的功能、变量或修改原类的行为。继承有助于将代码组织成层次结构，使得代码更加模块化，易于理解和维护。

3.1.1 类继承基本语法（公有继承）

类继承的一种常见情况是：a 是一种 b（‘a’ is a kind of ‘b’），我们称这种关系为 is-a 关系，可由 b 派生得到 a。例如学生是人，我们可用 person 类（基类）派生出 student 类（派生类）。派生类的声明方式如下：

```
class Person {
private:
    int age; // 私有成员变量 age

public:
    // 构造函数
    Person(int a=24) : age(a) {}
    // 公有函数 display 展示年龄
    void display() {
        cout << "Age: " << age << endl;
   }
}; //person基类

class Student : public Person {

}; //派生类声明

int main (){
    Student a;
    a.display(); //输出Age: 24
}
```

虽然我们没有为派生类编写任何函数，但是上述代码依然能输出。这是因为 Student 类是继承自 Person 类的。它自动继承了 Person 类的所有公有成员和保护成员。这意味着 Student 类的对象可以直接访问 Person 类的公有成员函数，包括 display() 函数。同时由于 Student 类没有自己的构造函数，编译器会自动调用基类 Person 的构造函数来初始化对象，因此 a 的值为 24。

现在我们为 student 类添加它自己的变量、构造函数与 display 函数。

在编写 student 类的构造函数以及重写 display 函数时，我们发现 age 作为 Person 类的私有成员变量，我们无法直接访问。同时相同的赋值及输出代码再写一遍也不符合代码重用原则。因此我们希望使用到 Person 类的构造函数与 display 函数。具体代码如下：

```
class Student : public Person {
private:
    int grade; // 私有成员变量 grade

public:
    // 构造函数。使用Person(a)，以a为参数调用Person的构造函数，将age的值初始化为a
    Student(int a=0, int g=0) : Person(a), grade(g) {}
    // 公有函数 display 展示年龄与年级
void display() {
    // 调用基类的 display 函数，语法为： 基类名+::(域操作符) + 函数名
        Person::display();
        cout << "Grade: " << grade << endl;
    }
};

int main() {
    // 创建 Student 对象
     Student student1(20, 10);
     Student student2;
     student1.display(); // 展示年龄与年级
     student2.display();
     return 0;
}
```

但是并不是所有的关系都是 is-a 关系，还有 has a 关系（例如午餐中含有水果，但是午餐不是水果）、use a 关系（吃午餐用筷子，但筷子不是午餐，午餐也不是筷子）、is like a 关系（冰箱与衣柜有相似之处，但是冰箱不是衣柜，衣柜也不是冰箱）和 is implement as a 关系（栈可以用数组实现，但是栈不是数组（无法下标访问））。描述上述关系的方法超出了本材料讨论范围，同学们可以自行查阅。

3.1.2 多次继承

一个类可以同时继承自多个基类, 只要用逗号将基类名称隔开, 并且要为每个基类指明 public/private/protected 继承 (注: 见下几节)。

3.1.3 多态继承

静态类型和动态类型 静态类型是指对程序进行编译分析时，所得到表达式的类型。相对应的概念是动态类型：如果某个指针或者引用指向一个多态对象 (派生类对象)，那么其最终派生对象为其动态类型。

以下是一个简单的例子:

```
struct Base { virtual ~Base() {} };
struct Derived : Base {};
Derived ObjectDerived; // 最终派生对象Base ObjectBase;
Base *P1 = &ObjectDerived; // (*P1) 的静态类型为Base, 因P1是
    Base型指针// (*P1) 的动态类型为Derived, 因其指向一个派生类
    对象
Base *P2 = &ObjectBase;// (*P1) 的静态类型为Base, 动态类型也是
    Base
```

引入多态继承的目的 虽然派生类和基类之间的关系是 IS-A 关系, 即派生类对象是一个基类。因此可以直接使用基类指针或引用指向派生类对象。但有些时候, 在派生类中某个基类方法和基类中的行为不完全一样, 但又想通过一种方法统一管理基类对象和派生类对象, 这就需要引入多态继承。

首先来看一个直观的想法, 直接通过基类指针试图调用派生类方法:

```
#include<bits/stdc++.h>
using namespace std;
struct Student{
private:
    string Name;
    string ID;
public:
    void print()
    {
        cout<<"Name: "<<Name<<endl;        cout<<"ID: "<<ID<<
            endl;
```

```
    }
};
struct InternationalStudent : Student{
    string National;
    void print()
    {
        Student::print(); // 必须使用域解析作用符,否则陷入递归
            cout<<"National: "<<National<<endl;
    }
};
int main(){    // 创建同时含有基类和派生类的数组
    vector<Student*> Stus;     for(int i = 0; i < 10; i++)
    {
        if(i&1)
            Stus.push_back(new Student);
        else
            Stus.push_back(new InternationalStudent);
    }
    for(size_t i = 0; i < Stus.size(); i++)
    {
        Stus.at(i) -> print();
    }
    // ...
}
```

上述代码能不能实现预期目标: 编译器根据指针指向对象的动态类型来调用对应方法? 答案是否定的,Stus.at(i)->print(); 一句永远只能调用基类方法, 也就是说, 即使输入的学生是个 InternationalStudent, 也不会打印其国籍。

在上述代码前提下,C++ 没有提供方法判断对象动态类型, 即使是 dynamic_cast 和 typeid 也要求类型是多态的. 一个取巧的方法是, 假如知道该数组的奇数位是基类类型, 偶数位是派生类类型, 那么可以直接使用 (无运行时类型检查) 强制类型转换来搞定这件事:

```
int main(){    // ...
    for(size_t i = 0; i < Stus.size(); i++) {
        if(i&1){
            Stus[i] -> print();// 已经判断为基类类型
        }
        else{
```

```
            static_cast<InternationalStudent*>(Stus[i])->print
                ();  // 或(InternationalStudent*)(Stus[i])->
                print();
        }
    }    // ...
}
```

这种方法并不推荐, 因为他有几个严重的问题:

1. 必须已知哪些元素是基类对象而哪些是派生类对象;

2. 如果不小心对基类对象作强制类型转换并且解引用, 就会导致未定义行为;

3. 在析构 (使用 delete) 时, 必须重新进行判断, 否则就会调用基类析构函数, 造成" 派生类" 部分资源的内存泄露。

引入多态 使用 virtual 函数说明符, 能将类的非静态成员函数声明为一个虚函数, 使得该函数的行为可以在派生类中被覆盖, 或者说, 当通过基类指针或引用调用该函数时, 编译器根据他们所指对象的动态类型调用该类型中的对应实现, 称最终调用的版本为最终覆盖函数. 一个类被称为多态的, 当其含有至少一个虚成员函数. 基于多态类的继承叫做多态继承. virtual 函数说明符一般放在返回类型前面, 但其实他们的顺序可以是任意的:

```
#include<bits/stdc++.h>
using namespace std;
struct Student{
    string Name;
    string ID;
    virtual void print() // void virtual print()也ok
    {
        cout<<"Name: "<<Name<<endl;
        cout<<"ID: "<<ID<<endl;
    }
    virtual ~Student() {}
};
struct InternationalStudent : Student{
    string National;
    void print(){
        Student::print();
        cout<<"National: "<<National<<endl;
```

```
18      }
19      ~InternationalStudent() {}
20  };
21  int main(){
22      // 创建同时含有基类和派生类的数组
23      vector<Student*> Stus;
24      for(int i = 0; i < 10; i++){
25          if(i&1)     Stus.push_back(new Student);
26          else    Stus.push_back(new InternationalStudent);
27      }
28      for(size_t i = 0; i < Stus.size(); i++){
29          Stus.at(i) -> print(); // 将根据Stus[i]的实际类型调用
30      }
31      for(size_t i = 0; i < Stus.size(); i++){
32      delete Stus[i];
33      }
34  }
```

在上述例子中, 基类的 print 函数和析构函数被声明为虚, 那么就可以在遍历时通过基类指针调用这两个函数:

1. 对于动态类型为 Student 的对象, 调用基类的 print() 和 Student()。

2. 对于动态类型为 InternationalStudent 的对象, 调用派生类的 print() 和 InternationalStudent()。

3.1.4 细节

派生类中的虚函数 注意到, 我们在派生类中没有声明 print 为虚的, 这是因为当基类的某个函数 vf 是虚函数时, 派生类的函数自动成为虚函数并且将覆盖 vf, 无论其声明中是否出现 virtual, 当且仅当其满足如下四个条件:

1. 名字相同, 即也为 vf

2. 形参列表 (不是返回类型)

3.const 限定符 // volatile 限定符

4.// 引用限定符

注: 带//标记的行是不在考纲范围内的内容, 但为列举严谨性而写出. 在考试出现的所有内容中均可以当作这些要求不存在, 下同。

这说明, 如果在基类中重载了某个虚函数, 要想在派生类中使用覆盖他们, 就必须为每个想要覆盖的函数写出实现 (其他函数会被隐藏, 见" 不满足

函数覆盖的情形")

析构函数 析构函数是个特例, 只要底层基类的析构函数是虚的, 那么派生类的析构函数也是虚的。若底层基类的析构函数非虚, 那么试图通过基类指针删除派生类对象是未定义行为, 即使这样做不会造成资源泄露。一个常用的方针是: 要么基类的析构函数是 public 且 virtual 的, 要么是 protected 且非 virtual 的. 后者可以保证不能通过基类指针删除派生类对象 (而必须通过派生类对象本身或其指针或其引用调用派生类的析构函数)

最终覆盖函数 如果派生类中没有能覆盖底层基类中 vf 的函数（包括在派生类中声明和通过多重继承得到的候选函数), 那么底层基类中的 vf 就是最终覆盖函数。

返回类型要求 上面的要求中并没有提到返回类型, 是因为对返回类型有特别要求: 若基类中某个函数 vf 是虚函数, 而派生类中声明了满足上述四个条件的函数, 那么两个函数的返回类型要么是相同的, 要么是协变的, 否则是编译错误。

若基类 vf 返回类型 T1 和派生类 vf 返回类型 T2 满足以下要求, 则他们协变 (Covariance):

1. 两个类型都是到类类型的单级指针或引用 (// 都为左值引用或都为右值引用.)

2.T1 所指向 (或被引用, 下同) 的类, 必须是 T2 所指向的类的无歧义可访问的直接或间接基类。

3.T2 必须较 T1 拥有同等或更少的 const(// volatile) 限定。

也就是说,T1 所指向的类在继承层级上的关系要比 T2 所指向的类更"基层"。

例如, 如果在不改变基类函数声明的情况下, 将 InternationalStudent 中的函数声明修改成

int print() // ... return 1;

就会编译不通过, 报错大意为: 覆盖函数的返回类型与基类类型既不相同, 也不协变。

插入一句完全的题外话, 在除了 main 函数之外的所有有返回值函数中, 语句执行到最后而没有 return 语句是未定义行为, 即便调用方无需获取其值. 在严格编译环境下 (treat all warnings as errors) 会抛出编译错误, 在

其他情况下可能会导致程序出现错误结果. 因此必须保证函数末尾有一句 return 语句。

有限定名字查找 vs 无限定名字查找 有限定名字查找，即函数名出现在作用域解析运算符:: 的右侧时，明确指明调用某个函数时使用哪一个版本（相对于调用者静态类型的直接或间接基类中各实现中的一个, 且可以为其本身），其语法为：对象名/基类引用. 基类名:: 函数名 (参数列表) 基类指针-> 基类名:: 函数名 (参数列表) 以下是一个例子：

```
// 类的定义省略
int main(){
InternationalStudent IStu;
Student *pStu = &IStu;
Student &rStu =  IStu;
// 虚调用
pStu->print(); // 派生类版本，打印国籍
rStu.print(); // 派生类版本，打印国籍
// 非虚调用
pStu->Student::print(); // 基类版本，不打印国籍
rStu.Student::print(); // 基类版本，不打印国籍
// 通过对象直接调用
IStu.Student::print(); // 基类版本，不打印国籍}
```

否则是无限定名字查找，这时应用虚调用。

不满足函数覆盖的情形 若名字相同但形参列表不同, 且基类函数为虚: 不覆盖基类函数, 但隐藏基类函数。通过基类指针调用时，始终调用基类版本 (派生类版本不可见, 试图调用派生类版本是编译错误)；通过派生类对象或其指针/引用调用时，始终调用派生类版本（基类版本不可见，试图调用基类版本是编译错误，唯通过有限定名字查找方式调用则仍成功）。

若一个函数拥有不止一个最终覆盖函数，那么是编译错误：

```
struct A{
    virtual void f();
};
struct D1 : A{
    void f(); // 覆盖A::f()
};
```

```
struct D2 : A{
    void f(); // 覆盖A::f()
};
struct D : D1, D2{}
```

以上代码中 D 同时从 D1 和 D2 两个地方继承了函数 void f(), 并且他们互不覆盖, 这样编译器无法确定要使用哪一个, 就会抛出二义性错误。

构造函数 构造函数不允许是虚的。派生类创建对象时, 先调用派生类的构造函数, 在执行其函数体前, 通过成员初始化列表调用基类的构造函数。这是一种调用, 而非继承, 也就是说派生类构造函数无意覆盖基类的构造函数, 不需要也不能把基类构造函数设置为虚。

友元 virtual 关键字只能用于类的非静态成员函数。友元函数不是成员函数, 因而不能为虚。但此方面设计问题可透过在友元中调用虚函数来解决。

3.1.5 静态联编和动态联编

指针和引用的向上转换和向下转换 一般来说,C++ 不允许 (隐式地) 将除了空指针常量之外的一种类型的地址赋给另一类型的指针, 亦不允许使用一种类型的引用指向另一种类型. 即使使用显式转换这样做了, 也不能安全地解引用转换后的指针. 但由于派生类中包含了 (至少) 一个完整的基类, 因而可以使用基类指针或引用指向派生类对象而无需使用强制类型转换, 这种隐式类型转换叫做向上强制转换 (upcasting). 相对应地, 将基类指针或引用转换为派生类指针或引用称为向下强制转换 (downcasting). 此转换必须通过强制 (显式) 类型转换完成. 一种常用的手段是使用 dynamic_cast, 该转换能安全地沿着继承层级向上, 向下及侧向转换指针和引用. 注意, 该手段要求多态继承。

这里仅仅简要地列出 dynamic_cast< 目标类型指针或引用类型 >(待转换指针/引用) 的常见用法:

```
struct Base{
    virtual void f() {}
};
struct Derived : Base {};
int main(){
    Derived D;
```

```
7       Base B;
8       Base *ptrB1 = &D;
9       Base *ptrB2 = &B;
10      if( dynamic_cast<Derived*>(ptrB1) ) // 成功转换
11      {
12      Derived *ptrD = dynamic_cast<Derived*>(ptrB1); // 拿到派生
            类指针
13      }
14      if( dynamic_cast<Derived*>(ptrB2) ) // 转换失败返回空指针,
            隐式转换到false
15      {         // 不会执行
16      }
17  }
```

当指针转换失败时, 会返回目标类型的空指针值, 这会经过 bool 隐式转换到 false, 因而判断会失败. 当引用转换时, 会抛出 std::bad_cast 错误, 需要使用 try-catch 块捕捉异常。

静态联编动态联编 编译器将源代码中函数调用与函数代码绑定的过程称为函数名联编. 对于一般函数, 函数名和形参列表就足以在编译时完成这一任务, 这种方式被称为静态联编. 但对于虚函数, 由于在编译时, 编译器无法确定调用方的动态类型, 因而无法确定应调用哪个版本的函数, 因而确定调用哪个函数将需要在运行时完成, 这种方法被称为动态联编. 编译器对虚调用使用动态联编, 对非虚调用使用静态联编。

为什么动态联编不是默认行为? 因为动态联编会引入虚函数表 (内存开销), 在运行时决定调用哪个函数会带来时间开销. 当某类无需用作基类或某函数在派生类中无需覆盖时, 没有必要使用动态联编. 因而按照 C++ 的设计原则, 在确实需要这样做的时候才使用虚函数和动态联编。

虚函数表 一种常见的动态联编实现是使用虚函数表. 编译器为多态类中添加一个隐藏成员, 该成员为一指向函数地址数组的指针, 称该函数地址数组为虚函数表, 虚函数表中储存了为类对象进行声明的虚函数的地址. 每个多态类有自己的表. 底层基类的虚函数表保存了其所有虚函数, 派生类的虚函数表保存最终覆盖函数的地址 (未覆盖的函数保存基类的地址, 覆盖了的函数保存覆盖函数的地址), 在派生类中新加入的虚函数亦被加入表中. 调用虚函数时, 程序查看储存在对象中的隐藏成员, 转向相应的函数地址表. 使用声

明顺序的第几个虚函数, 就调用函数地址表中的第几个函数。

使用 g++ 编译时, 加入编译指令-fdump-lang-class 可以查看虚函数表。

3.1.6 访问控制

类的 protected 成员 首先需要强调的是, 在 C++ 中, 结构体关键字 struct 和类关键字 class 的作用完全一致, 他们都声明一个类 (可以有成员, 构造函数等), 除了以关键字 struct 声明的类的成员在不加注明的情况下是 public 的, 以 class 声明的不加注明的情况下是 private. 同样地, 以这两种关键字声明的类在继承中有完全一样的行为, 除了以关键字 struct 声明的类在不加注明的情况下是 public 继承, 以 class 声明的类在不加注明的情况下是 private 继承。

以下重温 public 成员和 private 成员的可见性, 并引入 protected 可见性:

- public 成员总是可见;
- private 成员仅对同一个类的成员和友元可见, 允许是同一个类的两个实例;
- protected 成员允许以下两种情况的访问:

1. 当前类的成员和友元; 2. 派生自该类的任何类的成员和友元, 但仅仅在通过该派生类或该派生类的派生类的对象访问时允许。

以下是来自 cppreference 的参考代码, 解释了 protected 成员的特性:

```
1  struct Base{
2  protected:
3      int i;
4  private:
5      void g(Base& b, struct Derived& d);
6  };
7  struct Derived : Base{
8  friend void h(Base& b, Derived& d);
9  void f(Base& b, Derived& d) // 派生类的成员函数
10     {         ++d.i;  // OK: d 的类型是 Derived
11               ++i;    // OK: 隐含的 '*this' 的类型是 Derived//
12               ++b.i;  // 错误: 不能通过 Base 访问受保护成员
13               // (否则可能更改另一派生类, 假设为 Derived2 的基
                     实现)
14     }
```

```
};
void Base::g(Base& b, Derived& d) // 基类的成员函数{
    ++i;    // OK
    ++b.i;  // OK
    ++d.i;  // OK
}
void h(Base& b, Derived& d) // 派生类的友元{
    ++d.i; // OK: 派生类的友元可以通过派生类的对象访问受保护成
        员//
    ++b.i; // 错误: 派生类的友元并非基类的友元
}
void x(Base& b, Derived& d) // 非成员非友元{
    ++b.i; // 错误: 非成员不能访问
    ++d.i; // 错误: 非成员不能访问
}
```

public 继承 private 继承 protected 继承 这三种继承的语法如下:

```
class Base {
    public:
        virtual void f() {}
};
class D1 : public Base {}; // public继承
class D2 : protected Base {}; // protected继承
class D3 : private Base {}; // private继承
```

三种继承的作用如下: • 类使用 public 成员访问说明符从基类派生时，基类的所有公开成员可作为派生类的公开成员访问，基类的所有受保护成员可作为派生类的受保护成员访问（基类的私有成员始终不可访问，除非设为友元）。

• 当类使用 protected 成员访问说明符从基类派生时，基类的所有公开和受保护成员可作为派生类的受保护成员访问（基类的私有成员始终不可访问，除非设为友元）。

• 当类使用 private 成员访问说明符从基类派生时，基类的所有公开和受保护成员可作为派生类的私有成员访问（基类的私有成员始终不可访问，除非设为友元）。

公有继承最好地描述了 is-a 关系，无论继承多少层，派生类总是一个

(always is-a) 其直接或间接基类: 经公有继承派生的派生类指针/引用总是能向上转换强制到基类指针/引用。

受保护继承其次, 因为描述的关系是: 在派生类成员, 以及所有进一步派生的类中, 派生类 is-a 基类, 因而只有在派生类和派生类的派生类的成员内部, 才能对派生类指针/引用向上强制转换到基类指针/引用。

私有继承最次, 因为描述的关系是: 在派生类成员中 (而非进一步派生的类中), 派生类 is-a 基类, 因而只有在派生类的成员内部, 才能对派生类指针/引用向上强制转换到基类指针/引用。

3.1.7 抽象基类

抽象基类 (Abstract Base Class, ABC) 是为了解决不完美的 is-a 关系而存在的东西. 请看下例:

现在有个椭圆类

```
class Ellipse{
    private:
        double x; // 中心点坐标x
        double y; // 中心点坐标y
        double a; // 长半轴
        double b; // 短半轴
        double angle; // 长半轴与x轴正方向的夹角
    public:
        void Move(int xx, int yy) { x = xx, y = yy; }
        virtual double Area() const { rturn 3.14159 * a * b; }
        virtual void Rotate(double newAngle){ angle = newAngle;
            }
}
```

显然圆是椭圆, 但是使用多态公有继承会出现问题, 逐个分析成员:

1. 圆有 x 和 y, 这没有问题

2. 圆无需使用 a 和 b 来描述其形状, 因为 a=b=r

3. 圆没有角度, 进而 Rotate 不应在圆中实现

4. 圆的面积函数需要修改通过手段来隐藏这些不需要的内容不如重新写一个, 但这样就放弃了他们的共性部分.

C++ 允许引入抽象基类来抽象出这样一个” 共性部分”, 尽管他们没有任何物理意义. 这样的共性部分可以包括属性和方法, 进而通过继承来:

1. 增添属性 (例如 a,b,r)

2. 增添方法 (例如 Rotate 函数)

3. 为不同的派生提供同含义但实现不同的方法 (例如 Area 的不同实现)

首先, 考虑共性部分:

```
class BaseEllipse{
    private:
        double x; // 中心点坐标x
        double y; // 中心点坐标y
    public:
    BaseEllipse(double xx = 0, double yy = 0) : x(xx), y(yy) {}
    virtual ~BaseEllipse() {}
    void Move(int nx, int ny) { xx = nx, yy = ny; }
    virtual double Area() const = 0;
}
```

共性部分包括了 x,y 和具体的 Move 和一个抽象的 Area, 因为我们无法从已知参数求得 Area. 称形如 Area 函数为纯虚函数, 其语法为

virtual 返回类型函数名 (形参列表) 可选 cv 限定 = 0;

表示该函数留待派生类实现。

拥有至少一个纯虚函数的类称为抽象基类, 由于这样的类通常没有物理意义, 因而不允许创建抽象基类的对象, 但仍允许使用其指针和引用指向其派生类。

这样就可以进而实现两个派生类:

```
class Circle : public BaseEllipse {
    private:
        double r;
    public:
        Circle(double xx = 0, double yy = 0, double rr = 0) :
            BaseEllipse(xx, yy), r(rr) {}
        ~Circle() {}
        double Area(){
            return 3.14159 * r * r;
        }
}
class Ellipse : public BaseEllipse{
    private:
        double a;
```

```
14          double b;
15          double angle;
16      public: // 构造函数,析构函数留作练习
17          double Area(){            return 3.14159 * a * b;
18          }
19          double Rotate(double newAngle){
20          angle = newAngle;
21          }
22  }
```

关于纯虚函数的一个重要细节是: 纯虚函数将在派生类层级中保持纯虚, 直到某个派生类给出一个实现. 也就是说, 在某个派生类里, 通过继承或定义方式得到的最终覆盖函数中有至少一个纯虚函数, 该派生类就仍为抽象基类, 不能构建该派生类的对象。

抽象基类的理念更像一种规范, 他要求所有自其派生的类 (要想使用的话) 都至少实现了抽象基类的所有要求 (尽管可以通过 private+ 空实现来逃课。)

语法细节 1. 纯虚函数说明不能和定义 (函数体) 同时出现。

2. 不能把友元声明为纯虚。

3. 抽象类型不能作为形参类型, 函数返回类型或显式类型转换的目标类型, 以上检查在函数定义和调用点检查。

4. 可以为纯虚函数提供定义 (而且如果纯虚函数是析构函数就必须提供, 因为在销毁派生类时, 所有基类析构函数都会被调用): 派生类的成员函数可以自由地用有限定的函数标识 (:: 作用符) 调用抽象基类的纯虚函数. 此定义必须在类体之外提供 (函数声明的语法不允许纯说明符 =0 和函数体一起出现)。

5. 从抽象类的构造函数或析构函数中进行纯虚函数的虚调用是未定义行为 (无论纯虚函数是否拥有定义)。

3.1.8 继承和动态内存分配

析构顺序 对于用户定义或隐式定义的析构函数, 在析构函数体执行后, 编译器会以声明的逆序调用该类的所有非静态 (// 非变体) 数据成员的析构函数, 然后以构造的逆序调用所有直接 (// 非虚) 基类的析构函数 (继而调用它的成员与它的基类的析构函数, 以此类推).(// 最后, 如果此对象类型

是最终派生类, 那么调用所有虚基类的析构函数.) 上述规则在显式调用析构 (Object.~ClassName()) 时也应用。

派生类的析构 由于上述规则的存在, 无需在派生类析构函数函数体中对他的任何基类作出任何动作. 如果派生类中没有新增的使用动态内存分配的成员, 那么使用默认析构函数就可以. 否则, 必须为派生类提供析构函数定义, 并且该定义中只需要处理派生类构造函数中那些使用动态内存分配的成员, 而基类中那些动态内存分配的成员期望基类的析构函数能正确处理。

派生类的复制构造 如果派生类中没有新增的使用动态内存分配的成员, 那么使用默认复制构造函数就可以, 因为默认行为会调用基类的复制构造函数, 并将传入的派生类对象中的基类部分通过基类复制构造函数构造对象的基类部分; 其他平凡成员通过各自的复制构造函数完成构造。

如果派生类中有新增的使用动态内存分配的成员, 那么必须显式提供复制构造函数定义. 同上, 函数体中只需要处理派生类中那些使用动态内存分配的成员, 而基类部分直接使用成员初始化列表完成 (或省却这一步并调用基类的默认构造), 平凡成员可选在成员初始化列表中或者在函数体内完成。

请注意, 这里发生了从派生类引用到基类引用的向上转换, 其语法类似

Derived(const Derived& rhs) : Base(rhs) // 正常处理动态内存分配的复制构造

派生类的赋值构造运算符 这里的赋值构造运算符是指复制赋值运算符, 即为新对象重新开辟空间, 旧对象依旧保持有效。

如果派生类中没有新增的使用动态内存分配的成员, 那么使用默认赋值运算符就可以, 因为默认行为会调用基类的赋值运算符, 并将传入的派生类对象中的基类部分通过基类的赋值运算符赋值到调用方的基类部分, 这期待基类的赋值运算符能够正确处理基类中各成员的复制; 同时调用其他平凡成员各自的赋值运算符完成赋值。

如果派生类中有新增的使用动态内存分配的成员, 那么必须显式提供复制构造函数定义. 同上, 定义中只需要处理派生类中的成员, 而基类部分通过显式调用基类的赋值运算符完成. 注意, 要重新判断自赋值。其语法类似:

```
1 struct Base{
2     virtual void f() {}
3 };
```

```
struct Derived : Base{
    Derived& operator=(const Derived& rhs){
        if(this == & rhs) return *this; // 处理基类
    Base::operator=(rhs); // 处理其他派生类成员
    }
};
```

题外话, 标准库中提供的类都已经保证自赋值安全, 在所使用成员中如果没有裸露的指针, 可以省略自赋值检查。

3.2 认识模板

3.2.1 什么是模板？

模板是一个 C++ 实体, 其定义以下其一:

- 类模板: 一族类, 可以是子类
- 函数模板: 一族函数, 可以是成员函数
- // 别名模板: 一族类型的别名 (C++11 起)
- // 变量模板: 一族变量 (C++14 起)
- // 概念 (C++20 起) 在教学及期末考试中只会涉及到类模板和函数模板, 因而只会针对这两类模板展开叙述; 为求严谨, 在本节涉及到定义处会列举所有情况, 即使他们不在考纲内, 这部分内容以行注释标记。

对于上述定义中" 一族" 的定义, 可以类比数学中的曲线系:

$(x-a)^2+(y-b)^2=C : a,b$ 为已知常数

定义了一簇曲线 (一系列同心圆), 当给定一个 C 的时候, 上述方程转而指定一个特定的圆. 模板也是这样, 他拥有至少一个模板形参, 进而表示一族类或者函数, 当所有模板形参被指定时, 类模板指定一个特定的类而函数模板指定一个特定的函数。

3.2.2 声明模板

声明模板最简单的方式是使用 template 关键字,template 关键字代表接下来要声明一个模板, 紧随其后的尖括号内是模板形参序列, 用以指定模板形参, 模板形参分为以下三种

- 类型模板形参
- 非类型模板形参

- // 模板模板形参

一个模板形参序列可以是上述三种模板形参的随意组合, 两个模板形参之间以逗号隔开。

使用 typename 关键字或 class 关键字声明的模板形参是类型模板形参, 代表了这里需要填入一个类型, 两个关键字在声明模板形参时作用完全一致, 没有分别。

```
template <typename T>
class MyClass{}; // 声明一个类模板MyClass

template <class T>
void MyFunc(); // 声明一个函数模板MyFunc
```

这种方式引入一个类型 T, 他是一个等待确定的类型 (一如曲线系中的参数), 该类型 T 被当作已知类型, 并可以在声明的类模板或者函数模板中使用, 例如用来声明成员等:

```
template <typename T>
class Circle
{
    T radius; // 用T来声明一个成员
}

template <typename T>
T DoNothing(T a) // 用T来指定返回类型和函数参数类型
{
    return a;
}
```

非类型模板形参则不同, 他引入一个对象作为模板形参, 该对象的类型可以是:

- 引用 (// 左值引用)
- 整数类型
- 指针类型
- 其他特定类型, 详见模板形参与模板实参

例如:

```
template <unsigned int Size>
class Array{}; // 声明一个类模板Array
```

这时, 在模板内可以视作已经定义了一个这样的变量 Size, 该变量不可修改 (除非他是引用类型)。

将上面提到的两种模板形参综合起来, 我们可以定义一个简单的模板类:

```
template <typename T, unsigned int Size> // 一个类型模板形参,一个非类型模板实参
struct Array // Array是一个模板类
{
    T* pointer; // 声明一个类型为T*的指针作为成员

    Array() { pointer = new T[Size]; }
    // 构造函数: 开辟一个能装下Size个T的内存并赋给pointer

    ~Array() { delete[] pointer; }
    // 析构函数: 将初始化时分配的内存回收.
};
```

3.3 模板的实例化

一个模板声明不会在编译期生成具体的代码，当要对模板的一个特化生成具体的代码的时候，该模板被实例化（instantiated）。当语境要求一个完整类型（对类模板）或者要求函数定义存在（对函数模板）时，模板即被实例化，除非该模板已经被显式实例化或显式特化。其中类模板的实例化不会一并实例化其成员函数，除非这些成员函数亦被使用。

也就是说，模板的实例化分为：1. 显式实例化；

2. 隐式实例化。

3.3.1 类模板的实例化

对下面这个类模板而言：

```
template <typename T, unsigned int Size>
struct Array
{
    T* pointer;

    Array() { pointer = new T[Size]; }
```

```
7
8       ~Array() { delete[] pointer; }
9   };
```

可以显式地实例化，语法如下：

```
1   template
2   struct Array<int, 10>;
```

也可以让编译器在使用这个类时隐式实例化：

```
1   struct Array<int, 10> myArray;
```

这里 myArray 是一个 Array<int, 10> 类型的对象。

3.3.2 函数模板的实例化

对于下面这个函数模板而言：

```
1   template <typename T>
2   void swap(T& a, T& b)
3   {
4       T c = b;
5       b = a;
6       a = c;
7   }
```

可以显式地实例化，语法如下：

```
1   template
2   void swap<int>(T& a, T& b);
```

也可以在函数调用时隐式实例化：

```
1   int a;
2   int b;
3   swap(a,b); // 由编译器推导模板实参；或
4   swap<int> (a,b); // 显式地给出模板实参
```

3.3.3 非类型模板形参的限制

特别注意, 非类型模板形参的实例化必须遵守以下限制: 实参必须是编译期常量, 即必须在编译期能确定其值. 也就是说, 下面的代码不能通过编译:

```
template <size_t Cols, size_t Rows>
void func(int (&A)[Rows][Cols]) {}
int main()
{
    int a,b;
    cin>>a>>b;
    int Matrix[a][b];
    func(Matrix); // 错误, a和b不是编译期常量

    int Matrix2[3][4];
    func(Matrix2); // 正确, 3和4是编译期常量
}
```

3.3.4 实例化的时机

对类类型而言, 他仅在语境要求完整类型时实例化, 以下所有语境都要求完整类型 T:

1. 定义或调用返回类型为 T 或参数类型为 T 的函数;

2. 定义类型为 T 的对象;

3. 声明类型为 T 的非静态类数据成员 (不包括到 T 的引用和指针);

4.new 表达式用于类型为 T 的对象或元素类型为 T 的数组;

5. 对类型 T 进行隐式或显式类型转换;

6. 对类型 T* 或 T& 进行隐式转换、dynamic_cast 或 static_cast, 除了从 nullptr 或 (可有 const/volatile 限定的)void* 进行转换;

7. 对类型 T 的表达式应用类成员访问运算符 (即使用 T.a 访问成员 a);

8. 对类型 T 应用 typeid、sizeof 或 alignof 运算符;

9. 对指向 T 的指针应用算术运算符 (例如表达式 a+1, 其中 a 由 T *a; 定义);

10. 定义具有基类 T 的类;

11. 将值赋给类型 T 的左值;

12. 类型 T、T& 或 T* 的 catch 块。

13.// 对 T 类型左值应用左值到右值的转换

简单地来说，就是需要知道类型 T 的大小和布局时，要求一个完整类型. 而一个类型是完整的，当且仅当他不是以下任何一种不完整类型：

1. 可有 const/volatile 限定 void 类型；

2. 不完全定义的对象类型：

(1). 已经声明而未定义的类类型；

(2). 未知大小的数组；

(3). 元素类型为不完整类型的数组；

(4). 枚举类型，直到可确定底层类型为止。简单地来说，不能确定大小的类型不完整，即 sizeof(T) 无效的类型是不完整的。

以下通过几个例子来说明上面的内容：

```
class T; // 声明而未定义，类T不完整

int a[]; // 错误： 未知边界数组，类型不完整

T t[100]; // 错误： 元素类型不完整，不允许定义到不完整类型的数组。

enum A
{
    a = sizeof(A) // 错误： A不完整，因无法确定底层类型而无法确定大小
};
-----------------------------------------------------------------
class T
{
    int x;
}; // 给出类T的定义，从这里开始类型T完整。

T tt[100]; // 该数组类型是完整的。

enum A : int // 确定底层类型，A从这之后是完整类型
{
    a = sizeof(A); // 正确： A已经是完整类型
};
```

值得一提的是，虽然不允许声明未知边界数组和元素类型不完整的数组，但这两种写法可以作为函数形参而存在，这时他们仅仅是声明一个对应类型指针的语法糖，并不声明数组。

回归主题，用一些例子来探究是否发生实例化：对于这样一个类模板：

```
template <typename T>
class MyClass{};

MyClass<char> func1(Myclass<int>); // 声明函数不要求完整类型，不发生实例化

MyClass<char> func2(MyClass<int>) {} // 定义函数的返回类型和形参要求完整类型，两次实例化

MyClass<char>* func3(MyClass<int>*) {} // 不发生实例化，因为指针不要求完整类型

MyClass<int> *p; // 定义指针，不要求完整类型，不发生实例化

MyClass<int> a; // 定义对象要求完整类型，发生实例化

struct s
{
    Myclass<int> \&ref; // 引用成员不要求完整类型，不发生实例化
    MyClass<int> b; // 定义非静态成员要求完整类型，发生实例化
    static MyClass<int> c; // 定义静态成员不要求完整类型，不发生实例化
};

*p; // 解引用本身不要求完整类型，不发生实例化
func1(*p); // 发生两次实例化（MyClass<char>和Myclass<int>），函数调用要求完整类型
```

对于函数模板而言，调用函数的地方发生隐式实例化，若其未被显式实例化或显式特化。在多文件编译中，会独立实例化多个相同的类模板，可以使用 extern 关键字阻止隐式实例化，并在某个被所有使用该特化的文件都包含他的文件中显式实例化。但是，在链接阶段，不同文件产生的相同实例都会被合并。

```
A.h
template <class T>
class S{}; // 定义模板

A.cpp
#include "A.h"
template
S<int>; // 显式实例化

file1.cpp
#include "A.cpp"
extern template class S<int>; // 阻止隐式实例化
S<int> s; // 使用显式实例化的代码
```

3.3.5 实例化的一个常见错误

模板定义必须在隐式实例化点可见，因而定义模板通常在头文件中提供所有模板定义，也就是说，不宜在 A.h 中声明一个模板，而在 A.cpp 中定义他，这样仅使用#include"A.h" 不能正常使用模板，这一点和一般的“声明和实现相分离”思路不同。

3.4 模板的特化（具体化）

当提供了模板实参，或特定情况下编译器自动推导出这些实参时，他们替换对应的各模板形参，以获得一个模板的特化（又称具体化，Specialization）。

对类模板而言，允许

1. 显式（全）特化（简称特化，Explitcit (full) template specialization）：给出所有模板实参，这时类模板特化为一个类。

2. 偏特化（部分具体化，partially specialized）：给出部分模板实参，这时类模板特化为一个比原模板更特殊的模板。对函数模板而言，只允许显式（全）特化，这时函数模板指明一个函数对象。这里的特化，允许给模板针对特定类型另写一些代码，使得模板针对不同类型能有不同的行为。用一个例子来说明这部分内容：

设计一个复数类模板，实部和虚部的类型可能不同，要求实现复数的一般加法，除此之外：

1. 在实部元素类型为 unsigned int 时，两复数相加等于一般加法所得结果的共轭。

2. 在实部元素类型和虚部元素类型都为 char 时，两个复数相加永远等于 0。

虽然这个题目看起来很无厘头，但有时候卷子上就是会出现些无厘头的东西不是吗?

以下是主模板，他实现了复数的一般加法:

```
template <typename RealType, typename ImgType>
class Complex
{
    private:
        RealType real;
        ImgType img;
    public:
        Complex() = default; // 这是一种让编译器重新自动生成默认构造函数的方法

        Complex(const RealType& r, const ImgType& i) : real(r), img(i) {} // 一般构造函数

        Complex(const Complex& rhs) : real(rhs.real), img(rhs.img) {} // 复制构造函数

        Complex& operator=(const Complex& rhs) = default; // 使用编译器自动生成的复制赋值函数

        friend Complex operator+(const Complex& A, const Complex& B) // 一般的复数加法
        {
            return Complex(A.real + B.real, A.img + B.img);
        }

        friend ostream& operator<<(ostream& os, const Complex& C)
        {
            os<<C.real<<"+"<<C.img<<"i";
            return os;
        }
```

```
};
```

使用显式特化或者偏特化来特化一个类模板时，必须重写整个类，包括了构造函数等部分，编译器不会自动补充那些你认为没有修改的内容。以下是一个偏特化 Complex<unsigned int, T>，他处理了实部类型为 unsigned int 的特化：

```
template <typename ImgType>
class Complex<unsigned int, ImgType>
{
    private:
        unsigned int real;
        ImgType img;
    public:
        Complex() = default; // 这是一种让编译器重新自动生成默认构造函数的方法

        Complex(const unsigned int& r, const ImgType& i) : real(r), img(i) {} // 一般构造函数

        Complex(const Complex& rhs) : real(rhs.real), img(rhs.img) {} // 复制构造函数

        Complex& operator=(const Complex& rhs) = default; // 使用编译器自动生成的复制赋值函数

        friend Complex operator+(const Complex& A, const Complex& B) // 一般的复数加法
        {
            return Complex(A.real + B.real, -(A.img + B.img)); // 返回共轭
        }

        friend ostream& operator<<(ostream& os, const Complex& C)
        {
            os<<C.real<<"+"<<C.img<<"i";
            return os;
        }
```

```
};
```

偏特化仍旧产生一个模板，也就是说，他需要在使用时实例化，其实例化规则和上一节一样。

以下是一个全特化 Complex<char, char>, 他处理实部和虚部类型都是 char 的情况：

```
template <>
class Complex<char, char>
{
        private:
        char real;
        char img;
    public:
        Complex() = default; // 这是一种让编译器重新自动生成默认构造函数的方法

        Complex(const char& r, const char& i) : real(r), img(i) {} // 一般构造函数

        Complex(const Complex& rhs) : real(rhs.real), img(rhs.img) {} // 复制构造函数

        Complex& operator=(const Complex& rhs) = default; // 使用编译器自动生成的复制赋值函数

        friend Complex operator+(const Complex& A, const Complex& B) // 一般的复数加法
        {
            return Complex(0, 0); // 按题目返回0
        }

        friend ostream& operator<<(ostream& os, const Complex& C)
        {
            os<<C.real<<"+"<<C.img<<"i";
            return os;
        }
};
```

全特化由于给出了一个类模板实例化所需的所有参数，因而他给出了一个类。他不需要再显式或隐式实例化。

在调用哪个特化上，一方面可以显式指定，例如：

```
Complex<int, int>(1.0, 2.0); // 调用主模板，1.0和2.0通过隐式类型转换变成1和2

Complex<unsigned int, int>(2, 0); // 调用特化<unsigned int, int>

Complex<char, char>(0, 0); //调用特化<char, char>
另一方面可以由编译器推导，例如：
Complex(1, 2); // 由主模板推导成Complex<int, int>(1,2)
Complex(1u, 2); // 由特化Complex<unsigned int, T>推导成Complex<unsigned int, int>
Complex(char(1), char(1)) // 特化Complex<char, char>
```

需要注意，不同的特化是不同的类，例如 Complex<int, int> 就不能同 Complex<char, char> 相加，除非另外定义这样的运算符。

函数模板的特化是类似的，但只允许全特化：

```
template <typename T>
void swap(T& A, T& B)
{
...
}
这是对T=int的一个全特化：
template <>
void swap<int>(int &A, int &B)
{
...
}
```

而编译器可以推导实参，因而上面的写法等价于

```
template <>
void swap(int &A, int &B)
{
...
}
```

让两个形参的类型不一样会导致推导失败，产生编译错误。

这部分的考察到语法为止。

3.4.1 小技巧

1. 如果你忘记了特化的语法，打开 Dev-C++（如果考场上有提供类似的 IDE 的话），新建一个文件并用一个标准模板库容器如 vector，按住 Ctrl 并左键点击上面写的 vector 就（有可能）能翻到实现这些容器的代码，语法基本上都在里面了。

2. 如果你忘记了特化的语法，但是题目要求你针对不同类型作处理，可以善用 if 语句和不同类型的特性，例如：

T(0.5) = 0 -> T 是整型