论 C++语言在信息学竞赛中的应用

浙江省余姚中学 韩文弢

摘要 程序设计语言是信息学竞赛的一个重要组成部分，任何算法只有通过程序设计语言实现之后才能真正解决问题。C++语言凭借其高度的灵活性和强大的功能在大学生竞赛中被非常广泛地使用，在中学生竞赛中的使用也越来越广泛。本文分为三章，由浅入深地介绍了 C++语言的基础知识、面向对象编程基础和常用标准库、标准模板库（STL）。希望本文能够对想在信息学竞赛中使用 C++语言的读者有所帮助。

关键字 信息学竞赛 C++程序设计语言 标准模板库

目录

前言

信息学竞赛一般要求在一定的时间内，理解并分析题意，设计符合给定时间和空间复杂度要求的算法，并在计算机上使用一定的程序设计语言正确地实现算法。由于整个竞赛存在时间限制（特别是 ACM/ICPC 类竞赛，在解决问题数目相等的情况下以做题累计时间的多少来决定名次），因此所使用的程序设计语言能否正确、快速地实现算法对竞赛的成绩影响颇大。所以，编程复杂度越来越受到重视。编程复杂度在很大程度上与所选用的程序设计语言有关。一般信息学竞赛比较常用的程序设计语言有以下几种：BASIC、Pascal、C/C++、Java，它们的特点如下表所示：

	BASIC	Pascal	C++	Java
学习难度	容易	一般	较难	较难
语言特点	简单	严谨	灵活	高度面向对象
程序运行速度	慢	较快	快	慢
库函数功能	弱	一般	很强	强

在目前的中学生信息学竞赛中，Pascal 语言使用较为广泛。但是 C++语言凭借其本身所具有的高度的灵活性，以及它所带的库的强大功能，被越来越多的选手所使用。本文就是在这样一个背景下撰写的。在本文中，Pascal 语言以 Free Pascal[1]为准；而 C++语言则以标准的 ANSI/ISO C++[2]为准。

另外，需要注意的一点是，本文并不是参考资料。因此，对于 C++语言，本文不可能介绍得面面俱到，十分详细。如果想更加深入的学习 C++语言，可以阅读有关资料。

[1] 原因有二：1、标准 Pascal 所提供的东西实在是太少了；2、Free Pascal 在目前竞赛中使用较为广泛。推荐编译器：Free Pascal 1.0.10 (http://www.freepascal.org)。

[2] C++标准请参考《C++程序设计语言（特别版）》。推荐编译器：gcc 2.95.3 (http://www.gnu.org/software/gcc/gcc.html, http://www.delorie.com/djgpp)。

1 从 Pascal 到 C++

阅读本章的必要条件：会使用 Pascal 语言、了解一定的算法知识

由于目前中学生信息学竞赛中使用 Pascal 语言的选手较多，本章的目的就是使读者在已经掌握 Pascal 语言的情况下，能够尽快地从 Pascal 语言转型到 C++语言。如果你已经对 C++语言有一定的了解，可以跳过本章，直接阅读下一章。

1.1 世界，你好！

首先，让我们通过一个非常简单的 C++程序，来初步地了解 C++语言。

```
//: C01:Hello.cpp
// Hello, world!
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, world!" << endl;
} ///:~
```

这个程序的作用就是在屏幕上输出“Hello, world!”的字样。

第一行、第二行以及最后一行中，从“//”开始到行末的部分都是注释。在 C++中，注释有两种写法。一种是从 C 中继承的块注释，以“/*”开始，到“*/”结束。另一种就是新增的行注释，从“//”开始，到行末为止。

第三行中以“#”开始的内容被称为预处理指令，这一行的作用是把一个叫做 iostream[3] 的头文件包含到我们的程序中来，相当于 Pascal 中使用单元的 uses 语句。不过 Pascal 中默认使用 System 单元，其中包含了我们常用的绝大多数过程和函数，而 C++默认是不包含任何头文件的。另外，C 语言中的头文件都是以.h 结尾的，而标准的 C++提倡使用没有扩展名的头文件。

第四行让我们可以在程序中直接使用 std 名字空间内的标识符。std 名字空间包含了所有标准 C++提供的类和函数，为了简便起见，一般总在包含头文件的预处理命令后写上这一行。有关名字空间的详细内容可以参考有关资料。

第五行到第七行就是程序的主体。在这个程序中，我们定义了一个名字为 main 的函数。这个函数就相当于 Pascal 中的主程序。任何一个能够独立执行的 C++程序都需要有一个 main 函数。我们还可以看出，在 C++中，“{”和“}”就相当于 Pascal 中的“begin”和“end”。

第六行的作用就是向屏幕输出“Hello, world!”，并换行。其中，cout 就代表标准输出（一般就是屏幕），相当于 Pascal 中的 Output。“<<”在 C++中本来是位左移运算符，但是在这里被重新定义为插入符，作用是把一段内容插入到一个输出流中。endl 就是换行的意思。另外， C++中的字符串是用双引号括起来的，每一个语句也用“;”表示结束。

[3] 头文件 iostream 中包含了 C++中流的类和相关函数的定义，可以用来进行输入输出。详细的使用方法将在第二章中进行描述。

1.2 类型和定义

类型

与 Pascal 相似，C++也提供了基本类型以及程序员可以自定义的类型。下表展示了 C++中一些常用的基本类型以及与之对应的 Pascal 类型：

名称	C++类型	Pascal 类型	范围	大小
布尔型	bool	Boolean	true / false	1
字符型	char	Char	所有单字节字符	1
8 位有符号整型	char	ShortInt	-128 .. 127	1
8 位无符号整型	unsigned char	Byte	0 .. 255	1
16 位有符号整型	short	SmallInt	-32768 .. 32767	2
16 位无符号整型	unsigned short	Word	0 .. 65535	2
32 位有符号整型	int	LongInt	-2147483648 .. 2147483647	4
32 位无符号整型	unsigned int	LongWord	0 .. 4294967295	4
64 位有符号整型	long long	Int64	-2^{63} .. $2^{63}-1$	8
64 位无符号整型	unsigned long long	QWord	0 .. $2^{64}-1$	8
单精度浮点型	float	Single	1.17e-38 .. 3.40e38	4
双精度浮点型	double	Double	2.22e-308 .. 1.79e308	8
扩展浮点型	long double	Extended	3.36e-4932 .. 1.18e4932	10/12

布尔型

与 Pascal 相似，布尔型用来表示逻辑运算的结果。一个布尔型量的值只可能是 true 或者 false，true 的数值是 1，而 false 的数值是 0。与 Pascal 不同的是，在 C++中，很多其他类型的量都可以隐式地转化为布尔型，这时，非零的值都被转化成 true，而零被转化成 false。

字符型

单个字符的常数要用单引号括起来，一些不能显示的字符可以通过转义符来表示（参见下表）。另外，从上表中可以看出，在 C++中，字符型和单字节的整型实际上是等价的。举例来说，'A'的数值就是 65。

名称	ASCII 名称	C++名称
换行	NL(LF)	`\n`
水平制表符	HT	`\t`
竖直制表符	VT	`\v`
退格	BS	`\b`
回车	CR	`\r`
复位	FF	`\f`
铃声	BEL	`\a`
反斜杠	\	`\\`
问号	?	`\?`

单引号	'	\'
双引号	"	\"
八进制 ASCII 码	*ooo*	*ooo*
十六进制 ASCII 码	*hhh*	\x*hhh*

整型

C++中的整型和 Pascal 差不多，值得注意的是 C++中的各种整型的类型名实际上是由三部分组成的。首先是 signed 或者 unsigned，表示有符号或者无符号，默认为有符号；其次是长度的修饰词（char 除外），有 short、long、long long，默认为 long；最后就是 int，在前面有修饰词的情况下可以省略。

在 C++中，整型的常数是由前缀、数字序列以及后缀构成的。前缀用来描述常数所使用的进制，默认是十进制，以“0”开头的是八进制，以“0x”开头的是十六进制。后缀描述该常数的长度和有无符号，“l”表示 32 位整数，“ll”表示 64 位整数，“u”表示无符号整数，可以组合使用。另外，前缀和后缀中的字母大小写不限。

浮点型

C++中的浮点型和 Pascal 也差不多。浮点型的常数也有定点和浮点两种写法。定点数的写法基本与 Pascal 一致，区别在于在 C++中如果整数部分或者小数部分为零就可以省略，但是小数点必须保留。例如，“39.”和“.142857”在 C++中都是合法的。浮点数的写法与 Pascal 相同，底数和阶码之间用“e”分隔。另外，C++中的浮点数还可以带后缀，“f”表示单精度型，“l”表示扩展型，省略则表示双精度型。

枚举类型

与 Pascal 类似的，C++中也有枚举类型。例如：

```
enum keyword {ASM, AUTO, BREAK}
```

其中，该枚举类型的名称是 keyword，值可以是 ASM、AUTO 或者 BREAK。

C++语言没有提供内置的子界和集合类型。另外也没有提供内置字符串类型，但是字符串处理的方式却有两套。一套是从 C 语言中继承的字符指针的处理方式，比较麻烦，具体请参考有关资料；另一套则是 C++标准库提供的 string 类，比较方便，将会在 2.3 中详细描述。

为了了解各种类型实际所占的字节数，我们来看下面这个程序：

```
//: C01:Types.cpp
// Built-in types
#include <iostream>
using namespace std;
int main() {
    cout << "Size of bool: "
         << sizeof(bool) << endl;
    cout << "Size of char: "
```

```
        << sizeof(char) << endl;
    cout << "Size of unsigned char: "
        << sizeof(unsigned char) << endl;
    cout << "Size of short: "
        << sizeof(short) << endl;
    cout << "Size of unsigned short: "
        << sizeof(unsigned short) << endl;
    cout << "Size of int: "
        << sizeof(int) << endl;
    cout << "Size of unsigned int: "
        << sizeof(unsigned int) << endl;
    cout << "Size of long long: "
        << sizeof(long long) << endl;
    cout << "Size of unsigned long long: "
        << sizeof(unsigned long long) << endl;
    cout << "Size of float: "
        << sizeof(float) << endl;
    cout << "Size of double: "
        << sizeof(double) << endl;
    cout << "Size of long double: "
        << sizeof(long double) << endl;
} ///:~
```

其中，sizeof 的作用就是返回括号里的类型的大小（也可以是变量或者常量）。

定义

与 Pascal 不同，C++中的定义几乎可以出现在程序的任何地方。定义由四部分构成：修饰符（可选）、基本类型、主体以及初始值（可选），一般要以分号结尾。修饰符一般用来指定一些与类型无关的内容。主体由一个名字和一些修饰的操作符组成（各种修饰的操作符将在 1.3 中详细描述）。例如：

```
int number;
char c = 'H';
const double pi = 3.141592653579893;
```

分别定义了一个整型变量 number，一个被初始化为'H'的字符型变量 c，以及一个被初始化为 3.141592653579893 的双精度浮点型常量 pi。另外，同一类型的变量的定义可以写成一个语句，例如：

```
int a = 39, b, c = 23;
```

定义了 3 个整型变量 a、b、c，其中 a、c 分别被初始化为 39 和 23。

对于没有给出初始值的变量，如果它是一个全局变量（定义在任何函数之外的地方），那么将被初始化为零（与类型有关）；如果它是一个局部变量（定义在某个函数中），那么将不被初始化。

在 C++中，名字（标识符）可以由字母和数字组成，但第一个字符必须为字母（下划线“_”也被认为是一个字母）。**记住，与 Pascal 不同，C++对于大小写是敏感的。**

类型定义

如果在一个定义之前冠以“typedef”的话，那么这个定义就将变成一个类型定义。例如：

```
typedef unsigned long long QWord;
```

这时，QWord 将不再是一个变量的名字，而是一个类型的名字。这就相当于 Pascal 中 type 子句中所做的事情。接下来，你就可以用这个名字来定义新的变量。这样往往能够简化后面的定义。例如：

```
QWord x;
```

这个语句就定义了一个 QWord 类型（也就是 unsigned long long）类型的变量 x。

定义的生命周期

在 C++中，名字一旦被定义之后就可以被使用。但是，名字也有一定的生命周期（有效域）。当超过这个周期时，名字就会失效。同时，也就可以被重新定义了。

一般来说，全局变量的生命周期是整个程序。而局部变量的生命周期从被定义之处开始，到定义所出现的程序块结束为止（程序块是一段由花括号括起来的程序）。当局部变量和全局变量重名时，全局变量被隐藏起来，暂时不可用。

1.3 指针、数组和结构

指针

虽然 Pascal 中也有指针，但是 C++中的指针功能要比 Pascal[4]中的强大得多。在 C++中，对于任何一种类型 T，类型 T*就表示指向类型 T 的量的指针类型，这里*就是上一节所提到的类型的修饰操作符。例如：

```
char c = 'a';
char* p = &c;
```

其中，变量 p 就是一个指向 char 类型的量的指针，并被初始化为指向变量 c（这里，操作符“&”的作用就是求它后面所跟变量的内存地址）。

对一个指针而言，最基本的操作就是提取它所指向的内容。在 C++中，这个操作由操作符“*”完成。例如：

```
char c2 = *p;
```

这时，变量 c2 的值就是'a'。

指针变量还可以被赋值为 0，这时，指针变量不指向任何有效的存储单元。对一个值为 0 的指针变量进行取值运算会引起错误。

另外，C++中的指针还支持很多其他运算，将会在 1.4 中详细描述。

[4] 其实经过 Free Pascal 的扩展，Pascal 中指针的功能也已经和 C++中的差不多了。

动态内存分配

与 Pascal 类似的，C++也可以动态地分配内存。动态内存分配和回收是通过操作符 new 和 delete 来实现的，与 Pascal 中的 New 和 Dispose 类似，但用法不同，例如：

```
int* p = new int;
int* p2 = new int[10];
delete p;
delete[] p2;
```

第一个语句分配了一个整型变量的空间，并把首地址存在指针 p 中。而第二个语句则分配了连续 10 个整型变量的空间，并把第一个变量的首地址存在了指针 p2 中。后面两个语句分别回收了前面两个语句所分配的空间，注意第四个语句的 delete 后面有一对空的方括号，表示回收一组变量的空间。在分配连续多个变量时，个数可以用整型变量或者常量来描述。

指针常量和指向常量的指针

由于指针也是一种数据类型，因此有指针变量，也有指针常量。不过这个问题稍微有些复杂，因为指针除了本身之外还带有一个所指向的基础类型。因此，与指针有关的常量就分为指针常量和指向常量的指针。

所谓指针常量，就是说，指针本身是个常量，所指向的地址不能被改变，但是这个指针所指向的内容却可以被改变。也就是说，p 是常量，而*p 不是常量。指针常量的类型为 T*const。

所谓指向常量的指针，就是说，指针本身不是常量，所指向的地址可以被改变，但是这个指针所指向的内容却是一个常量，不能被改变。也就是说，p 不是常量，而*p 是常量。指向常量的指针的类型为 const T*或者 T const*。

当然，这两样东西可以结合起来，结果就得到了指向常量的指针常量（听上去有点复杂）。这时，p 和*p 都是常量，类型为 const T*const 或者 T const*const。

数组

在 C++中，对于任何一种类型 T，T[n]就表示元素类型为 T、有 n 个元素的数组类型。其中，n 必须是在编译时就能够确定的常量。例如：

```
float v[3];
char* a[32];
```

上面第一个语句定义了一个有 3 个单精度浮点型元素的数组 v，第二个语句定义了一个有 32 个字符指针型元素的数组 a。**记住，C++中的数组元素的编号都是从 0 开始的。**因此，数组 v 的 3 个元素分别为 v[0]、v[1]和 v[2]，数组 a 也同样。这可能会使熟悉 Pascal 的人感到很不习惯。

在 C++中，多维数组是通过元素为数组的数组来实现的。例如：

```
int d2[10][20];
```

就定义了一个二维数组，第一维的大小为 10，第二维的大小为 20。同 Pascal 类似的，C++中多维数组也是按照行优先的方式存储的。**另外，值得引起注意的是，C++中多维数**

组无论是定义，还是使用其中的某个元素，各个维都要分别用方括号括起来，而不能使用逗号分隔，这与 Pascal 是不同的。更有意思的是，逗号在 C++中也是一种操作符，它的作用是依次对逗号分隔的各个表达式进行计算，并最终返回最后一个表达式的值。因此，d2[1,2]并代表 d2[1][2]，而是等价于 d2[2]，表示一个大小为 20 的一维整型数组。从 Pascal 转到 C++的读者要避免出现这样的问题。**还有，绝大多数 C++编译器生成的程序在运行过程中不检查数组下标是否越界，需要读者自己注意这个问题。**

数组的初始化

数组的初始化和一般的变量类似，不过所有元素要用花括号括起来，元素之间用逗号分隔。例如：

```
int v1[] = {1, 2, 3, 4};
char v2[3] = {'a', 'b', 'c'};
int v3[8] = {1, 2, 3, 4};
```

这时，方括号内的数字可以省略。如果省略，数组的大小将由后面花括号中元素的个数决定。如果方括号内的数字比花括号中元素的个数大，则数组中最后缺少初始值的那些元素将被初始化为零。如果方括号内的数字比花括号中元素的个数小，则编译器将会报错。

数组与指针的联系

在 C++中，指针和数组有着密切的联系。一个数组的名字可以被用作指向该数组中第一个元素（或者说是第零个）的指针。例如：

```
int v[] = {1, 2, 3, 4};
int* p1 = v;
int* p2 = &v[0];
int* p3 = &v[4];
```

同时，指针也可以被当作一个数组来使用。例如：

```
int v[] = {1, 2, 3, 4};
int* p = v;
v[2] = 1;
p[2] = 1;
```

上面的最后两个语句是等价的。

引用

引用就是一个变量的别名。引用主要用于函数的参数以及重载操作符的返回值。对于任何一种类型 T，T&就表示类型 T 的引用类型。例如：

```
int i = 1;
int& r = i; // r 和 i 表示同一个整型变量
int x = r;  // x = 1
r = 2;      // i = 2
```

另外，引用在定义时一定要被初始化，以保证它的确代表了一定的存储空间。

结构

C++中的结构类型就相当于 Pascal 中的记录类型。例如：

```
struct character_type {
    char name[20];
    int level;
    int life;
    int mana;
    int experience;
    int money;
};
```

这就定义了一种名为 character_type 的类型。与 Pascal 类似的，记录类型的变量可用通过“.”来访问其中的各个成员。例如：

```
character_type necromancer;
necromancer.level = 99;
necromancer.money = 123456789;
character_type* p = &necromancer;
(*p).life = 1024;
p->mana = 2048;
```

从中我们还可以看出，要使用一个指向结构类型的量的指针所指向的元素的各个成员时，除了采用(*ptr).field 的方式外，我们还可以使用 ptr->field 的方式。但不能写成*ptr.field，因为“.”的优先级要比“*”高，这样写的意思是：ptr 是一个结构类型的变量，其中有一个名为 field 的成员，其类型是一种指针，我要使用 ptr.field 所指向的内容。

另外，在 C++中，链表的定义要比 Pascal 来得简便：

```
struct node_type {
    data_type data;
    node_type* next;
};
```

不过，有些时候相互嵌套还是不能避免的。例如：

```
struct list;    // 留到以后给出完整的定义
struct link {
    list* data;
    link* prev;
    link* next;
};
struct list {
    link* head;
};
```

这时候就必须先空地声明一次，把其中某一种类型的名字先提出来。这种情况在函数相互递归调用时也会遇到。

1.4 表达式和语句

表达式

和 Pascal 一样，C++中的表达式也是由操作符和操作数构成的。

操作数

与 Pascal 类似的，C++表达式中的操作数可以是常数、常量或者变量，也可以是函数的返回值。但与 Pascal 不同的是，C++中的某些操作符具有副作用（换句话说，这些操作符会对操作数的值进行修改），这些操作符只能作用在左值上。左值最初的意思是可以出现在赋值[5]号左边的东西，它的特点就是代表内存中一定的存储空间，能够被修改。

操作符

首先，我们来看下表：

操作符名称	C++操作符	Pascal 操作符
加法	+	+
减法	-	-
乘法	*	*
整数除法	/	div
实数除法	/	/
取余数	%	mod
小于	<	<
小于等于	<=	<=
大于	>	>
大于等于	>=	>=
相等	==	=
不等	!=	<>
位非	~	not
逻辑非	!	not
位与	&	and
逻辑与	&&	and
位或	|	or
逻辑或	||	or
位异或	^	xor
位左移	<<	shl
位右移	>>	shr

从中可以看出，C++语言最大的特点就是几乎所有的操作符都是由符号字符构成的。

[5] 很有意思的是，赋值在 C++中也是一种运算，这与 Pascal 是截然不同的。C++中的赋值号只有一个等号，作用是把等号右边的值赋给等号左边的左值，同时返回这个左值。这样定义就能使例如 a = b = c 这样的表达式有意义，意思就是先把 c 的值赋给 b，同时返回的 b 的值又被赋给了 a。

表中的这些操作符的用法与 Pascal 中的完全一致，但有 3 点值得我们注意：

1、C++中，**整数除法和实数除法都是由“/”来完成的**，当两个操作数都是整数时进行整数除法，当至少有一个是实数时进行实数除法；

2、C++中，**判断相等要写两个等号**，这是因为一个等号被用作赋值号，使用 Pascal 的读者一开始可能会很不习惯[6]；

3、C++中，位运算与逻辑运算的操作符是不同的。

此外，C++中还有一些 Pascal 所没有的操作符。

在 C++中，对于一个双目的操作符@，赋值表达式“x = x @ y”可以简写为“x @= y”。例如：“x += 39”就等价于“x = x + 39”。支持这种简写的操作符有：+、-、*、/、%、<<、>>、&、|、^等。

C++中还有“++”和“--”这两种运算，它们都是单目的，作用是对操作数进行加 1 和减 1。它们都有前缀和后缀两种写法。当使用前缀写法时，它们先对变量进行加 1（或者减 1），然后返回变量的值；而当使用后缀写法时，它们先返回变量的值，然后再对变量进行加 1（或者减 1）。例如：设整型变量 x 的值为 12，表达式“++x”的值是 13，而表达式“x++”的值是 12，但此后 x 的值已经变成了 13。

C++中还有一种很有趣的三目操作符“?:”。对于表达式“表达式 1 ? 表达式 2 : 表达式 3”，C++首先判断表达式 1 的真假，若为真，则返回表达式 2 的值，若为假，则返回表达式 3 的值。

C++中，类型转化也是一种运算，T(x)和(T)x 都能把 x 转化成类型 T 的值。

运算符+、-、++、--、+=、-=都可以用在指针上。“++”的作用是让指针指向内存中紧挨着当前所指向的存储空间的下一个位置。从地址上来看，增量取决于针指所指向的类型所占的大小。其他运算可以依次类推。同时“-”还可以用在两个相同类型的指针之间，返回它们之间的距离。另外，数组的下标操作实际上就是一种指针运算，p[i]就等价于*(p + i)，所以下标在 C++中也是一种运算。指针运算是 C++灵活性的一种体现。但同时，错误的指针运算会使程序试图访问一些禁止访问的地址，从而引起程序的崩溃。

关于操作符优先级的问题，我们只需要记住一些常用的就可以了。例如，乘除运算的优先级要高于加减运算。另外，**与 Pascal 不同，C++中逻辑运算的优先级要低于关系运算**，因此，复合条件中的一些括号就可以省略了。不清楚时可以通过加括号来控制运算顺序，具体的操作符优先级可以参考有关资料。

从操作符中可以看出，C++要比 Pascal 简洁、灵活。

语句

C++中的语句与 Pascal 略有区别。C++中的语句可以是定义，可以是单个的表达式，可以是复合语句，也可以是下文将要提到的条件或者循环语句。

Pascal 中的复合语句被“begin”和“end”所包括，C++中只要把它们分别换成左右花括号就可以了。

[6] 更麻烦的是，如上文所提到的，C++中的赋值也会返回一定的值。如果在条件判断语句中把判断相等写成一个等号的话，在语法上也是成立的，但意思却错了。不过好的编译器往往会对条件判断语句中的赋值给出警告。

条件语句

C++中的条件判断语句也是 if，同样也有带 else 和不带 else 两种形式。与 Pascal 不同的是，要判断的条件需要用括号括起来，并且随后没有 then。

```
if(条件)
    语句
```

或者

```
if(条件)
    语句
else
    语句
```

C++中还有与 Pascal 中 case 语句所对应的 switch 语句，其格式如下：

```
switch(表达式) {
    case 常量:
        语句
        break;
    case 常量:
        语句
        break;
    ...
    default:
        语句
}
```

其中，每一个 case 之后可以跟多个简单语句，且最后的一个 break 语句并不是必须的。如果一个 case 之后没有 break 语句，那么在执行完这个 case 之后的语句后，程序还将继续执行后面紧跟着的 case 后的语句，直到遇到 break 或者执行完整个 switch 为止。这样的好处就是可以同时处理多个值，例如：

```
switch(c) {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        // 处理数字
        break;
    ...
}
```

但如果你要求每一个 case 结束后都退出 switch 语句，就必须在后面加上 break 语句。

循环语句

与 Pascal 类似的，C++中也有 3 种循环语句，不过在格式上与 Pascal 相差很大。

while 循环的格式如下：

```
while(条件)
    语句
```

do ... while 循环的格式如下：

```
do
    语句
while(条件)
```

这两种循环都是当条件为真时一直执行，直到条件为假为止。区别是第一种是在每次循环开始前判断条件，而第二种则是在每次循环结束后判断条件。

for 循环可是说是 C++与 Pascal 区别最大的语句之一，同时也充分体现了 C++的灵活性。C++中 for 循环的格式如下：

```
for(初始化语句; 条件; 改变语句)
    语句
```

注意，**括号内各部分之间用分号隔开**。

在循环开始前，for 语句首先执行括号中的初始化语句，可以用来初始化循环变量，**在 for 的初始化语句中定义的变量生命周期到循环结束为止**。然后，在每次循环前，for 语句都要检查条件，若条件为真，则执行循环体，否则退出循环。在每次循环后，改变语句都会被执行，可以用来改变循环变量的值。例如：

```
int sum = 0;
for(int i = 1; i <= 100; i++)
    sum += i;
```

与 Pascal 类似的，3 种循环中都可以用 break 和 continue 语句来改变循环的流程。

什么时候使用分号

一般来说，C++中每个语句之后都要加分号表示结束（**包括 if 和 else 之间的那个语句，这和 Pascal 不同**）。但如果是复合语句，那么右花括号后面就不用加分号。当然，这只限于表示复合语句的花括号，如果是枚举或结构类型定义，或者是数组初始化，这些地方的右花括号之后原本该用分号的地方还是要用分号的。

1.5 函数

C++中，定义一个函数的格式如下：

```
返回类型 函数名(参数列表) {
    语句
}
```

与 Pascal 相比，C++中没有过程的概念，过程就是返回类型为 void（也就是什么都不返回）的函数。与 Pascal 不同，C++中的函数不能嵌套在另一个函数中定义。进入一个函数时，可用的变量只有全局变量和参数列表中指定的变量。另外，Pascal 中过程或函数没有参数时要把参数列表两侧的括号省略，而 C++中的括号则始终不能省略。参数列表中的各项之间用逗号分隔，形式和变量的定义相同。例如：

```
int f(int a, int b) {
    ...
}
```

很有意思的是，与下标操作类似的，函数调用在 C++中也算作是一种运算，参加运算的量有函数本身和实际的参数，运算的结果就是函数的返回值。

参数的传递方式

Pascal 中参数有值参和变参之分。在 C++中，一般的参数就是按照值参的形式传递的，而用引用形式定义的参数则是按照变参的形式传递的。另外，由于按照值参传递时要复制整个变量的值，在传递较大的变量时，如果函数中可以保证不改变变量的值，那么传递时可以采用常引用的方式。例如：

```
int f(int a, int& b, const int& c) {
    ...
}
```

这里，参数 a 是按照值参（直接传变量的值）的形式传递的，参数 b 是按照变参（传变量的地址）的形式传递的，而参数 c 传的也是变量的地址，但由于 const 的修饰，在函数 f 中，参数 c 的值不能被改变。函数 f 中任何试图改变 c 值的语句都将被编译器报错。此外，与 Pascal 类似的，在调用函数时，变参的位置只能用变量替代，或者说，只能用左值来替代。不过，如果参数是常引用的话，一般的表达式也可以，系统会自动为它创建一个临时变量。

数组在 C++中是按照指针的形式传递的。因此，任何在函数体内对数组中单个元素的改变都会引起被调用时的数组中元素的改变（因为它们在内存中的地址是相同的）。

函数值的返回

与 Pascal 不同，C++中，函数值的返回由 return 语句完成。return 语句在返回函数值的同时，会终止函数的执行，并回到调用该函数的位置继续执行程序。return 语句的格式为：

```
return 函数返回值;
```

另外，void 类型的函数可以直接用“`return;`”来终止函数的执行。

内联函数

一个普通的函数定义之前加上“`inline`”就变成了内联函数定义。这时，编译器会尽可能地把内联函数内的指令直接写在被调用的地方，而不是通过参数的传递和指令的跳转。因此，一般把简单并且会多次被使用的函数定义为内联函数。例如：

```
inline int max(int x, int y) {
    if(x > y)
        return x;
    else
        return y;
}
```

如果内联函数不能直接展开（比如，内联函数中有像循环这样较为复杂的结构），那么效果就和普通函数一样。

函数重载和默认参数

C++还支持函数重载和默认参数，它们都体现了 C++的简洁和灵活。

函数重载的意思就是相同名字的函数可以被多次定义，但每次定义时的参数列表都不同。当函数被调用时，编译器会根据实际调用时的参数列表决定到底调用哪个函数。一般地，被重载的函数用来完成同一件事情，只是参数略有不同。例如：

```
void print(int i) {
    cout << i << endl;
}
void print(const char* s) {
    cout << s << endl;
}
int main() {
    print(39);                  // 调用第一个函数
    print("Hello, world!");     // 调用第二个函数
}
```

所谓默认参数就是在定义函数时指定某些参数的默认值。在调用时，如果这些参数的值和默认值相同就可以被省略。但是这些被指定的参数只能出现在参数列表的最后面。例如：

```
void inc(int& x, int d = 1) {
    x += d;
}
int main() {
    int i = 1;
    inc(i);      // 等价于 inc(i, 1);
    inc(i, 3);
}
```

无论是函数重载或是默认参数，要注意的一点就是不要出现二义性。由于函数重载和默认参数在信息学竞赛中并不是十分有用，这里只简单地被提及。详细的内容请参考有关资料。

1.6 常用的库函数

正如前言所说，标准 C++提供了十分强大的库。在这一节，我们只介绍一些和 Pascal 所提供的标准过程和函数功能相似的库函数。

函数定义	头文件	作用	备注
void* memset(void* p, int b, size_t n);	cstring	把 p 所指向的连续 n 个字节的值都设置成 b	与 FillChar 类似，但要注意参数的顺序
void* memmove(void* p, const* q, size_t n);	cstring	把 q 所指向的连续 n 个字节的值复制到 p 所指向的位置	与 Move 类似，p、q 所指向的内存区域可以部分重叠

double atof(const char* p);
int atoi(const char* p);
long atol(const char* p);
	cstdlib	把字符串 p 转化成所表示的数	与 Val 类似

double fabs(double);	cmath	绝对值函数	与 Abs 类似
double ceil(double); double floor(double);	cmath	取整函数，前者为上取整，后者为下取整	
double sqrt(double);	cmath	平方根函数	与 Sqrt 类似
double pow(double d, double e);	cmath	幂函数，返回 d 的 e 次方	

double sin(double);
double cos(double);
double tan(double);
	cmath	三角函数	

double asin(double);
double acos(double);
double atan(double);
	cmath	反三角函数	

double atan2(double y, double x);	cmath	增强型反正切函数，返回点(x, y)的辐角	很有用，会根据点所在的象限调整弧度值

double sinh(double);
double cosh(double);
double tanh(double);
	cmath	双曲函数	

double exp(double);	cmath	指数函数，以 e 为底	与 Exp 类似
double log(double); double log10(double);	cmath	对数函数，前者以 e 为底，后者以 10 为底	与 Ln 类似

另外，标准 C++中并没有提供函数 Pi，要获得 Pi 的值一般这样做：

```
const double pi = acos(0.) * 2;
```

更详细的库函数用法请参考有关资料。

1.7 本章小结

在掌握前面几节的内容之后，基本上我们已经可以用 C++来完成平时我们在 Pascal 语言中所做的事情了。在本节中，我们将用 C++来解决一道实际的题目。

NOIP2003 提高组第三题 加分二叉树

题目描述

设一个 n 个节点的二叉树 T 的中序遍历为(1,2,3,...,n)，其中数字 1,2,3,...,n 为节点编号。每个节点都有一个分数（均为正整数），记第 j 个节点的分数为 d_j。T 及它的每个子树都有一个加分，任意一棵子树 S（包括 T 本身）的加分等于 S 的左子树的加分×S 的右子树的加分＋S 的根的分数。

若某棵子树为空，规定其加分为 1。叶子的加分就是叶节点本身的分数，不考虑它的空子树。

试求一棵符合中序遍历为(1,2,3,...,n)且加分最高的二叉树 T。要求输出 T 的最高加分和前序遍历。

输入格式

第 1 行：一个整数 n(n<30)，为节点个数。

地 2 行：n 个用空格隔开的整数，为每个节点的分数(分数<100)。

输出格式

第 1 行：一个整数，为最高加分（结果不会超过 4,000,000,000）。

第 2 行：n 个用空格隔开的整数，为该树的前序遍历。

输入样例

5

5 7 1 2 10

输出样例

145

3 1 2 4 5

为了简便起见，这里我们采用键盘输出、屏幕输出。C++中的屏幕输入也十分简单，cin 表示标准输入（一般是键盘），对 cin 进行提取（用“>>”）操作就可以实现输入了，具体请参考程序中的例子。

解：从题目的描述中，我们可以看出，在求 T 的最高加分的过程中，有很多求各棵子树加分的子问题，且易知这些子问题也同样满足最优性要求的。由于在中序遍历中，同一棵子树的各个节点的编号是连续相邻的，我们只要指定编号的起始位置和终止位置就可以把一个子问题描述完整。因此，本题我们可以采用动态规划的思想加以解决。设 $best_{i,j}$ 表示从第 i 个节点开始到第 j 个节点为止的子树加分的最高值，由题意得：

$$best_{i,j}=\begin{cases} d_j (1\le i=j\le n) \\ \max\{best_{i+1,j}+d_i, best_{i,j-1}+d_j, \max_{i<k<j}\{best_{i,k-1}best_{k+1,j}+d_k\}\} (1\le i<j\le n) \end{cases}$$

由于题目还要求我们输出加分最高的二叉树的前序遍历，因此我们还要用 $root_{i,j}$ 来记录从第 i 个节点开始到第 j 个节点为止的子树的根节点的编号，最后，我们可以用递归的方法来输出所求的前序遍历。下面是完整的程序：

```
//: C01:Tree.cpp
// Binary Tree with Scores (NOIP2003 Advanced Division)
#include <iostream>
#include <cstring>
using namespace std;
const int max_nodes = 30;
int n;
unsigned int best[max_nodes][max_nodes];
int root[max_nodes][max_nodes];
bool first;
void visit(int i, int j) {
    if(i > j)
        return;
    if(first)
        first = false;
    else
        cout << ' ';
    cout << root[i][j] + 1;
    visit(i, root[i][j] - 1);
    visit(root[i][j] + 1, j);
}
int main() {
    cin >> n;
    memset(best, 0, sizeof(best));
    memset(root, 0xFF, sizeof(root));
    for(int i = 0; i < n; i++) {
        cin >> best[i][i];
        root[i][i] = i;
```

```
    }
    for(int d = 1; d < n; d++)
        for(int i = 0; i + d < n; i++) {
            int j = i + d;
            if(best[i + 1][j] > best[i][j - 1]) {
                best[i][j] = best[i][i] + best[i + 1][j];
                root[i][j] = i;
            }
            else {
                best[i][j] = best[j][j] + best[i][j - 1];
                root[i][j] = j;
            }
            for(int k = i + 1; k < j; k++) {
                unsigned int value = best[k][k]
                    + best[i][k - 1] * best[k + 1][j];
                if(value > best[i][j]) {
                    best[i][j] = value;
                    root[i][j] = k;
                }
            }
        }
    cout << best[0][n - 1] << endl;
    first = true;
    visit(0, n - 1);
    cout << endl;
}
///:~
```

需要注意的一点是，由于 C++中的数组下标是从 0 开始编号的，所以程序中要进行一定的调整。

本章的内容就写到这里。通过本章的学习，我们可以发现 C++与 Pascal 之间存在很多相似之处，同时也存在很多差异。我们要使用类比的方法，以相似之处为基础，进而掌握差异所在，并通过不断地练习，最终达到熟练掌握 C++语言的目的。

2 深入 C++语言

阅读本章的必要条件：会使用 C++编写简单的程序、了解一定的算法知识

在本章中，我们将介绍 C++语言中的类和对象，以及 C++标准库中一些常用的类。

2.1 类

类在信息学竞赛中的直接应用并不是很多。但由于 C++所提供的强大的库都是建立在类的基础上的，因此在这一节，我们来简单地了解一下 C++中的类。在本节中，类和对象这两个名词出现的频率很高。所谓类，其实就是一种类型，而对象则是以某一种类为类型的一个变量。

类的定义

简单地说，类是对结构类型的一种扩展。结构类型中加上一些函数，就成了最简单的类。例如：

```
struct Date {
    int d, m, y;
    void init(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int n);
    void add_day(int n);
};
```

这时，我们可以像访问成员变量一样来访问成员函数：

```
Date national_day;
national_day.init(1, 10, 2003);
Date next_day;
next_day = national_day;
next_day.add_day(1);
```

另外，在定义类的成员函数时，为了避免不同的类之间函数名的重复，要在函数名之前加上类名，并用“::”分隔。例如：

```
void Date::init(int dd, int mm, int yy) {
    d = dd;
    m = mm;
    y = yy;
}
```

在类的成员函数中使用类的成员变量时，只要直接写出变量名即可。

操作权限

面向对象编程的一个重要思想就是数据的封装，也就是把对象中的数据隐藏起来，只通过成员函数来对对象进行操作。这时，就要对类中的成员指定操作权限。例如：

```
struct Date {
private:
    int d, m, y;
public:
    void init(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int n);
    void add_day(int n);
};
```

这里，成员变量 d、m、y 的操作权限为私有，只有类本身的成员函数能够访问它们。而下面的四个成员函数的操作权限为公有，都可以被外界访问。当然，并不是成员变量就一定要是私有的，而成员函数就一定要是公有的，一些内部使用的成员函数也可以是私有的，而成员变量也可以是公有的。

另外，在定义类时，我们一般用 `class` 来代替 `struct`。例如：

```
class Date {
private:
    int d, m, y;
public:
    void init(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int n);
    void add_day(int n);
};
```

`class` 与 `struct` 唯一的区别就是，在不指定操作权限时，`struct` 的默认操作权限为公有，而 `class` 的默认操作权限为私有。

构造函数和析构函数

在一个类中，可以定义两种特殊的函数。一种被称为构造函数，该函数没有返回类型，名字与类名相同，会在对象创建时自动调用，往往用来完成一些对象初始化的工作，在它的参数列表后可以跟初始化列表；另一种被称为析构函数，该函数也没有返回类型，名字是由类名之前加一个“~”符号构成的，会在对象死亡（也就是超出生命周期）时自动调用，往往用来完成一些扫尾工作。构造函数可以带一些参数，并且可以重载，而析构函数不带任何参数。例如：

```
class Date {
public:
    Date(int dd, int mm, int yy);
    Date(char* date);
    ~Date();
```

```
};
Date::Date(int dd, int mm, int yy) : d(dd), m(mm), y(yy) {
}
Date national_day(1, 10, 2003);
Date christmas("December 25th, 2003");
```

在 Date 类构造函数的实现中，冒号后面的就是初始化列表，可以用来初始化一些成员变量。当然，你也可以在构造函数的函数体中初始化成员变量，不过一般推荐使用初始化列表来完成这项工作。

静态成员

在一个类中，有些成员变量是这个类的所有对象所共享的。我们可以用全局变量来实现，但这样做会破坏封装。为了解决这个问题，我们可以把这些成员变量定义为静态成员变量。还有一些成员函数，它们可以在脱离具体对象的情况下被调用，我们可以把它们定义为静态成员函数。这些静态成员的定义只要在一般成员的定义之前加上“static”就可以了。

另外，静态成员同样也服从类的操作权限，静态成员变量要在类定义的外面被初始化，静态成员函数不能使用一般（非静态）的成员变量，也不能调用一般的成员函数，但可以使用静态成员变量，也可以调用其它静态成员函数。一般的成员函数可以使用静态成员变量，也可以调用静态成员函数。我们通过一个简单的程序来演示静态成员的定义与使用。

```
//: C02:Counter.cpp
// Static members in class
#include <iostream>
using namespace std;
class Counter {
public:
    Counter(int id2);
    ~Counter();
    int get_id();
    static int get_tot();
private:
    int id;
    static int tot;
};
int Counter::tot = 0;
Counter::Counter(int id2) :id(id2) {
    tot++;
}
Counter::~Counter() {
    tot--;
}
int Counter::get_id() {
    return id;
```

```
}
int Counter::get_tot() {
    return tot;
}
int main() {
    Counter c1(0), c2(1);
    cout << c1.get_id() << endl;
    cout << c1.get_tot() << endl;
    Counter c3(2);
    cout << Counter::get_tot() << endl;
} ///:~
```

常成员函数

在一个类中，有些成员函数不会改变对象的状态，比如上面一个程序中 Counter 类中的 get_id()成员函数。这时，我们就可以在这些成员函数定义中的参数列表后面加上“const”，把它们声明为常成员函数。在常成员函数中，我们不能写任何修改成员变量的语句，否则编译器会报错。

另外，对象的常量只能使用它的常成员函数，比如上一章中函数调用时提到的常引用的传递方法，在函数内部就只能使用对象的常成员函数。静态成员函数不能被定义为常成员函数。我们把上面的程序加以修改：

```
//: C02:Counter2.cpp
// Constant member functions in class
#include <iostream>
using namespace std;
class Counter {
public:
    Counter(int id2) : id(id2) {
        tot++;
    }
    ~Counter() {
        tot--;
    }
    int get_id() const {
        return id;
    };
    static int get_tot() {
        return tot;
    }
private:
    int id;
    static int tot;
};
```

```
int Counter::tot = 0;
int main() {
    const Counter c1(0), c2(1);
    cout << c1.get_id() << endl;
    cout << c1.get_tot() << endl;
    Counter c3(2);
    cout << Counter::get_tot() << endl;
} ///:~
```

从上面的程序中，我们还可以看出，成员函数的实现也可以在类定义中完成。如果在类定义中完成，编译器会尽量用内联的方式来处理它们。一般把较短的成员函数的完整实现放在类定义中，而较长的成员函数则在类定义的外面实现。

this 指针

有些时候，我们要在对象的成员函数中引用对象本身，这时就要用到 this 指针。在成员函数中，this 指针是一个指向该对象类型的指针，并总指向当前的对象。另外，当类的成员函数中有局部变量与类的成员变量重名时，直接写变量名代表局部变量。如果要访问成员变量，则需通过 this 来引用（this->xxx）。

以上就是对 C++中类的简单介绍。实际上，类和对象最大的用处是大大简化了大型软件编写和维护的过程。不过在信息学中适当地使用类和对象同样也能降低编程复杂度。

2.2 操作符重载

操作符重载的意思就是对语言提供的操作符进行重新定义。这项技术最初的目的只是为了简化代码。比如我们要写一个表示高精度整数的类，这个类的对象要支持各种算术运算。如果把这些运算都写成函数，代码就会很不好看。例如，x + y * z 这个简单的表达式也许就要写成 add(x, multiply(y, z))。如果可以重新定义语言提供的操作符，那么我们就可以像使用语言内置的数据类型一样来使用自己定义的类型了。

正如上一章所说，C++中的操作符十分丰富，连下标和函数调用都可以被视为是一种操作符。在进行操作符重载时，我们不但可以重新定义普通的算术运算符，几乎所有的操作符都可以被重新定义。

当然，操作符重载也有一定的约束条件。首先，我们只能重载 C++中已经存在的操作符，而不能自己定义新的操作符；其次，被重载的操作符必须至少有一个操作数的类型是用户定义的，也就是说，不能重载操作数都是语言内置类型的操作符；再次，重载后的操作符优先等级与原来的一样；另外，有些操作符不能被重载，如：“.”、“?:”等。

一般地说，被重载的操作符所完成的功能应该与这个操作符原来的意思相符。当然，这并不是绝对的。像我们已经接触过的流的输入输出中，提取符（>>）与插入符（<<）就完全改变了操作符原来的作用，但这个改动十分巧妙。

我们可以把操作符看成是一种特殊的函数，操作数就是这种函数的参数。因此，我们可以像定义函数一样来重载操作符。操作符的函数名就是“operator”加上操作符本身。对于一种双目操作符@，表达式 a @ b 就等价于 operator @(a, b)，或者 a.operator @(b)；而对于一种单目操作符@，表达式@a 就等价于 operator @(a)，或者 a.operator @()。也就是说，重载操作符既可以定义为全局函数，也可以定义为类的成员函数。例如：

```
class complex {
    double re, im;
public:
    complex(double r, double i) : re(r), im(i) {}
    double real() const { return re; }
    double imag() const { return im; }
    complex operator +(const complex& x) {
        return complex(re + x.re, im + x.im);
    }
};
complex operator -(const complex& x, const complex& y) {
    return complex(x.real() - y.real(), x.imag() - y.imag());
}
```

在上面的程序段中，我们把加法操作符定义成了类的成员函数，而把减法操作符定义成了全局变量。通常把普通的双目操作符定义为全局函数，把单目操作符定义为类的成员函数。但是返回结果是左值的操作符（赋值类操作符等），一般也定义为类的成员函数，它们的返回类型必须是引用。

几种特殊操作符的重载

下标操作符

下标操作符“[]”必须被定义为类的成员函数，一般在定义类似数组的类时被重载。有趣的是，被重载后方括号内的值可以不是整型的，这也为 STL 中的映射类提供了方便。

函数调用操作符

函数调用操作符“()”必须被定义为类的成员函数，参数的个数和类型都没有限制。一般来说，函数调用操作符通常用来定义多维数组或者是可以像函数那样被使用的对象。

增减操作符

增减操作符“++”和“--”有前缀和后缀两种用法。前缀用法的重载和普通的单目操作符相同，而后缀用法的重载则被人为地加上了一个整型的哑参数，这样就可以根据函数重载的法则与前缀用法区别开来。例如：

```
class counter {
    int v;
public:
    counter(int val = 0) : v(val) { }
    int get_value() cosnt { return v; }
```

```
    counter& operator ++() {   // 前缀用法
        v++;
        return *this;
    }
    counter operator ++(int) { // 后缀用法
        counter temp = *this;
        v++;
        return temp;
    }
};
```

操作符重载具体的例子我们会在以后的内容中继续介绍。

2.3 字符串

C++的标准库中提供了 string 类来进行一般的字符串处理。string 类的定义在头文件 string 中，使用时不要忘了把这个头文件包含进去。

实际上，C++为了支持其他的字符集，先用模板实现了一个一般化的字符串类 basic_string，而 string 类只不过是它的一个特化版本（字符类型为 char）。为了简便起见，本节中认为 string 是一个单独的类，并对它的一些常用函数进行介绍。

首先，我们来讲一些 string 类中基础的东西。string 类的字符串理论上没有长度限制，会根据实际字符串的长度自动分配空间。string::size_type 类型实际上就是 unsigned int 类型，用来描述字符串中的位置和长度。string::npos 是 size_type 类型的静态成员常量，它的值是 unsigned int 的最大值，通常用来描述最大的长度。对于一个 string 类的对象 s，我们可以用 s[i]来访问 s 中的第 i 个字符，与 C++中的数组下标相类似的，字符的编号也同样是从 0 开始的。

由于 C++语言提供了操作符重载，因此，string 类的赋值、连接以及比较操作就和 Pascal 中的 string 类型没有什么区别了。值得注意的是，在需要另外一个字符串的地方我们不但可以使用 string 类的对象，C 语言中保留下来的字符指针同样也可以被使用。另外，我们还可以使用 c_str()成员函数来得到一个内容和 string 类对象相同的字符指针。Pascal 中提供的字符串操作过程或函数在 string 类中也有相应的成员函数：

Pascal 中的过程或函数	作用	string 类中相对应的成员函数
Length	取得字符串的长度	length
Copy	取得字符串的一个子串	substr
Insert	插入一段字符串	insert
Delete	删除一段字符串	erase
Pos	查找一个子串出现的位置	find

string 类中的这些成员函数都提供了多个重载的版本，以便于灵活地使用。在这里，我们只介绍最常用的使用方法：

```
size_type length() const;
string substr(size_type pos = 0, size_type n = npos) const;
string& insert(size_type pos, const string& s);
string& erase(size_type pos = 0, size_type n = npos);
size_type find(const string& s) const;
```

在上面各成员函数的参数表中，pos 表示原字符串中的位置，n 表示一段子串的长度，而 s 则表示另外一个字符串。此外，如果 n 大于字符串中可用的长度，则子串将从 pos 开始，到原字符串结束为止。如果我们不指定 n 的话，默认参数 npos 将会把子串取到原字符串结束为止。

与 Pascal 不同的是，由于 0 在 string 类的字符串中是一个合法的位置，因此在查找子串时，如果子串没有在原字符串中出现的话，find 函数将返回 npos 而不是 0。另外，rfind 函数的作用也是查找子串。与 find 不同的是，rfind 找子串在原字符串中最后的一个出现位置，而 find 返回的是第一个出现位置。

当然，这些只是 string 类中一些最常用的成员函数最常见的用法。string 类的详细内容请参考有关资料。

2.4 流

在 C++中，输入和输出是通过流来实现的。C++标准库中流类的设计，充分体现了 C++语言的灵活性以及面向对象程序设计的优势。

在 C++中，所有输入流的基类都是 istream，所有输出流的基类都是 ostream。像标准输入流 cin、文件输入流、字符串输入流等都是由 istream 类派生出来的，因此，在可以使用 istream 类的地方，以上这些输入流都可以被使用，输出流也是同样。

对于流来说，最常用的操作就是输入和输出。在前面的例子中我们已经提到，对于一个输入流，我们可以使用提取符（>>）来进行输入，而对于一个输出流，我们可以使用插入符（<<）来进行输出。而且这些操作几乎可使被使用在所有语言内置的类型上。要注意的是，这些操作会按照类型自动进行数据的格式化处理。特别是在输入字符和字符串（包括字符指针和 string 类）时，提取符会先跳过所有的空白字符（空格、制表符、换行符等），然后再提取。对于字符串，当再一次遇到空白符时提取操作就告结束。

对于一些特殊的输入要求，我们可以使用非格式化输入来实现。常用的与非格式化输入有关的成员函数有以下这些：

```
int get();
istream& get(int& c);
istream& get(char* p, int n, char term = '\n');
istream& getline(char* p, int n, char term = '\n');
```

前两个函数的作用都是从流中读取一个字符（包括空白字符），注意在应该出现 char 的地方都是 int，这主要是因为如果读到流的结尾它们都要返回-1。后面两个函数的作用都是从流中读入多个字符，最多读 n 个，但如果遇到字符 term 读取会提前结束，两个函数都要保证 p 中有足够的空间。所不同的是，get()函数仍然把字符 term 留在流中，而 getline()函数则直接把字符 term 扔掉，这点区别对之后的输入会有一定的影响。

另外，成员函数 peek()返回流中下一个将被读取的字符，但是并不把这个字符真的读取出来，也就是说，仍然把这个字符留在流中。成员函数 unget()用来把最近读取的一个字符退回到流中。成员函数 eof()用来判断流是否到了结尾，就像 Pascal 中的 Eof 一样。不过有些时候也可以直接用流对象本身。在需要一个逻辑值的地方（比如 if 语句的括号里），C++会自动把流转换成一个逻辑值，表示流是否正常。如果已经读到流的结尾，这个逻辑值就会变成假。例如，我们可以用以下这个程序段来实现不断地从流中读取整数，直到流的结尾：

```
int i;
while(cin >> i) {
```

```
        // Do something
    }
```

流的成员函数中没有像 Pascal 中的 Eoln 那样的函数，不过判断行末可以用表达式 `cin.peek() == '\n'`来实现。

格式化输出

在 Pascal 中，我们在输出时可以指定场宽和小数部分的位数。C++的流同样也可以完成这些操作。指定场宽由成员函数 width()来完成，而指定小数部分的位数则稍微麻烦一些，要先把浮点数的输出方式设置为定点输出方式，然后再设置小数部分的位数。例如：

```
cout.setf(ios::fixed, ios::floatfield);
cout.precision(2);
cout << 1.2345 << endl;
```

以上程序段中第一个语句的作用就是把浮点数的输出方式设置为定点输出方式，第二个语句的作用是把小数部分的位数设置为 2。和 Pascal 一样，小数部分的最后一位也会进行四舍五入的处理。

需要注意的是，width()只对接下来的一个格式化输出有效，如果有多个输出需要指定场宽，那么就要写多个 width()函数。而 precision()则对之后所有的浮点数输出都有效。例如：

```
cout.width(3);
cout << 1 << 2;  // 1 的场宽为 3，而 2 采用实际宽度
cout.width(4);
cout << 2;
```

如果你觉得这样写太麻烦，你也可以把它写成：

```
cout << setw(3) << 1 << 2 << setw(4) << 2;
```

不过这时不要忘了在程序的最前面加上#include <iomanip>，因为 setw 是在 iomanip 中被定义的。

这样，我们就可以完成一般的格式化输出了。C++中流的格式化输出还有很多内容，如果有兴趣可以参考有关资料。

文件流和字符串流

C++把输入输出抽象为流的一大优点就是，我们不必关心到底我们操作的流是以什么形式存在的。也就是说，在输入时，不管流的来源是键盘，还是文件，或者是一个字符串中的内容，除了在流的创建时有些区别外，以后的操作都是一样的。

文件流

C++中，输入文件流的类名是 ifstream，输出文件流的类名是 ofstream，它们都被定义在头文件 fstream 中。

为了打开一个文件，我们既可以在流被创建时指定文件名，也可以在创建之后用 open()成员函数来指定文件名。例如：

```
ifstream fin("test.in");  // 创建时指定，马上可以被使用
ofstream fout;
fout.open("test.out");  // 创建之后指定
```

当一个文件流被打开后，我们就可以像使用 cin 或者 cout 那样来使用它们了。一般来说，我们不用特意地去写文件流的关闭语句，因为文件流类的析构函数会帮我们自动关闭文件。但如果想在析构之前关闭文件流，我们也可以使用 close()成员函数来实现。

字符串流

C++中字符串流的作用就是从一个字符串中读取数据，或者把数据输出到一个字符串中。输入字符串流的类名为 istringstream，输出字符串流的类名为 ostringstream，它们都被定义在头文件 sstream 中。

在创建输入字符串流时，我们需要指定一个字符串，作为数据的来源。在使用过程中，我们可以随时调用成员函数 str()来获得对应的字符串。

字符串流可以用来实现特定类型的数据和字符串之间的互相转换。例如：

```
string s("123");
istringstream sin(s);
int i;
sin >> i;
cout << i << endl;
ostringstream sout;
sout << i;
string s2 = sout.str();
cout << s2 << endl;
```

自定义类型的输入输出

我们已经知道，C++中的流对于系统内置的数据类型提供了提取符和插入符，极大的方便了输入输出操作。那么，对于我们自己定义的类型，是否也可以用提取符和插入符来进行输入输出呢？答案是肯定的，而且很简单：重载操作符>>和<<。下面我们来看一个简单的例子：

```
//: C02:Complex.cpp
// Input & Output for User-defined types
#include <iostream>
using namespace std;
class Complex {
public:
    Complex(double r = 0., double i = 0.)
      : re(r), im(i) {}
    double real() const { return re; }
    double imag() const { return im; }
private:
    double re, im;
```

```
    friend istream& operator >>(
        istream& is, Complex& c);
    friend ostream& operator <<(
        ostream& os, const Complex& c);
};
istream& operator >>(istream& is, Complex& c) {
    is >> ws;
    if(is.peek() == '(') {
        char chr;
        is >> chr;  // ignore '('
        is >> c.re;
        is >> chr;  // ignore ','
        is >> c.im;
        is >> chr;  // ignore ')'
    }
    else {
        is >> c.re;
        c.im = 0.;
    }
    return is;
}
ostream& operator <<(ostream& os, const Complex& c) {
    return os << '(' << c.re << ',' << c.im << ')';
}
int main() {
    Complex c;
    cin >> c;
    cout << c << endl;
} ///:~
```

上面这个程序实现了一个简单的复数类型，并对提取符和插入符进行了重载。由于我们不能修改流类的代码，因此，这两个操作符只能进行全局重载。返回类型是一个流的引用，这样使提取符或者插入符能够连续使用。参数列表中，第一参数必须是流的引用。在重载提取符是，第二个参数也必须是一个引用，因为在输入时，我们要能够改变变量的值；而重载插入符时，第二个参数一般是一个常引用，这样一可以保证运行速度（引用以指针的形式传递），二可以保证变量不会被修改。在函数结束时，不要忘了返回流本身，以确保操作符可以连续使用。

2.5 本章小结

也许大家已经发现在本文中的程序的开头和结尾有一些奇怪的注释[7]，它们有什么用处呢？原来，这些记号可以让我们有办法从本文的纯文本格式的文件[8]中提取各个程序的源代

[7] 这个点子来源于参考文献 2

[8] 获取纯文本文件的办法：用 MS Word 打开本文档，文件—另存为，保存类型—纯文本，保存，允许字

码。下面，我们就来编一个程序，实现这个功能：

```
//: C02:Extract.cpp
// Extracts codes from documents
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
using namespace std;
void error(const char* msg) {
    cerr << msg << endl;
    exit(1); // similar to Pascal's Halt
}
int main(int argc, char* argv[]) {
    if(argc <= 1)
        error("Document filename missing");
    ifstream fin(argv[1]);
    if(!fin)
        error("Cannot open file");
    string line;
    bool is_in_code = false;
    ofstream fout;
    while(getline(fin, line))
        if(line.find("//:") == 0) {
            if(is_in_code) {
                fout.close();
                error("Begin tag unexpected");
            }
            else {
                is_in_code = true;
                string name = line.substr(line.rfind(':') + 1);
                fout.open(name.c_str());
                if(!fout)
                    error("Cannot create file");
                fout << line << endl;
            }
        }
        else if(line.find("///:~") != string::npos) {
            if(is_in_code) {
                is_in_code = false;
                fout << line << endl;
                fout.close();
            }
            else
```

符替换（不然会出现一些乱码），确定。

```
                error("End tag unexpected");
        }
        else if(is_in_code)
            fout << line << endl;
    if(is_in_code) {
        fout.close();
        error("End tag expected");
    }
    fin.close();
} ///:~
```

在本程序中，我们演示了本章所介绍的 C++中的字符串和流。

该程序的使用方法很简单，编译后在命令行中输入程序名，并加上本文档的纯文本文件名作为参数，该程序就会自动把本文中源程序提取到当前目录。

在本章中，我们介绍了 C++中对数据进行抽象的机制——类，并进而介绍了建立在类的基础上的一些语言特性，这些都为我们的编程提供了有力的工具。

3 STL 简介

阅读本章的必要条件：了解 C++面向对象程序设计的基础知识、了解一定的算法知识

在本章中，我们将介绍标准 C++语言提供的功能强大的 STL（Standard Template Library，标准模板库）。

3.1 STL 概述

有时，我们会发现一些函数具有相同的功能，但由于参数类型不同，我们不得不编写多个类似的函数，很不方便。为了解决这个问题，C++提供了一般化编程的手段。所谓一般化编程，就是把类型也作为参数来处理。例如，我们经常使用的交换两个变量的值的函数：

```
void swap(int& x, int& y) {
    int t = x;
    x = y;
    y = t;
}
```

以上这个函数只能完成交换两个整型变量的值的功能，对于其他类型，我们不得不再写相似的函数。那么，我们如何用一般化编程来简化它呢？

模板函数

对于这个问题，在 C++中，我们可以使用模板函数来解决。在使用模板函数时，我们只需对原来的函数做一些较小的改动。例如，对于 swap 函数，我们可以这样把它写成模板函数：

```
template <class T> void swap(T& x, T& y) {
    T t = x;
    x = y;
    y = t;
}
```

在上面的 swap 函数中，template 就是模板函数的前缀，表示下面定义的是一个模板函数；<和>之间描述了模板参数，class T 表示 T 是一种类型（不一定是类）。在函数的定义中，我们就可以使用 T 来替代原来的 int。在定义了这个模板 swap 函数之后，我们就可以用它来交换任何两个类型相同的变量的值了，编译器会根据不同的类型生成不同版本的 swap 函数，大大简化了编程的复杂度。

一般地，编译器会根据调用时的实际参数的类型来特化模板函数。但我们也可以人为地指定特化时的参数类型。例如：

```
swap<int>(x, y);
```

这里，swap<int>()函数其实就和最初的那个交换两个整型变量的值的函数是一样的。

模板类

类似的，我们也可以对类进行一般化。例如：

```
template <class T, int max> struct c_array {
    typedef T value_type;
    typedef T& reference;
    typedef const T& const_reference;
    T v[max];
    operator T*() { return v; }
    reference operator [](size_t i) { return v[i]; }
    const_reference operator [](size_t i) const { return v[i]; }
    size_t size() const { return max; }
};
```

在上面的程序段中，我们定义了一个模板类 c_array，可以用来替代 C++语言内置的一维数组。模板参数有两个，第一个指定数组元素的类型，第二个指定数组元素的个数。如果我们要定义一个有 100 个元素的整型数组，就可以这样写：

```
c_array<int, 100> a;
```

由于我们重载了下标操作符和类型转换操作符，我们可以像使用普通的数组那样来使用这个类。

标准模板库

标准模板库 STL 就是建立在模板函数和模板类基础之上的功能强大的库。由于有了模板函数，我们就可以写出支持几乎所有类型（包括用户自定义的类型）的函数，能够实现一些一般化的常用算法（如统计、排序、查找等）；由于有了模板类，我们就可以写出支持几乎所有类型的容器，用来实现一些常用的数据结构（如链表、栈、队列、平衡二叉树等）。

有意思的是，与一般的库不同，组成 STL 的头文件中包含了所有函数和类的源代码。我们可以通过查阅这些头文件来学习 STL 是如何实现这些功能的。

下面，我们将陆续介绍 STL 的各个组成部分。

3.2 迭代器

迭代器实际上是一种一般化的指针类型，是对指针类型的抽象。根据所支持操作的不同，迭代器被分为五大类：输出迭代器、输入迭代器、前向迭代器、双向迭代器和随机迭代器。它们所支持的操作如下表所示：

迭代器类型	输出迭代器	输入迭代器	前向迭代器	双向迭代器	随机迭代器
缩写	Out	In	For	Bi	Ran
读取	不支持	`x = *p`	`x = *p`	`x = *p`	`x = *p`
操作	不支持	`p->x`	`p->x`	`p->x`	`p->x p[i]`
写入	`*p = x`	不支持	`*p = x`	`*p = x`	`*p = x`
迭代	`++`	`++`	`++`	`++, --`	`++ -- + - += -=`
比较	不支持	`== !=`	`== !=`	`== !=`	`== != < >`

					<= >=

从上表中我们可以看出，指针类型其实就是一种特殊的随机迭代器类型。对于一般的迭代器，这些功能都是通过操作符重载来实现的。

此外，前向迭代器兼容输出迭代器和输入迭代器，双向迭代器兼容前向迭代器，而随机迭代器则兼容双向迭代器，并且兼容关系具有传递性。

迭代器的基本类型在头文件<iterator>中被定义。由于它是一个抽象的类，因此它的定义看上去很奇怪：

```
template <class Cat, class T, class Dist = ptrdiff_t,
    class Ptr = T*, class Ref = T&>
struct iterator {
    typedef Cat iterator_category;
    typedef T value_type;
    typedef Dist difference_type;
    typedef Ptr pointer;
    typedef Ref reference;
};
```

这里，iterator 类只为迭代器类型提供了一个接口，具体的实现将留在需要的地方进行。

迭代器不但可以用来访问元素，在整个 STL 中，它还起到了纽带的作用，把算法和容器紧紧地联系在了一起。这个作用我们会在接下来的几节中看到。

另外，头文件<utility>中还定义了一个比较常用的模板类 pair，用来表示一个二元组：

```
template <class T1, class T2> struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair() : first(T1()), second(T2()) { }
    pair(const T1& x, const T2& y) : first(x), second(y) { }
    template<class U, class V> pair(const pair<U, V>& p)
        : first(p.first), second(p.second) { }
};
template <class T1, class T2>
pair<T1, T2> make_pair(const T1& t1, const T2& t2) {
    return pair<T1, T2>(t1, t2);
}
```

这个类将会在描述算法和容器时多次被使用。

3.3 算法

STL 中的算法提供了大量基本的操作，其中大部分都是对序列的操作。在介绍 STL 算法（以下简称算法）之前，让我们来了解一些与算法有关的基础知识。

序列

几乎所有的算法的操作对象都是序列。在 STL 中，描述一个序列需要两个迭代器，第一个迭代器指向序列中的第一个元素，而第二个迭代器指向序列中最后一个元素的后一个位置。也就是说，由这两个迭代器作为端点所代表的半闭半开区间就是整个序列。

对于这样的定义，一开始我们可能会觉得奇怪。但这样做的确能带来很多便利。例如，对于一个序列[first, last)，表达式 last - first 就表示这个序列的长度（如果迭代器支持减法运算）；对于一个有 n 个元素的一维数组 s，我们可以用[s, s + n)来描述代表整个数组的序列；在描述插入位置时，我们可以用迭代器 p 来表示插入到 p 所指向的元素之前，这样，first 就表示插入到序列最前端，而 last 就表示插入到序列最末端，中间的位置也都有迭代器与之一一对应。

对于下一节中将要介绍的 STL 容器，我们可以使用容器类所提供的成员函数来产生迭代器，进而产生包含整个容器中的所有或者部分元素的序列。

函数对象

有一些算法的参数中，除了要描述序列之外，还要求提供函数对象。例如：

```
template<class In, class Op>
    Op for_each(In first, In last, Op f);
```

这个函数的作用就是对[first, last)中的每一个元素 x，执行一次 f(x)。我们当然可以把函数名作为参数 f，但对于某些操作（比如求和），我们可能要用到一些全局的变量。如果我们把这些变量定义成全局变量，就破坏了封装的原则。那么，我们该怎么办呢？

我们可以使用函数对象来解决这个问题。例如，对于求和操作，我们可以编写这样一个类：

```
template<class T> class Sum {
    T res;
public:
    Sum(T i = T()) : res(i) { }
    void operator ()(const T& x) { res += x; }
    T result() const { return res; }
};
```

这时，如果我们就可以这样来对一个序列进行求和操作：

```
Sum<int> sum;
for_each(s, s + n, sum(0));
cout << sum.result() << endl;
```

实际上 STL 已经提供了求和的算法，这里用这种方法求和只是用来演示函数对象的用法。

返回类型为 bool 的函数对象（包括函数）被称为谓词，常用来判断某些关系是否成立。

另外，在头文件<functional>中，STL 提供了一些常用运算的函数对象（都是模板类），如下表所示：

类名	类型	作用
equal_to	双目	arg1 == arg2
not_equal_to	双目	arg1 != arg2

greater	双目	arg1 > arg2
less	双目	arg1 < arg2
greater_equal	双目	arg1 >= arg2
less_equal	双目	arg1 <= arg2
logical_and	双目	arg1 && arg2
logical_or	双目	arg1 || arg2
logical_not	单目	!arg
plus	双目	arg1 + arg2
minus	双目	arg1 - arg2
multiplies	双目	arg1 * arg2
divides	双目	arg1 / arg2
modulus	双目	arg1 % arg2
negate	单目	-arg

算法概述

STL 提供了 60 多个算法，它们都是模板函数，一般都有两个重载版本。一个版本采用默认的操作符，一般用等于或者小于（与算法种类有关）；另一个版本则可以人为指定函数对象，用来替代默认的操作符。

下表列举了 STL 所提供的算法，除了最后四个定义在头文件<numeric>中以外，其余都定义在<algorithm>中：

算法名称	作用
for_each()	对一个序列中的每个元素进行一项操作
find()	在一个序列中查找某个值第一次出现的位置
find_if()	在一个序列中查找第一个符合某个谓词的元素的位置
find_first_of()	在一个序列中查找另一个序列中的任意元素第一次出现的位置
adjacent_find()	在一个序列中查找第一对相邻且相等的元素的位置
count()	统计某个值在一个序列中出现的次数
count_if()	统计一个序列中符合某个谓词的元素的个数
mismatch()	查找两个序列中第一对相异元素的位置
equal()	判断两个序列是否完全相等
search()	在一个序列中查找某个子序列第一次出现的位置
find_end()	在一个序列中查找某个子序列最后一次出现的位置
search_n()	在一个序列中查找某个值连续出现 n 次的位置
transform()	对序列中的每个元素进行一项操作，并产生相应的输出序列
copy()	复制一个序列中的元素到另一个序列（从前往后）
copy_backward()	复制一个序列中的元素到另一个序列（从后往前）
swap()	交换两个元素的值
iter_swap()	交换两个迭代器所指向的元素的值
swap_ranges()	交换两个序列中的元素
replace()	把一个序列中的某个值替换成另一个值
replace_if()	把一个序列中符合某个谓词的元素替换成另一个值
replace_copy()	同 replace()，但结果输出到另一个序列
replace_copy_if()	同 replace_if()，但结果输出到另一个序列
fill()	把一个序列中的所有元素都填充为某个值
fill_n()	同 fill()，但用起始迭代器和序列长度来描述一个序列
generate()	把一个序列中的所有元素都填充为某个发生器的返回值

generate_n()	同 generate()，但用起始迭代器和序列长度来描述一个序列
remove()	删除一个序列中等于某个值的元素
remove_if()	删除一个序列中符合某个谓词的元素
remove_copy()	同 remove()，但结果输出到另一个序列
remove_copy_if()	同 remove_if()，但结果输出到另一个序列
unique()	删除一个序列中多余的相邻相等元素
unique_copy()	同 unique()，但结果输出到另一个序列
reverse()	反转一个序列
reverse_copy()	同 reverse()，但结果输出到另一个序列
rotate()	旋转一个序列
rotate_copy()	同 rotate()，同结果输出到另一个序列
random_shuffle()	随机打乱一个序列中各元素的位置
sort()	对一个序列中的元素进行排序
stable_sort()	同 sort()，但保证排序的稳定性[9]
partial_sort()	对一个序列的前面部分元素进行排序
parital_sort_copy()	同 partial_sort()，但结果输出到另一个序列
nth_element()	把一个序列中第 n 小的元素放到它排序后的位置上
lower_bound()	在一个有序序列中二分查找某个值，返回第一次出现的位置
upper_bound()	在一个有序序列中二分查找某个值，返回最后一次出现的位置的后一个位置
equal_range()	在一个有序序列中二分查找某个值，返回所有与这个值相等的元素构成的子序列
binary_search()	在一个有序序列中二分查找某个值，返回这个值是否出现在这个序列中
merge()	合并两个有序序列
inplace_merge()	合并两个连续的有序序列
partition()	重新调整一个序列中所有元素的位置，使得符合某个谓词的元素被排在前面
stable_partition()	同 partition()，但保证调整的稳定性
includes()	判断一个序列是否为另一个序列的子序列（可以不连续）
set_union()	对两个有序序列进行集合并运算
set_intersection()	对两个有序序列进行集合交运算
set_difference()	对两个有序序列进行集合差运算
set_symmetric_difference()	对两个有序序列进行集合对称差运算
make_heap()	把一个序列调整为一个堆
push_heap()	向堆中插入一个元素
pop_heap()	从堆中提取堆顶的元素
sort_heap()	把一个堆调整为一个有序序列
min()	返回两个值中较小的值
max()	返回两个值中较大的值
min_element()	返回一个序列中最小的元素的位置
max_element()	返回一个序列中最大的元素的位置
lexicographical_compare()	按照字典次序比较两个序列
next_permutation()	返回字典次序中由一个序列中所有元素构成的下一个排列序列
prev_permutation()	返回字典次序中由一个序列中所有元素构成的上一个排列序列
accumulate()	计算一个序列中所有元素的和
inner_product()	计算两个序列的内积

[9] 所谓稳定性，就是指在排序后的序列中，具有相等次序的元素之间的相对位置与原序列中的保持一致。

partial_sum()	计算一个序列的部分和序列
adjacent_difference()	计算一个序列的相邻元素之差序列

上表只是对 STL 中的算法作了简单的功能介绍，下面我们来详细介绍几个常用的算法。

在信息学竞赛中，我们使用频率最高的算法是排序。STL 中的 sort()算法使得排序变得十分方便：

```
template<class Ran> void sort(Ran first, Ran last);
template<class Ran, class Cmp>
   void sort(Ran first, Ran last,  Cmp cmp);
```

对于已经定义小于操作符的类型，我们只需简单地写 sort(s, s + n)就可以完成排序。如果没有定义小于操作符，或者要用其他关系操作符进行排序，我们就要使用第二个重载版本。这时，除了给出要排序的序列外，我们还要给出一个谓词，作为排序的依据。例如，sort(s, s + n, greater<int>())就可以实现从大到小排序。

sort()算法一般用快速排序来实现，算法的平均时间复杂度为 $O(n \log n)$。但最坏时间复杂度为 $O(n^2)$，不过出现这种情况的概率很小。

另外，二分查找在信息学竞赛中也是相当常用的算法。STL 中与二分查找有关的算法有 4 个：

```
template<class For, class T> bool
    binary_search(For first, For last, const T& val);
template<class For, class T, class Cmp> bool
    binary_search(For first, For last, const T& val, Cmp cmp);
template<class For, class T> For
    lower_bound(For first, For last, const T& val);
template<class For, class T, class Cmp> For
    lower_bound(For first, For last, const T& val, Cmp cmp);
template<class For, class T> For
    upper_bound(For first, For last, const T& val);
template<class For, class T, class Cmp> For
    upper_bound(For first, For last, const T& val, Cmp cmp);
template<class For, class T> pair<For, For>
    equal_range(For first, For last, const T& val);
template<class For, class T, class Cmp> pair<For, For>
    equal_range(For first, For last, const T& val, Cmp cmp);
```

虽然从名字上来看，二分查找的算法应该是 binary_search()，但实际上这个算法的用处并不大，因为它只返回某个值是否在序列中。真正有用的二分查找算法是后面三个。

如果被查找的值出现在序列中，那么 lower_bound()将返回序列中第一个和被查找的值相等的元素的位置，而 upper_bound()将返回序列中最后一个和被查找的值相等的元素的后一个位置。这样的定义和 STL 中序列的定义是一致的。设 lower_bound()的返回值为 l，upper_bound()的返回值为 u，那么序列[l, u)就是原序列中所有和被查找的值相等的元素所构成的子序列。如果被查找的值没有出现在序列中，那么 lower_bound()和 upper_bound()的返回值是相等的，都指向一个位置，使得如果把被查找的值插入到这个位置，序列仍然能够保持有序性。而 equal_range()则是 lower_bound()和 upper_bound()的综合版本。

对于其他算法的详细使用说明，请参考有关资料。

下面，我们来看一道例题：

最长单调递增子序列

题目描述

给定一个长度为 n 的整数序列 $A=\{a_1,a_2,\ldots,a_n\}$，求一个最大的整数 m，使得存在另一个序列 $P=\{p_1,p_2,\ldots,p_n\}$，满足 $1\le p_1<p_2<\cdots<p_n\le n$，且 $a_{p_1}<a_{p_2}<\cdots<a_{p_n}$。

输入格式（文件名：LMIS.in）

第 1 行：一个正整数 $n(n\le 30,000)$，表示原序列的长度。

第 2 行：n 个正整数 $a_1,a_2,\ldots,a_n(a_i\le 1,000,000,000)$，表示整个序列。

输出格式（文件名：LMIS.out）

只有一行，包含一个整数 m，表示最长单调递增序列的长度。

输入样例

10

3 1 4 1 5 9 2 6 5 3

输出样例

4

解：这是一道很典型的动态规划题目。设 f_i 表示结尾元素为原序列中第 i 个元素的最长单调递增序列的长度（为了简便，设 $a_0=-\infty$，$f_0=0$），动态规划的状态转移方程如下：

$$f_i=\max_{0\le j<i\wedge a_j<a_i}\{f_j+1\}$$

最后所要求的结果就是{f_i}中的最大值。

实现这个算法的程序如下（由于 C++中数组元素下标从 0 开始，因此算法稍有改动）：

```
//: C03:LMIS1.cpp
// Longest Monotonically Increasing Subsequence
// Original Algorithm O(n^2)
#include <fstream>
#include <algorithm>
using namespace std;
ifstream fin("LMIS.in");
ofstream fout("LMIS.out");
const int max_n = 30000;
int n;
int a[max_n];
int f[max_n];
int main() {
    fin >> n;
    for(int i = 0; i < n; i++)
        fin >> a[i];
    for(int i = 0; i < n; i++) {
        f[i] = 1;
        for(int j = 0; j < i; j++)
            if(a[j] < a[i])
                f[i] = max(f[i], f[j] + 1);
```

```
    }
    fout << *max_element(f, f + n) << endl;
} ///:~
```

这个算法的时间复杂度为 $O(n^2)$，不能满足题目所给数据范围的要求。那么，怎样才能改进这个算法呢？

为了改进这个算法，我们需要引入一个辅助数组。设 g_i 表示到目前为止，所有长度为 i 的单调递增子序列中最后一个元素的最小值。易知，$g_{i-1}\le g_i(1\le i\le n)$（可以用反证法来证明），也就是说，$\{g_i\}$是一个不下降序列。当到第 i-1 个字符为止的$\{g_i\}$已知时，f_i 就等于在$\{g_i\}$中第一个大于或等于 a_i 的元素的位置（也就是 lower_bound()所给出的位置）。然后，我们要对$\{g_i\}$进行更新。更新十分简单，只要令 $g_{f_i}=a_i$ 就可以了。一开始，令 $g_i=\infty(1\le i\le n)$。由于每次查找位置的时间复杂度为 $O(\log n)$，而更新的时间复杂度为 $O(1)$，因此，这个算法总的时间复杂度就是 $O(n\log n)$，符合题目要求。

实现这个算法的程序如下（由于 C++中数组元素下标从 0 开始，因此算法稍有改动）：

```
//: C03:LMIS2.cpp
// Longest Monotonically Increasing Subsequence
// Improved Algorithm O(n log n)
#include <fstream>
#include <algorithm>
using namespace std;
ifstream fin("LMIS.in");
ofstream fout("LMIS.out");
const int infinity = 2000000000;
const int max_n = 30000;
int n;
int a[max_n];
int f[max_n];
int g[max_n];
int main() {
    fin >> n;
    for(int i = 0; i < n; i++)
        fin >> a[i];
    fill(g, g + n, infinity);
    for(int i = 0; i < n; i++) {
        int j = lower_bound(g, g + n, a[i]) - g;
        f[i] = j + 1;
        g[j] = a[i];
    }
    fout << *max_element(f, f + n) << endl;
} ///:~
```

通过比较两个程序可以看出，由于 STL 的使用，改进后的程序反而比原来的更为简单，很好地体现出了 STL 能够降低编程复杂度的优势。

3.4 容器

我们知道，算法只有作用在一定的数据结构上才能发挥它的功能。STL 除了给我们提供了大量的算法之外，还给我们提供了一些常用的数据结构。由于这些数据结构都是用来存放一组同类的对象的，因此它们被称为容器。下表列出了常用的 STL 所提供的容器：

名称	描述	类型	所在头文件	迭代器类型
vector	向量	一般容器	<vector>	随机迭代器
deque	双头队列	一般容器	<deque>	随机迭代器
list	链表	一般容器	<list>	双向迭代器
stack	栈	容器适配器[10]	<stack>	不提供迭代器
queue	队列	容器适配器	<queue>	不提供迭代器
priority_queue	优先队列	容器适配器	<queue>	不提供迭代器
set	集合	关联容器[11]	<set>	双向迭代器
multiset	多重集合	关联容器	<set>	双向迭代器
map	映射	关联容器	<map>	双向迭代器
multimap	多重映射	关联容器	<map>	双向迭代器

所有的容器都是用模板类来实现的，使用时要指定元素的类型。这样可能会使类型名称显得比较长，写起来不方便。必要时可以使用 typedef 来进行简化。例如：

```
typedef vector<string> str_vec;
```

STL 容器能够自动为元素分配空间。比如 vector 会先保留一部分的空间，当插入过多的元素时，vector 容器会自动重新分配一块更大的空间，并把原来的元素复制过去。在容器对象析构时，元素所占用的空间也会被自动地回收。

对于这些容器，STL 提供了基本相同的成员类型和函数接口（当然，不同的容器成员函数略有不同）：

名称	类型	简要描述
value_type	成员类型	元素类型
iterator	成员类型	迭代器类型，类似于 value_type*
const_iterator	成员类型	常迭代器类型，类似于 const value_type*
reverse_iterator	成员类型	反转迭代器类型，类似于 value_type*
const_reverse_iterator	成员类型	常反转迭代器类型，类似于 const value_type*
reference	成员类型	元素引用类型，等价于 value_type&
const_reference	成员类型	元素常引用类型，等价于 const value_type&
key_type	成员类型	关键字类型（只用于关联容器）
mapped_type	成员类型	被映射的类型（只用于关联容器）
begin()	迭代器类操作	返回指向第一个元素的迭代器
end()	迭代器类操作	返回指向最后一个元素的后一个位置的迭代器
rbegin()	迭代器类操作	返回指向最后一个元素的反转迭代器
rend()	迭代器类操作	返回指向第一个元素的前一个位置的反转迭代器
front()	元素操作	第一个元素
back()	元素操作	最后一个元素
operator []()	元素操作	下标操作符
push_back()	栈和队列操作	向末尾添加一个元素
pop_back()	栈和队列操作	删除最后一个元素

[10] 容器适配器只提供一个函数接口，不提供迭代器，同时包含有一个一般容器，元素的存取由这个一般容器来完成。

[11] 关联容器中的元素一般由关键字和被映射的值两部分组成，可以通过关键字快速查找元素。

push_front()	栈和队列操作	向开头添加一个元素（vector 不支持）
pop_front()	栈和队列操作	删除第一个元素（vector 不支持）
insert()	列表操作	插入一些元素
erase()	列表操作	删除一些元素
clear()	列表操作	清空容器
size()	其他操作	返回元素个数
empty()	其他操作	判断容器是否为空
resize()	其他操作	调整容器中元素的个数
find()	关联容器类操作	查找某个关键字
lower_bound()	关联容器类操作	查找某个关键字，返回第一个符合条件的位置
upper_bound()	关联容器类操作	查找某个关键字，返回最后一个符合条件的后一个位置
equal_range()	关联容器类操作	查找某个关键字，返回符合条件的区间

下面，我们来详细地介绍一些常用容器。

vector 就是一个智能型的一维数组，能够自动调整存储空间。vector 还优化了vector<bool>的存储方式，用每一位来存储一个 bool。但是由于信息学竞赛中数据的规模一般是给定的，再加上 vector 在调整存储空间时移动数据会花费较多的时间，因此在信息学竞赛中 vector 并不是很实用。

deque 是用指针数组来实现的，两头都留有一定的空间，因此首尾的插入/删除操作很快。deque 一般用作容器适配器（stack、queue 等）的内置容器。

list 一般用双向链表来实现，可以使我们避免书写大量重复的链表基本操作，降低编程复杂度。

stack、queue、priority_queue 都是容器适配器。由于提供的接口函数较少，因此 stack 和 queue 的用处有所限制，但应付一般的情况都已经够用了。priority_queue 是一个用二叉堆实现的优先队列，与一般队列的区别是在提取元素时，优先队列总是先提取最小的元素。priority_queue 在做模拟题时非常有用，因为做模拟题的关键就是要处理好事件的先后次序。我们可以把题目中的各种事件都作为对象，并按照题目中的规则赋予它们不同的优先级，重载比较操作符。然后，我们把这些对象一一推入优先队列，再提取出来并一一加以处理就可以了。另外，由于接口函数的限制，优先队列很难处理队列中的元素需要改变的情况（例如 Dijkstra 算法、Prim 算法等）。

STL 提供的关联容器功能十分强大。这四种关联容器都是用平衡二叉树（一般是红黑树）来实现的。容器内的元素都是有序的，插入、删除、查找等操作都可以在 $O(\log n)$的时间内完成。set、multiset 中元素本身直接参与排序，或者说元素本身就是关键字；而 map、multimap 中只有关键字参与排序，被映射的值只作为周围数据存在，并且可以被修改。另外，set、map 中不能出现相等的关键字，而 multiset、multimap 则允许出现相等的关键字。

这些关联容器都可以通过关键字来查找元素。进行查找操作的成员函数有 find()、lower_bound()、upper_bound()、equal_range()等。在没有找到要求查找的关键字时，find()返回成员函数 end()的返回值以表示失败，而其他三个函数的返回值则与<algorithm>中的同名函数相似。此外，由于 map 中的关键字和被映射的值是一一对应的，因此 map 类重载了下标运算符，使得元素的访问变得更加方便。例如：

```
map<string, int> week;
week["Sunday"] = 0;
week["Monday"] = 1;
week["Tuesday"] = 2;
week["Wednesday"] = 3;
week["Thursday"] = 4;
```

```
    week["Friday"] = 5;
    week["Saturday"] = 6;
    cout << week["Thursday"] << endl;
```

在使用 map 的下标操作符时，如果方括号内的关键字所对应的元素存在的话，这个元素中被映射的值的引用将被返回；如果这个关键字所对应的元素不存在，那么一个新的具有这个关键字的元素将会被插入，被映射的值被初始化为该类型的默认值，同时返回被映射的值的引用。

下面，我们来看一道例题：

支付帐单

题目描述

比尔最近遇到了一件麻烦事。每天上午，他会收到若干张帐单（也可能一张也没收到），每一张都有一定的面额。下午，他会从目前还没有支付的帐单中选出面额最大和最小的两张，并把它们付清。还没有支付的帐单会被保留到下一天。现在比尔已经知道他每天收到帐单的数量和面额，请你帮他给出支付的顺序。

输入格式

第 1 行：一个正整数 N(N≤30,000)，表示总共的天数。

第 2 行到第 N+1 行：每一行描述一天中收到的帐单。先是一个非负整数 M，表示当天收到的账单数，后跟 M 个正整数（都小于 1,000,000,000），表示每张帐单的面额。

输入数据保证每天都可以支付两张帐单，并且帐单会在最后一天全部付清。

输出格式

输出共 N 行，每行两个用空格分隔的整数，分别表示当天支付的面额最小和最大的支票的面额。

输入样例

```
4
3 3 6 5
2 8 2
3 7 1 7
0
```

输出样例

```
3 6
2 8
1 7
5 7
```

解：从题目给出的数据范围可以看出，可以算法的时间复杂度要比 $O(n^2)$低，$O(n \log n)$是可以接受的。$O(n \log n)$的算法也有很多，比如，我们可以建立两个堆，一个最大堆，一个最小堆，并在相应元素之间建立一个映射。收到账单时，就把面额推入两个堆中，调整位置；支付帐单时，分别从两个堆中取出堆顶元素，并进行调整。由于元素在入堆后位置需要进行调整，因此很难用 STL 中的优先队列来实现，如果采用这种算法，我们必须自己写堆的代码。

另外一种 $O(n \log n)$的算法就是用平衡二叉树，但如果自己写平衡二叉树的算法，实现起来比上面一种更繁。但由于 STL 中的集合是用红黑树来实现的，借助它，我们的程序可以变得十分简单（注意到帐单的面额可能会相同，我们必须使用 multiset）：

```
//: C03:Bills.cpp
// Pay 2 Bills Every Day
```

```
#include <iostream>
#include <set>
using namespace std;
int main() {
    int n;
    cin >> n;
    multiset<int> bills;
    while(n--) {
        int m;
        cin >> m;
        while(m--) {
            int a;
            cin >> a;
            bills.insert(a);
        }
        cout << *bills.begin() << ' ' << *--bills.end() << endl;
        bills.erase(bills.begin());
        bills.erase(--bills.end());
    }
} ///:~
```

上面一个例子又一次体现了 STL 的强大功能，大大降低了编程复杂度。

除了上面讲到的 STL 容器外，标准 C++还提供了位集合 bitset（类似于 Pascal 中的 set）、算术数组 valarray（能够对一组相同类型的数据进行运算，并可以用来实现矩阵）。现在很多 C++编译器所使用的由 SGI 实现的 STL[12]中，还提供了 slist（单链表，可以节约空间）、hash_set、hash_multiset、hash_map、hash_multimap（用哈希表实现的集合和映射）、rope（一种字符串数据结构，支持对数时间复杂度的插入和删除操作）等。这些容器都很实用，有兴趣可以参考有关资料。

3.5 本章小结

在前面三节中，我们分别讲了 STL 的三个重要组成部分：迭代器、算法和容器。但是，这三部分并不是相互独立的，只有把它们有机地结合起来才能使 STL 发挥应有的作用。在融会贯通的过程中，我们要注意一些问题。

首先，要优先使用容器本身所带的算法。有些容器类的成员函数中提供了一些功能和通用算法类似的函数，要优先使用这些成员函数。因为通用算法是为一般的序列设计的，要照顾到一般性就不可能做太多的优化；而容器类的成员函数则可以根据容器本身的特点对算法进行优化，或者完成通用算法所不能完成的功能。例如，list 容器类的成员函数中有 remove()、unique()、merge()、reverse()、sort()等，要优先使用，而且通用算法中的 sort()无法对 list 中的元素进行排序（因为 list 只能提供双向迭代器，而通用算法 sort()需要随机迭代器）。

其次，在容器和算法结合使用时要注意输出序列的问题。很多算法都要求指定输出序列，一般用一个指向输出序列开头的迭代器来表示的。在算法中会不断地对这个迭代器进

[12] 请访问http://www.sgi.com/tech/stl。

行增 1 运算，以输出整个序列。由于普通的迭代器是不进行越界检查的，因此除非能够保证容器中已经有足够多的元素来存储输出序列，否则就不要直接用容器中的迭代器来指定输出序列。正确的做法是用<iterator>中的插入迭代器。

插入迭代器有三种，分别为前插入迭代器、后插入迭代器和一般插入迭代器。对于一个容器 c，这三种迭代器可以分别用函数 front_inserter(c)、back_inserter(c)和 inserter(c, i)来产生，其中 i 是容器 c 的一个迭代器。它们都是输出迭代器，会在增 1 操作时进行相应的插入操作（push_front()、push_back()和 insert()）。例如：

```
unique_copy(v.begin(), v.end(), back_inserter(v2));
set_union(s1.begin(), s1.end(), s2.begin(), s2.end(),
   inserter(s, s.begin()));
```

上面第一个语句把向量 v 中相邻重复的元素删除，并把结果添加到向量 v2 的末尾；而第二个语句则对集合 s1 和 s2 进行并运算，并把结果添加到集合 s 中（这里 s.begin()只是用来凑参数的个数，元素在 set 中仍然按照有序排列，与插入位置无关）。

通过本章的学习，我们对 C++中的 STL 有了一定的了解。STL 的优点很多，能够降低编程复杂度，提高编程的正确率。但是，有利必有弊，STL 也不例外。由于 STL 采用了一般化编程的机制（模板类、模板函数等），因此只要程序稍有不对就可能在编译时得到很多奇怪的错误信息，而且错误的地方往往是在 STL 的头文件中（因为模板的容错性很大，所以错误往往要在特化时才能发现）。又由于 STL 采用了类的封装，因此也给我们的动态调试带来了很多不便(因为我们很难查看容器内部的元素)。下面是对在信息学竞赛中使用 STL 的几点建议：

1、多使用 STL 的算法，降低编程复杂度，提高正确率。

2、在同等条件下优先使用 C++内置数组，而不是 STL 的容器，便于调试查看；

3、尽量多用静态查错，少用动态查错；

4、动态查错时用向屏幕输出的方式来查看调试器不能查看的内容；

总结

程序设计语言的学习过程与自然语言的学习有着十分相似的地方。

本文第一章讲了从 Pascal 语言到 C++语言的转变过程。在学习的过程中，我们要充分利用它们之间的相同之处，明确不同之处，这样才能加快转变的速度。这就好像在掌握了母语的基础上去学习一门外语，利用好语言之间的相同之处能够使外语的学习变得简单；而如果不能解决好语言之间的不同之处，就会给外语的学习带来困难。要学好外语需要多加练习，而只有不断地使用一门程序设计语言写程序才能切实提高对程序设计语言的掌握水平。

第二章和第三章进一步深入，讲了一些 C++中较为高级的内容。这就好比外语中一些独特的文化底蕴，我们只有了解这些文化底蕴，才能真正掌握一门外语。程序设计语言也是如此，第二章和第三章中所讲的内容使我们能够更好地使用 C++语言，特别是第三章中所讲的 STL，大大降低了编程复杂度，提高了代码的正确率，对信息学竞赛有着很大的帮助。

而信息学竞赛中的程序设计又与写作十分相似。写作的基础是掌握好一门自然语言，程序设计的基础则是掌握好一门程序设计语言。但是，要写出优秀的文章仅仅靠掌握好一门自然语言是不够的，我们还需要有精巧的构思和华丽的词藻；信息学竞赛中的程序设计亦是如此，掌握好一门程序设计语言十分重要，但程序设计的核心是算法的设计。只有把

两者有机地结合起来，才能在信息学竞赛中过关斩将，取得佳绩。

参考文献

1、《C++程序设计语言（特别版 影印版）》
2、《C++编程思想（英文版 第 2 版）》
3、《算法导论（第二版 影印版）》