程序设计基础

Programming Fundamentals

计学组

西安交通大学

XI'AN JIAOTONG UNIVERSITY

作品信息

- **标题**：程序设计基础 : *Programming Fundamentals*
- **作者**：许孜诚 魏佳哲 李垦 林圣翔
- **校对排版**：成崔皓 武靖博 傅明泽 曹雨晗
- **出品时间**：2023 年 12 月 12 日
- **总页数**：44

目录

前言

欢迎大家来到计算机程序设计系列课程的学习！

对于在之前的学习生涯中从未接触过编程的同学来说，这门课程的学习曲线可能会相对陡峭。程序设计是学习计算机理论的基础，之后学年的课程将会与编程有着千丝万缕的联系，因此大家在学习过程中要尽量做好预习，配合讲义补充漏洞。更关键的是，注意提高自身的代码能力。程序设计是一门应用性极强的课程，只是纸上谈兵，理论侃侃而谈是远远不够的，在学习过程必须注意大量编程，这样才能切实提高编码能力，一方面在测试中取得优异成绩，另一方面为未来的工作/研究打下坚实的代码基础。

养成良好的提问习惯不仅对这门课的学习大有裨益，对之后的团队项目开发，实验室科研协作经历都会产生意想不到的帮助。知名企业高管教练粟津恭一郎先生曾著有《学会提问》一书。粟津恭一郎每天的工作内容就是对大企业高管不断地“提问”，用提问引导企业管理者获得更大的成功。十多年下来，粟津恭一郎成为了日本“提问力”专家，《学会提问》这本书也得到了很多人的认可。在这本书中，作者淋漓尽致地剖析了如何才能问出好的问题，怎样提问才能更快的得到想要的答案，并使自己可以从提问中收获能力的提高。

在学习过程中，“提问”是重要的一环。通过提问，很多你也许绞尽脑汁一天也不得其要的问题马上就能得到解决，而你的知识体系与技术能力也在一次次提问与解答的过程中与日俱长。然而，并不是所有提问都能有所收获，也并不是所有提问都能得到想要的答案，这一点在程序设计以及未来的技术开发中表现的尤为明显。由于计算机技术本身的繁琐性与碎片化的知识分布，初学者在学习过程中必然会遇到大量问题。简明扼要，信息详备的提问可以令解答者瞬间明白你的困境，并且做出指导；而模糊不清，似是而非的提问不仅很难得到答案，还会使学习曲线更加陡峭，破坏提问与解答者的心情。因此，在程序设计课程开始之前，我们有必要好好学一学在学习程设的过程中，如何提出你的问题，以得到最佳答案。

注意编译器的报错信息

编译器永远是最忠实的解答者。如果代码发生编译错误，可以先不着急提问，而是先阅读编译器提供的报错信息。如果自己发现不了代码的问题，可以将报错信息复制到搜索引擎中查询此类错误。

通过云剪切板向他人提供代码

如果还是解决不了，可以通过 paste.ubuntu.com，洛谷云剪切板 等方式将代码提供给他人请求 debug 帮助。

描述好你的目标与你的问题

当代码出现问题时，而且不是编译错误，最好不要直接问应该如何处理，同时也要说清你的代码的目标是什么。否则解答者就还需要依据你的代码揣测这段代码的目的是什么，然后才能下手修改。让我们再看一个例子：

如果是这样的提问：

- 我的代码就是无法打印题目要求的内容，怎么办呢？

这将是一个非常糟糕的问题，因为解答者根本不知道你想用代码干什么。而如果是这样：

- 我想筛出 1 - 1000 内的所有质数并五个一行输出，可是每 15 行我发现会多一行缩进，这是为什么呢？

那么解答者马上可以定位到代码的相应位置并检查。

第一章　数据类型

1.1　原始数据类型一览

原始数据类型是内置或预定义的数据类型，用户可以直接使用它们声明变量：

- 整型/浮点型：存放的信息为数值。
- 字符型：存放的信息为单个字符，例如 `'c'` `'@'` `')'`。
- 布尔型：存放的信息为 `true` 或 `false`（1 或 0）。

1.2　整型

C/C++ 中的整形数据主要有 `short int`，`int`，`long int`，`long long int`。其中 `short int` 可简写为 `short`，`long int` 可简写为 `long`，`long long int` 可简写为 `long long`。

各类型大小：除了 `char` 型在 C 标准中明确规定占一个字节之外，其它整型占几个字节属于 Implementation Defined；通常的编译器实现遵守 ILP32 或 LP64 规范[1]，其中，`int` 占 4 字节，`short` 占 2 字节，`long long` 占 8 字节。

不同的数据类型规定了不同的机器数长度，决定了对应数据的数值范围，当一个整数超出此范围时计算机会将其转换为在数值范围内所允许的一个数，称为整型数据的溢出处理。一般地，超过最大值的有符号整型数值会向上溢出变成负数，超过最小值的数据会向下溢出变成正数。我们以 `short` 型数据的溢出为例：

```
  0111111111111111  ……   32767
+ 0000000000000001  ……  +     1
  1000000000000000  ……   -32768（补码）
```

向上溢出

```
  1000000000000000  ……   -32768
+ 1111111111111111  ……  -     1
 1|0111111111111111  ……   32767
```

向下溢出

1.3 浮点型

C/C++ 中的浮点型又称实型，有单精度（`float`）、双精度（`double`）和长双精度（`long double`）。

各类型大小：通常的平台实现遵守 IEEE 754 规范。通常，`float` 型在内存中占用 4 个字节，提供 7 位有效数字；`double` 型在内存中占用 8 个字节，提供 16 位有效数字。

因为浮点型数据长度是有限的，所以浮点数存在计算误差。虽然浮点数精度越高计算结果越精确，但其处理时间也长。一个较大的浮点数与一个很小的浮点数做加法时，由于精度限制使得很小的浮点数被忽略了，从而使得这样的加法无意义。

我们可以观察一个浮点数的简单应用例子，代码如下：

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    float a = 0.00000678f;
    double b = 0.0000067;
    a = a + 111111.111f;
    b = b + 111111.1;
    cout << setiosflags(ios::fixed) << setprecision(16);
    cout << "a = " << a << ", b = " << b << endl;
}
```

运行结果：

```
a = 111111.1093750000000000, b = 111111.1000066999986302
```

1.4 字符型

C/C++ 中的字符型分为有符号（`signed char`）和无符号（`unsigned char`）两种。通常我们程序中使用的 `char` 是否为有符号属于 Implementation Defined，实践中我们通常不需要在意该问题。

类型大小：字符型数据在内存中占用 1 个字节。

在这里我们需要了解一个计算机用于存储字符信息的工具——ASCII 码。

ASCII 全称为 **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange，美国信息交换标准代码，是基于拉丁字母的一套电脑编码系统。它主要用于显示现代英

语。在 ASCII 码中，所有常用字符都对应着一个特定的值，它们组成了 ASCII 码表。ASCII 码表部分如下：

Control Characters				Graphic Symbols											
Name	Dec	Binary	Hex	Symbol	Dec	Binary	Hex	Symbol	Dec	Binary	Hex	Symbol	Dec	Binary	Hex
NUL	0	0000000	00	space	32	0100000	20	@	64	1000000	40	'	96	1100000	60
SOH	1	0000001	01	!	33	0100001	21	A	65	1000001	41	a	97	1100001	61
STX	2	0000010	02	"	34	0100010	22	B	66	1000010	42	b	98	1100010	62
ETX	3	0000011	03	#	35	0100011	23	C	67	1000011	43	c	99	1100011	63
EOT	4	0000100	04	$	36	0100100	24	D	68	1000100	44	d	100	1100100	64
ENQ	5	0000101	05	%	37	0100101	25	E	69	1000101	45	e	101	1100101	65
ACK	6	0000110	06	&	38	0100110	26	F	70	1000110	46	f	102	1100110	66
BEL	7	0000111	07	'	39	0100111	27	G	71	1000111	47	g	103	1100111	67
BS	8	0001000	08	(	40	0101000	28	H	72	1001000	48	h	104	1101000	68
HT	9	0001001	09	)	41	0101001	29	I	73	1001001	49	i	105	1101001	69
LF	10	0001010	0A	*	42	0101010	2A	J	74	1001010	4A	j	106	1101010	6A
VT	11	0001011	0B	+	43	0101011	2B	K	75	1001011	4B	k	107	1101011	6B
FF	12	0001100	0C	,	44	0101100	2C	L	76	1001100	4C	l	108	1101100	6C
CR	13	0001101	0D	−	45	0101101	2D	M	77	1001101	4D	m	109	1101101	6D
SO	14	0001110	0E	.	46	0101110	2E	N	78	1001110	4E	n	110	1101110	6E
SI	15	0001111	0F	/	47	0101111	2F	O	79	1001111	4F	o	111	1101111	6F
DLE	16	0010000	10	0	48	0110000	30	P	80	1010000	50	p	112	1110000	70
DC1	17	0010001	11	1	49	0110001	31	Q	81	1010001	51	q	113	1110001	71
DC2	18	0010010	12	2	50	0110010	32	R	82	1010010	52	r	114	1110010	72
DC3	19	0010011	13	3	51	0110011	33	S	83	1010011	53	s	115	1110011	73
DC4	20	0010100	14	4	52	0110100	34	T	84	1010100	54	t	116	1110100	74
NAK	21	0010101	15	5	53	0110101	35	U	85	1010101	55	u	117	1110101	75
SYN	22	0010110	16	6	54	0110110	36	V	86	1010110	56	v	118	1110110	76
ETB	23	0010111	17	7	55	0110111	37	W	87	1010111	57	w	119	1110111	77
CAN	24	0011000	18	8	56	0111000	38	X	88	1011000	58	x	120	1111000	78
EM	25	0011001	19	9	57	0111001	39	Y	89	1011001	59	y	121	1111001	79
SUB	26	0011010	1A	:	58	0111010	3A	Z	90	1011010	5A	z	122	1111010	7A
ESC	27	0011011	1B	;	59	0111011	3B	[	91	1011011	5B	{	123	1111011	7B
FS	28	0011100	1C	<	60	0111100	3C	\	92	1011100	5C	|	124	1111100	7C
GS	29	0011101	1D	=	61	0111101	3D	]	93	1011101	5D	}	125	1111101	7D
RS	30	0011110	1E	>	62	0111110	3E	^	94	1011110	5E	~	126	1111110	7E
US	31	0011111	1F	?	63	0111111	3F	_	95	1011111	5F	Del	127	1111111	7F

字符型变量参与算术运算等操作时，将会隐式转换为整型；其隐式转换所得数值为其对应 ASCII 码。另外整型也可转换为字符型。例如：

```cpp
#include <iostream>
using namespace std;
int main()
{
    int i = 'A', j = 66;
    char c1 = 'a', c2 = 98;
    cout << "i = " << i << " j = " << j << endl;
    cout << "c1 = " << c1 << " c2 = " << c2 << endl;
    cout << "c1 - 32 = " << c1 - 32 << endl;
}
```

运行结果：

```
i = 65, j = 66
c1 = a, c2 = b
c1 - 32 = 65
```

需要注意，高类型转换为低类型时，若出现溢出，属于 Implementation Defined，例如语句 `char c = 114514;` GCC 下，使用编译选项 `-Wall` 编译该语句时将生成警告。

第二章　表达式和分支控制结构

2.1　运算符与表达式

C++ 程序求解问题的基本操作是运算。C++ 语言丰富的运算符及其表达式构成实现算法的基本步骤，在不同程序结构的控制下有机地组织在一起形成程序。

我们的主要任务是了解 C++ 的常用运算符以及它们的具体用法。

1. 操作对象的数目：通常 C++ 运算符的操作对象是两个或一个数字，但也有针对两者以上的运算符，例如三目运算符。我们通常遇到的运算符都是一目或两目运算符。

2. 运算符的优先级：同一个式子中不同的运算符进行计算时，其运算次序存在先后之分。运算符的优先顺序和算术计算中各运算符的优先级相同。在一个式子中如果有两个以上同一优先级的运算符，其运算次序是按运算符的结合性来处理的。C++ 语言运算符分为左结合（方向）和右结合（方向）。

算术运算符

运算符	功能	目	结合性	用法
+ -	取正值 取负值	单目 单目	自右向左 自右向左	+expr -expr
* / %	乘法 除法 整数求余/模数运算	双目 双目 双目	自左向右 自左向右 自左向右	expr1 * expr2 expr1 / expr2 expr1 % expr2
+ -	加法 减法	双目 双目	自左向右 自左向右	expr1 + expr2 expr1 - expr2

几个例子：

```
1 -6 // 结果为-6
2 5 + 7 // 结果为12
```

```
35.5 % 7 // 错误，取模运算符只能对整数操作
8 % 3 // 结果为2
```

自增自减运算符

这类运算符的功能是令变量在自身基础上进行加减操作。

运算符	功能	目	结合性	用法
++ --	后置自增 后置自减	单目 单目	自右向左 自右向左	lvalue++ lvalue--
++ --	前置自增 前置自减	单目 单目	自右向左 自右向左	++lvalue --lvalue

```
int m = 4, n;
// 现分别进行如下操作：
n = ++m; // M 先增 1，M 为 5，然后表达式使用 M 的值，赋值给 N ，N 为 5
n = --m; // M 先减 1，M 为 4，然后表达式使用 M 的值，赋值给 N，N 为 4
n = m++; // 表达式先使用 M 的值，赋值给 N，N 为 4，然后 M 增 1，M 为 5
n = m--; // 表达式先使用 M 的值，赋值给 N，N 为 5，然后 M 减 1，M 为 4
```

自增自减运算符只能用于变量，而不能用于常量和表达式。

关系运算符

这类运算符返回的结果是 `bool` 值。

运算符	功能	目	结合性	用法
< <= > >=	小于比较 小于等于比较 大于比较 大于等于比较	双目 双目 双目 双目	自左向右 自左向右 自左向右 自左向右	expr1 < expr2 expr1 <= expr2 expr1 > expr2 expr1 >= expr2
== !=	相等比较 不等比较	双目 双目	自左向右 自左向右	expr1 == expr2 expr1 != expr2

几个例子：

```
int a = 5, b = 6, c = 6;
3 > 4 // 结果为 FALSE
a < b // 结果为 TRUE
b != c // 结果为 FALSE
b >= c // 结果为 TRUE
```

逻辑运算符

运算符	功能	目	结合性	用法
!	逻辑非	单目	自右向左	!expr
&&	逻辑与	双目	自左向右	expr1 && expr2
||	逻辑或	双目	自左向右	expr1 || expr2

附上真值表：

expr1	expr2	expr1 && expr2	expr1 || expr2	!expr1	!expr2
假（0）	假（0）	假（0）	假（0）	真（1）	真（1）
假（0）	真（非0）	假（0）	真（1）	真（1）	假（0）
真（非0）	假（0）	假（0）	真（1）	假（0）	真（1）
真（非0）	真（非0）	真（1）	真（1）	假（0）	假（0）

在给出一个逻辑运算或关系运算结果时，以 0 代表“假”，以 1 代表“真”，在判断一个量为真假时，以 0 代表“假”，以“非 0”代表“真”。

赋值运算符

赋值运算符的功能是将某个特定的值赋给某个特定的变量。

运算符	功能	目	结合性	用法
= += -= *= /= %= &= ^= |= <<= >>=	赋值 复合赋值	双目 双目	自右向左 自右向左	lvalue = expr lvalue+=expr lvalue-=expr lvalue*=expr lvalue/=expr lvalue%=expr lvalue&=expr lvalue^=expr lvalue|=expr lvalue<<=expr lvalue>>=expr

```
1 int k = 95, a = 6, b = 101;
2 b - a = k; // 错误，被赋值变量必须在左
3 5 = b - a; // 错误，5 是常量
4 b *= k; // 正确
```

条件运算符

运算符	功能	目	结合性	用法
? :	条件运算	三目	自右向左	expr1 ? expr2 : expr3

三目运算符翻译结果为：如果 expr1 为真，那么返回 expr2；否则返回 expr3。

例：

【例4.1】写出分段函数的C语言表达式。

$$y=\begin{cases} ax+b & x\geq 0 \\ x & x<0 \end{cases}$$

答案为：

```
y = (x >= 0 ? a * x + b : x);
```

2.2 顺序结构

C++ 程序由一条条语句组成，程序运行过程就是语句逐条执行的过程，而语句执行的次序称之为流程。有了求解问题的算法，还需要用程序将算法实现出来。多数情况下，这种实现表现为一定数量的语句和执行流程。通俗来说，C++ 顺序结构就是，代码从前往后依次执行，没有任何“拐弯抹角”，不跳过任何一条语句，所有的语句都会被执行到。首先我们来看组成程序的基本单位——语句。

1. 简单语句：简单语句多种多样，如表达式语句，函数调用语句，空语句。

```
x = t; t = a * b; // 表达式语句
print(a,b); // 函数调用语句
```

2. 复合语句：又称语句块，简称 block。

```
{
    double s, a = 5, b = 10, h = 8; // 局部声明
    s = (a + b) * h / 2.0;
    std::cout << "area = " << s << std::endl;
}
```

3. 控制语句：例如 `if` 语句，`switch` 语句等。

4. 注释：注释也是程序的一部分，可以帮助读者理解代码。在运行程序时注释并不会被编译且运行，也就是说注释对程序的运行结果没有任何影响。注释的写法如下：

单行注释

```
  // 在此处写入注释
```

多行注释

```
/* 注释从这开始


   到此结束 */
```

5. 语句书写的规范：C++ 中一个程序行可以书写无数个语句，但为了美观与方便起见，最好一个程序行只写一个语句。同时不同语句之间要用好 Enter 与 Tab 做好缩进保持可读性与美观。

2.3 C++ 的输入与输出

所谓输入是指从外部输入设备（如键盘、鼠标等）向计算机输入数据，输出是指从计算机向外部输出设备（如显示器、打印机等）输出数据。在 C++ 中我们可以用流对象（stream）进行输入输出操作。若在程序中使用流对象 `std::cin` 和 `std::cout`，应该将标准输入输出流库的头文件 `iostream` 包含到源文件中。

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

`iostream` 包含了 C++ 的标准输入输出流。第二行代码 `using namespace std` 意为引入命名空间 `std`；`cin`，`cout` 均定义于命名空间 `std`。

1. 通过 `cin` 从标准输入流（`stdin`）输入数据

```
cin >> 变量1 >> 变量2 >> ...;
```

2. 通过 `cout` 向标准输出流（`stdout`）输出数据。

```
cout << 表达式1 << 表达式2 << ...;
```

特别的，如果我们想输出一个换行，可使用 `endl`，其作用为输出一个换行符 `\n` 并清空缓冲区。在算法竞赛等神秘场合，如果不需要清空缓冲区，使用 `endl` 可能会比 `\n` 慢很多，导致某些题 TLE。

```
cout << .... << endl;
```

让我们看一个具体的例子：

```
int x, y;
cin >> x >> y;
cout << x << y << endl;
```

这就是一个输入 x，y 的值并将他们输出的例子。

3. 格式控制：当我们对输出内容的格式等有一定要求时，我们可以引入 iomanip 头文件。

例如我们希望输出 $\frac{1}{3}$ 这个分数的小数值，精确到第五位小数，那么可以这么操作：

```
#include <iomanip> //包含 IOMANIP 头文件
....
double a = 1.0 / 3;
cout << fixed << setprecision(5) << a << endl;
```

其中的 fixed 与 setprecision 规定了输出的数字将有五位小数。

如果我们想要让输出的每个数据占 n 个字位，可以使用 setw 函数。setw 函数只对紧接着的输出产生作用。当后面紧跟着的输出字段长度小于 n 的时候，在该字段前面用空格补齐，当输出字段长度大于 n 时，全部整体输出。

例如我们现在连续输出两个变量 a 和 b：

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int a = 5;
    int b = 10;
    cout << a << " " << b << endl;
}
```

输出：

```
5 10
```

这是普通的输出，a 和 b 之间隔了一个空格。

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int a = 5;
    int b = 10;
    cout << setw(5) << a;
    cout << setw(5) << b << endl;
}
```

输出：

```
    5   10
```

这是我们通过 setw 函数限定每个变量占五个字位的输出。

2.4 if 语句

if 语句的作用是计算给定的表达式，根据结果选择执行相应的语句。语句形式有两种：

1. if 形式：

```
if(a > b)
    puts("true");
```

2. if-else 形式：

如果满足 if 内的条件，就执行一条语句；否则执行 else 代码块中的语句。

```
if(check(mid))
    l = mid;
else
    r = mid - 1;
```

2.5 switch 语句

switch 语句的作用是计算给定的表达式，根据结果选择从哪个分支入口执行，语句形式为：

```
switch(表达式)
{
    case 常量表达式 1: ... 语句序列 1
    case 常量表达式 2: ... 语句序列 2
    ...
    case 常量表达式 n: ... 语句序列 n
    default: ... 默认语句序列
}
```

即表达式的值如果为常量表达式 i, 就执行对应的语句序列 i。

default 表示如果前面没有对应的常量表达式与条件吻合，就执行 default 之后的默认语句。

- `switch` 语句中 `case` 分支的语句序列可以是一个语句，也可以是任意多的语句序列，也可以没有语句。

  如果 `case` 后没有语句，则一旦执行到这个 `case` 分支，什么也不做，继续往下执行。

`switch` 语法中各个 `case` 分支和 `default` 分支的出现次序在语法上没有规定，但次序的不同安排会影响执行结果。

例如：

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int a;
    cin >> a;
    switch(a)
    {
        case 1: cout << 1;
        case 2: cout << 2;
        default: cout << 0;
    }
}
```

当 a 的值为 1，输出结果为：120；

将 `default` 语句的顺序交换：

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int a;
    cin >> a;
    switch(a)
    {
        default: cout << 0;
        case 1: cout << 1;
        case 2: cout << 2;
    }
}
```

当 a 的值为 1，输出结果为：12。

`switch` 语法中 `default` 分支是可选的，若没有 `default` 分支且没有任何 `case` 标号的值相等时，`switch` 语句将什么也不做。

`switch` 语句的分支表达式可以是任意表达式，但其值必须是整型、字符型或枚举类型。

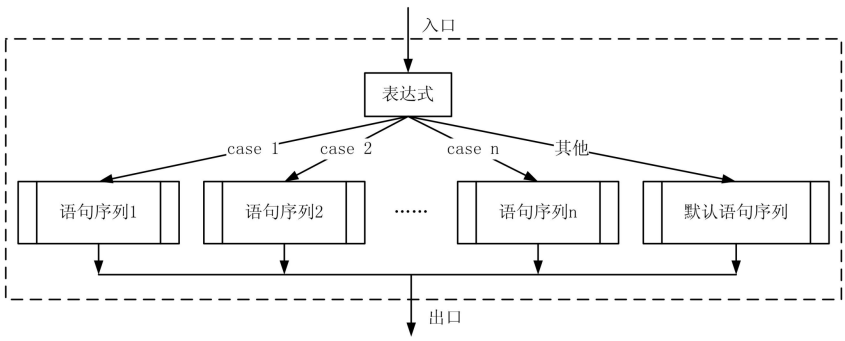
`switch` 语法中的 `case` 后的表达式必须是常量表达式且互不相同，即为整型、字符型、枚举类型的常量值，但不能包含变量。例如：

- 若 `c` 是变量，如 `case c >= 'a'&& c <= 'z':` 的写法是错的。这时应该用 `if` 语句。

在 `switch` 语句中任意位置上，只要执行到 `break` 语句，就结束 `switch` 语句的执行，转到后续语句。因此我们可以根据自身需求在 `switch` 语句中加入 `break` 控制程序何时退出：

```
switch(表达式)
{
    case 常量表达式 1: ... 语句序列 1; break;
    case 常量表达式 2: ... 语句序列 2; break;
    ...
    case 常量表达式 n: ... 语句序列 n; break;
    default: ... 默认语句序列
}
```

此时程序的结构如图：



2.6 循环语句

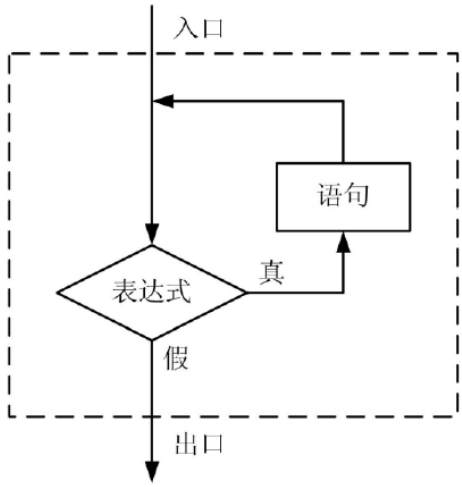
循环就是在满足一定条件时，重复执行某一段程序。C++ 中主要有三种编写循环语句的方式：`while` 语句，`for` 语句，`do-while` 语句。

2.6.1 while 语句

while 语句的大体结构为：

```
while(条件一/表达式)
{
    所要执行的操作
}
```

程序结构示意图如下：



例：计算 $s = 1 + 2 + 3 + 4 + ... + 100$，使用 while 语句编写如下：

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int sum = 0, i = 1;
    while(i <= 100)
    {
        sum += i;
        i++;
    }
    cout << sum << endl;
}
```

使用 while 循环时注意如下几点：

- 由于 while 语句先计算表达式的值，再判断是否循环，所以如果表达式的值一开始就为假，则循环一次也不执行，失去了循环的意义。

- `while` 语句循环条件可以是 C++ 语言的任意表达式。通常情况下，循环条件是关系表达式或逻辑表达式，应该谨慎出现别的表达式。
- 在循环过程中，需要有改变循环控制变量值的语句。否则循环将无法结束。

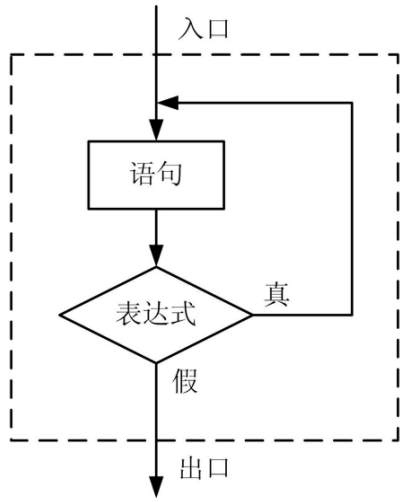
2.6.2 do while 语句

`do while` 语句的大体结构为：

```
do {
    .....
} while(表达式);
```

其中圆括号内的表达式就是循环条件。

示意图如下：



`do...while` 语句先执行后判定，`while` 语句则是先判定后执行。也就是说，无论表达式是真是假，`do...while` 语句至少要执行循环体一次，而 `while` 语句可能一次也不执行。`do...while` 语句的结构功能与 `while` 语句非常相似，两者可以相互替换。

例：连续输入多个数字，计算他们的乘积，当输入为 0 时停止。

```
#include <iostream>
using namespace std;
int main()
{
    int a = 1, ans = 1;
    do {
```

```
        ans *= a;
        cin >> a;
    } while(a!=0);
    cout << ans << endl;
}
```

2.6.3 for 语句

`for` 语句的结构为:

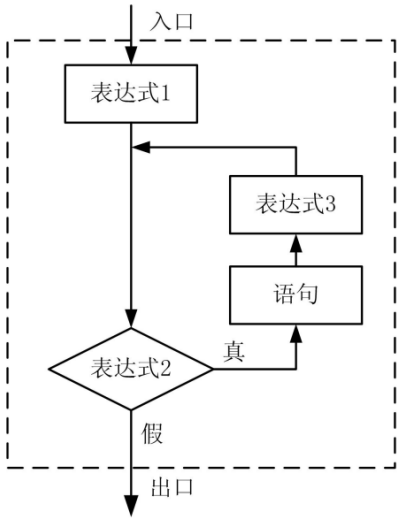
```
for(语句 1; 表达式 2; 语句 3)
{
    语句
}
```

整个循环过程中，语句 1 只运行一次，作用是给循环变量赋初始值。表达式 2 相当于是 `for` 的循环条件。语句 3 是重复执行的内容；通常是改变循环控制变量值的语句。

`for` 语句的应用结构如下：

```
for(循环初始; 循环条件; 循环控制)
{
    循环体
}
```

程序结构示意图如下：



2.6.4 循环的嵌套

当一个循环体内包含又一个循环语句时，就构成了循环的嵌套。C++ 语言的循环语句（`while`，`do`，`for`）可以互相嵌套，循环嵌套的层数没有限制，可以形成多重循环。

使用多重循环的时候，嵌套的循环控制变量不应相同。

例：打印九九乘法表

```cpp
#include <iostream>
using namespace std;
int main()
{
    for(int i = 1; i <= 9; ++i)
    {
        for(int j = 1; j <= i; ++j)
            cout << i << " * " << j << " = " << i * j << " ";
        cout << endl;
    }
}
```

运行结果：

```
1 * 1 = 1
2 * 1 = 2 2 * 2 = 4
3 * 1 = 3 3 * 2 = 6 3 * 3 = 9
4 * 1 = 4 4 * 2 = 8 4 * 3 = 12 4 * 4 = 16
5 * 1 = 5 5 * 2 = 10 5 * 3 = 15 5 * 4 = 20 5 * 5 = 25
6 * 1 = 6 6 * 2 = 12 6 * 3 = 18 6 * 4 = 24 6 * 5 = 30 6 * 6 = 36
7 * 1 = 7 7 * 2 = 14 7 * 3 = 21 7 * 4 = 28 7 * 5 = 35 7 * 6 = 42 7 * 7 = 49
8 * 1 = 8 8 * 2 = 16 8 * 3 = 24 8 * 4 = 32 8 * 5 = 40 8 * 6 = 48 8 * 7 = 56 8 *
    8 = 64
9 * 1 = 9 9 * 2 = 18 9 * 3 = 27 9 * 4 = 36 9 * 5 = 45 9 * 6 = 54 9 * 7 = 63 9 *
    8 = 72 9 * 9 = 81
```

第三章　数组

之前我们操作的数据对象通常都是单个的数据。当面对大量的数据时，我们可以使用数组。其内存模型为一块连续的内存，支持常数时间随机存取。

3.1　一维数组

定义方法为：

```
数据类型 数组名[表达式];
```

例如我们需要一个大小为 100 的 `int` 型的一维数组：

```
int arr[100];
```

在 C++ 中，数组的下标从 0 开始，也就是第一个元素是 `arr[0]`，后面以此类推，第 i 个元素的下标是 $i-1$。

值得注意的是，如下写法使用 GCC 可以编译通过，但不符合任何一版 C++ 语言标准：

```
int n;
...
int arr[n];
...
```

其中 `n` 为变量，其值不可由编译期确定为常量。此种语法被称为变长数组（Variable-Length Array，VLA）。VLA 未进入目前的任何一版 C++ 语言标准，在 C 语言中从 C99 起进入标准。尽管有部分 C++ 编译器会支持此类写法，我们仍强烈不推荐这么写。

定义数组就是定义了一块连续的空间，数组元素按序存放在这块空间中。一个数组所占用的内存遵循简单的线性计算。例如一个 `int` 型数据占用 4 byte, 那么数组 `int a[10];` 占用空间为 $4 \times 10 = 40$。

数组类型能够隐式转换为其元素类型对应的指针类型，该指针指向数组首地址。

3.1.1 一维数组的初始化

数组的初始化意味着在定义数组的时候就赋给数组某个值：

```
int arr[4] = {1, 2, 3, 4};
```

由此方法声明时可以不写长度：

```
int arr[] = {1, 2, 3, 4};
```

如此我们就得到了一个长度为 4，元素分别为 1，2，3，4 的数组。

如果给定的元素数量不及数组的大小，例如：

```
int arr[4] = {1};
```

该数组剩余的元素将会自动被赋值为 0。

开成全局变量的数组，一般情况下，其元素会被初始化为默认值（对于整型，该值为 0；对于自定义类型，由默认构造函数初始化）。

3.1.2 一维数组的访问

之前我们提过数组的下标，访问数组中任意元素只需要提供对应元素的位置即可，例如访问 `arr` 数组中的第二个元素：

```
arr[1]
```

3.1.3 数组赋值问题

若我们想让数组 `b` 的任何一个值 `b[i]` 被赋值为 `a[i]`，我们不能进行这种操作：

```
b = a;
```

因为 `b` 和 `a` 都是数组名。C++ 中数组不能进行整体赋值，对于这种情况我们必须一个一个赋值。假设 `a` 数组有 `maxn` 个元素，我们应该这么写：

```
for(int i = 0; i < maxn; ++i)
    b[i] = a[i];
```

3.1.4 数组越界问题

例如 a 数组只有 10 个元素，但是我们却在程序中修改了 a[10]，这种情况下编译器也许不会报错，运行时也许也不会报错，但是程序表现极有可能不符合预期。

3.2 二维数组

二维数组类似于一张表格，纵列与横列表示不同的意思。

roomnums	pave	price
1200	NA	1200$
3000	NA	1000$
2000	DAV	900$

这样的一张表格我们可以用二维数组来进行存储。将数据存储在一个三行三列的数组中。

二维数组的定义方式如下：

```
1 数据类型 数组名[表达式 1][表达式 2];
```

例如：arr[100][500] 就是一个有着 100 行 500 列的数组。

二维数组占用内存的计算方法也是线性的。例如上文定义的 arr 数组，空间占用为 $100 \times 500 \times 4$ byte。

3.2.1 二维数组的初始化

定义二维数组时直接进行赋值：

```
1 int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

将所有元素依照行直接全部写出，编译器会根据数组的行列数进行赋值：

```
1 int arr[2][3] = {1, 2, 3, 4, 5, 6};
```

这段代码得到的结果与第一种方法相同。

对全部数组赋值时，数组的行数可以不写，但是列数必须写出：

```
1 int arr[][3] = {{1, 2, 3}, {4, 5, 6}};
```

3.2.2 二维数组的访问

访问二维数组的元素的方法和访问一维数组元素方法一致。直接输入需要访问元素的名称即可。

第四章　指针

指针学习过程中会有比较多的技巧或者所谓规则，但对于日常使用可能意义并不大，本文主要聚焦于指针的主体理解和使用，一些较为繁琐或使用频率较低的知识点可能并不涉及，同时术语方面相对可能不会面面俱到，建议配合唐老师 PPT 以及 moodle 附件的 THU 指针讲义共同食用。

4.1　概念

4.1.1　指针与存储单元

内存区的每一个字节有一个编号，这就是**地址**，它相当于旅馆中的房间号。

在地址所标识的内存单元中存放数据，这相当于旅馆房间中居住的旅客一样。

由于**通过地址能找到所需的变量单元**，我们可以说，**地址指向该变量单元**。

将地址形象化地称为**指针**。

务必弄清楚存储单元的地址和存储单元的内容这两个概念的区别。简单来说，一个变量的地址是该变量的指针，而该指针又指向这个变量。

4.1.2　指针与指针变量

由前文可知，指针是一个变量的地址，而指针变量是存储指针的变量。

总体上，三个概念的关系是 **指针变量** $\longrightarrow$ **指针** $\longrightarrow$ **变量单元**，第一个箭头表示通过**指针变量**找到**变量单元**的指针，第二个箭头表示通过**指针**读取**变量单元**内容，

4.2 指针变量的应用

4.2.1 指针变量的定义，赋值和引用

```cpp
#include <bits/stdc++.h>
using namespace std;
int main()
{
    // 定义变量单元
    int a = 100, b = 10;
    // 定义指针变量
    int *pointer_1, *pointer_2;
    // 给指针变量赋值，值为 a, b 分别对应的地址
    pointer_1 = &a;
    pointer_2 = &b;
    // 直接输出 a, b
    cout << a << ' ' << b << endl;
    // 间接输出 a, b
    cout << *pointer_1 << ' ' << *pointer_2 << endl;
}
```

4.2.2 指针变量的函数应用

一个非常有 C 风格的应用：

```cpp
#include <bits/stdc++.h>
using namespace std;
// 定义交换函数并且以变量单元地址作为参数
void swap(int *m, int *n)
{
    int temp = *m;
    *m = *n;
    *n = temp;
}
```

4.2.3 指针与数组

数组类型能够隐式转换为其元素类型对应的指针类型，该指针指向数组首地址。

第五章　函数

5.1　函数的定义

函数定义的一般形式为：

```
返回类型 函数名(形式参数列表);

...

返回类型 函数名(形式参数列表)
{
  执行部分 （含 return）
}
```

函数名：实现函数需要确定函数名，以便使用函数时能够按照函数名进行引用

形式参数列表：形式参数列表是函数与调用者进行数据交换的途径，一般形式为 (类型 1 参数名 1，类型 2 参数名 2，...), 不过形参并不是必要的

返回类型：返回类型可以是 C++ 除数组之外的内置数据类型或自定义类型。函数也可以不返回数据，此时返回类型应该定义为 `void`：

```
void 函数名(形参)
{
    ...
}
```

函数体：实现函数最重要的就是函数体。函数体包含声明部分和执行部分，是一组能够实现特定功能的语句序列的集合。编写函数体是为了实现函数功能，故称函数定义为**函数实现**，简称**实现**。而函数头简称**接口**。

5.2 函数的参数

函数参数是实现函数的重要内容，是函数接口的首要任务，围绕这个目标需要研究：

形式参数的定义与实际参数的对应关系以及函数参数的数据传递机制，包括主调函数与被调函数的双向数据传递。

5.2.1 形式参数

函数定义中的形式参数列表，简称形参。例如：

```
1 int max(int a, int b)
2 {
3     return a > b ? a : b;
4 }
```

第一行函数名后括号内 a 和 b 就是形参。

对于形参：

函数定义时指定的形参，在未进行函数调用前，并不实际占用内存中的存储单元。只有发生函数调用时，形参才分配实际的内存单元，接受从主调函数传来的数据。

当函数调用结束后，形参占用的内存单元被自动释放。

5.2.2 实际参数

函数调用时提供给被调函数的参数称为实际参数，简称实参。

实参必须要有确定的值，因为调用函数会将它们传递给形参，实参可以是常量、变量或者表达式，还可以是函数的表达式。例如

```
1 x = max(a, b); // 实参为A, B
2 y = max(a + 3, 128); // 实参为A + 3, 128
3 z = max(max(a, b), c); // 实参为 MAX(A, B) 和 C
```

对于实参：

实参的类型、次序和数目要与形参一致。如果参数数目不一致，则出现编译错误。如果参数次序不一致，则传递到被调函数中的数据就不合逻辑，难有正确的程序结果。如果参数类型不一致时，则函数调用时按形参类型隐式类型转换实参。

5.2.3 参数传递机制

值传递

形参作为被调函数的内部变量来处理，即开辟内存空间以存放由主调函数复制过来的实参的值，从而成为实参的一个副本。值传递的特点是被调函数对形参的任何操作都是对内部变量进行，不会影响到主调函数的实参变量的值。

值传递时，实参数据传递给形参是单向传递，即只能由实参传递给形参，而不能由形参传回给实参

```
1 void fun(int x, int y, int m)
2 {
3     m = x > y ? x : y; // 仅修改函数内部的 m
4 }
5 void caller( ) // 主调函数，调用者
6 {
7     int a = 10, b = 5, k = 1;
8     fun(a, b, k); // 实参值传递
9 }
```

引用传递

与值传递不同，被调函数对形参的任何操作都会直接传回给实参，具体结合指针的讲义进行理解，传入的可以是引用或地址，多用于函数内部会对变量进行直接修改，而值传递对主调函数内的变量没有影响，使用时结合具体情况挑选传递方式。

第六章　结构体、函数重载

6.1　结构体

一种称为结构的构造型数据类型。结构是一组相关的不同类型的数据的集合。结构类型为处理复杂的数据提供了便利的手段。moodle 上学籍管理的题目就是典型的适合用结构体解决的题目。

其定义方式如下：

```
struct 结构类型名
{
  类型1 成员名1;
  类型2 成员名2;
  ……
};
```

而其初始化可以参考下例：

```
struct date
{
  int year;
  int month;
  int day;
};

struct person
{
  char name[30];
  char sex;
  struct date birthday;
};
date today = { 2001, 10, 1};
person man = { "zhang",'M',{1980,3,28} };
```

对于访问结构体中的数据，可以使用以下方法：

```
结构变量.成员变量名;
//以下为对 TODAY 结构体中数据的访问
today.year=2010; today.month=1; today.day=1;
```

指明结构成员的符号 . 是一种运算符，它的含义是访问结构中的成员。这样 today.year 的含义就是访问结构变量 today 中的名为 year 的成员。

需要注意的是，结构的成员可以像一般变量　样参与各种操作和运算，　作为代表结构整体的结构变量，能够对结构进　整体操作的运算不多，只有进行赋值 = 和取地址 & 操作。

6.2　结构数组

结构与数组的关系有两重：

在结构中使用数组类型作为结构的一个成员

用结构类型作为数组元素的类型构成数组

具体可以参考学籍管理问题中使用结构数组对应不同的学生，而确定一个学生后可以通过定义的结构体类型进一步访问其对应的班级学号等数据。

定义结构数组以下方代码为例：

```
struct person
{
  char name[30];
  char sex;
  struct date birthday;
} man;
struct person student[100];
```

访问结构数组中的数据可以通过

```
结构数组名[下标].成员名
```

6.3　结构作为函数参数

调用函数时，可以将结构作为参数进行传递。

在函数间传递结构的方式有三种:

1. 向函数传递结构的成员变量：同前文函数定义中传递参数方式。

2. 向函数传递整个结构：将结构变量作为形参，通过值传递方式将整个结构传递给函数。

   注意：

   在被调用函数中，对结构形参变量的值的任何修改，都不会影响到调用函数中的结构变量。

   将结构作为函数参数时，结构的类型必须完全匹配。

3. 向函数传递结构的地址或引用：向函数中传递结构的地址或引用要将函数的形参定义为指向结构的指针或引用，在调用时要用结构的地址或引用作为实参。这可以对定义好的结构变量在函数中进行直接修改。

6.4 函数重载

指声明多个同名函数，其形参不同。用于避免由于变量类型造成的重复编码。典型例子[2]：

std::min

Defined in header <algorithm>

`template< class T > const T& min( const T& a, const T& b );`	(1)	(constexpr since C++14)
`template< class T, class Compare > const T& min( const T& a, const T& b, Compare comp );`	(2)	(constexpr since C++14)
`template< class T > T min( std::initializer_list<T> ilist );`	(3)	(since C++11) (constexpr since C++14)
`template< class T, class Compare > T min( std::initializer_list<T> ilist, Compare comp );`	(4)	(since C++11) (constexpr since C++14)

于是同名函数 `std::min` 可以服务我们以以上四种形式传入的形参，示例分别如下：

```
1 std::min(1, 2) // 第一种，返回 1
2 std::min(1, 2, std::greater<int>()) // 第二种，返回 2
3 std::min({1, 1, 4, 5, 1, 4}) // 第三种，返回 1
4 std::min({1, 1, 4, 5, 1, 4}, std::greater<int>()) // 第四种，返回 5
```

其便利之处在于我们不需要为这几种形参传入形式分别定义 4 个异名函数。

第七章　对象和类

7.1　面向对象编程

面向对象编程是一种编程范式。它以将相关数据和函数分组到信息“孤岛”中的思想为基础，这些孤岛称为“对象”[3]。

例：一段体现了“面向对象精神”的 C 语言代码[4]

```c
struct Student {
    char *name; //姓名
    int num; //学号
    int age; //年龄
    char group; //所在学习小组
    float score; //成绩
};
char* GetStudentName(struct Student* stu)
{
    // 略
}
void SetStudentName(struct Student* stu, char* newName)
{
    // 略
}
int main()
{
    struct Student s1, s2, s3, s4; // 创建了多个学生
    SetStudentName(&s1, "小明");
    SetStudentName(&s2, "小红");
}
```

有了 `class` 或者 `struct`，我们就能把数据放在一起，能创建多个同类型的对象。只要合理定义了类的变量，事情就成了大半；再加上合理定义的操作（也就是

方法或函数），就已经足够写出足够好的程序出来。世界上有很多非常重要的程序，从操作系统到火控雷达，从火箭上天到载人登月，都就是用这种很朴素的思想写出来的[4]。

7.2 定义类

类定义时必须给出各个数据成员（data member）的数据类型声明。每个类还可以包含成员函数，能够访问类自身的所有成员。

通过标识，我们能够控制外部对类成员的访问。通常我们只想向用户暴露一部分接口，这部分可用 `public` 标识；另一部分不希望用户胡乱接触，这部分可用 `private` 标识。

例：矩形类定义及应用

```
#include <iostream>
#include <cmath>
class Rectangle {
private:
    int x1, y1, x2, y2;
public:
    Rectangle(int a, int b, int c, int d) : x1(a), y1(b), x2(c), y2(d) {}
    int Circumference() const;
    int Area() const;
};
int Rectangle::Circumference() const
{
    return 2 * (std::abs(x2 - x1) + std::abs(y2 - y1));
}
int Rectangle::Area() const
{
    return std::abs(x2 - x1) * std::abs(y2 - y1);
}
int main()
{
    int a, b, c, d;
    std::cin >> a >> b >> c >> d;
    Rectangle t(a, b, c, d);
    std::cout << t.Circumference() << " " << t.Area();
}
```

该示例下，使用 `Rectangle` 的用户将无法直接访问 `x1 y1 x2 y2` 等成员。

7.3 构造函数与析构函数

类中两种特殊的成员函数，分别在对象初始化时和对象将被销毁时调用。

7.3.1 构造函数

构造函数的名称与类名相同，本身无类型，无返回值。

构造函数的主要作用为对象属性的初始化。

我们上例中已经给出了一个构造函数的例子，其等价于：

```
1 Rectangle::Rectangle(int a, int b, int c, int d)
2 {
3     x1 = a;
4     y1 = b;
5     x2 = c;
6     y2 = d;
7 }
```

通过使用初始化列表，我们简化了上述代码。

7.3.2 析构函数

当对象将被销毁（静态对象，程序结束时；局部对象，代码块结束时；通过 `new` 生成的对象，用户调用 `delete`）时执行，通常用于一些额外的清理工作。

例：

```
1 Rectangle::~Rectangle()
2 {
3     // DO SOMETHING
4 }
```

7.4 this 指针

一个特殊的指针，指向当前对象自己。一个对象的所有成员函数中都可以使 `this` 指针。

第八章　使用类

8.1　运算符重载

与函数重载相似，通过重定义运算符，能够将不同形式的形参传入运算。

通过重载以类为形参的运算符函数，可能能够以运算形式操作对象，以此简化代码形式，使代码看起来更加自然优美。

例：矩形类定义及应用

```
#include <iostream>
#include <cmath>
class Rectangle
{
private:
    int x1, y1, x2, y2;
public:
    Rectangle(int a, int b, int c, int d) : x1(a), y1(b), x2(c), y2(d) {}
    int Circumference() const;
    int Area() const;
    int operator+(const Rectangle &rhs) const;
    int operator*(const Rectangle &rhs) const;
};
int Rectangle::Circumference() const
{
    return 2 * (std::abs(x2 - x1) + std::abs(y2 - y1));
}
int Rectangle::Area() const
{
    return std::abs(x2 - x1) * std::abs(y2 - y1);
}
int Rectangle::operator+(const Rectangle &rhs) const
{
    return this->Circumference() + rhs.Circumference();
```

```
}
int Rectangle::operator*(const Rectangle &rhs) const
{
    return this->Area() + rhs.Area();
}
int main()
{
    int a1, b1, c1, d1, a2, b2, c2, d2;
    std::cin >> a1 >> b1 >> c1 >> d1 >> a2 >> b2 >> c2 >> d2;
    Rectangle A(a1, b1, c1, d1), B(a2, b2, c2, d2);
    std::cout << A + B << " " << A * B;
}
```

8.2 友元函数

动机：若我们不得不从类访问类的非公开成员时，可以使用友元函数。

考虑如下场景，我们需要重载双目运算符 `+`；左目传入整数，右目传入自定义类型 `Point`。此时我们若需要访问 `Point` 类型对象的非公开成员，则需要使用友元函数。

例：点类 `Point` 及友元函数

```
#include <iostream>
#include <cmath>
class Point
{
private:
    int x, y;
public:
    Point(int a = 0, int b = 0) : x(a), y(b) {}
    friend std::istream& operator>>(std::istream& lhs, Point& rhs);
    friend int calcH(const Point &p1, const Point &p2);
    friend int calcV(const Point &p1, const Point &p2);
};
int calcH(const Point &p1, const Point &p2)
{
    return std::abs(p1.x - p2.x);
}
int calcV(const Point &p1, const Point &p2)
{
    return std::abs(p1.y - p2.y);
```

```
}
std::istream& operator>>(std::istream& lhs, Point& rhs)
{
    return (lhs >> rhs.x >> rhs.y);
}
int main()
{
    Point a, b;
    std::cin >> a >> b;
    std::cout << calcH(a, b) << " " << calcV(a, b);
}
```

缺点：破坏封装，我们应尽量确保在合理的场合使用它。

8.3 类的自动转换和强制类型转换

对于自定义类，若我们提供转换函数，则可以将自定义类对象转换为其他类型。

一个简单示例：

```
#include <iostream>
class sth
{
private:
    int x;
public:
    sth(int a) : x(a) {}
    operator int() {return x;}
};
int main()
{
    sth x(114514);
    std::cout << x;
}
```

输出：

```
114514
```

该示例中，输出 `x` 时发生了隐式类型转换。

第九章　类和动态内存分配

动机：设想神秘用户分别以如下方式使用我们提供的自定义类型 `String`:

```
class String
{
private:
    char* content;
...
};

...

String a, b;
...
String c = a; // (1)
b = a; // (2)
auto d = new String(...); delete d; // (3)
```

对于 (1)、(2)，我们希望每次操作后总是每一个 `String` 类内部的 `content` 指针对应各自的一块内存；对于 (3)，我们需要在析构函数回收这块内存。下面我们一一给出解决方案。

9.1　复制构造函数

对于 `String c = a;` 这类操作，其会调用 `String` 类的复制构造函数。因此我们需要对此进行重载，示例如下：

```
class String
{
private:
    char* content;
...
```

```
public:
    String(const String& rhs)
    {
        content = new char[strlen(rhs.content) + 1];
        strcpy(content, rhs.content);
    }
};
```

如此，我们解决了上述问题 (1)。

9.2 赋值运算符

`b = a` 使用了赋值运算符，我们应当对此进行重载。

```
class String
{
private:
    char* content;
...
public:
...
    String& operator= (const String& rhs)
    {
        if(&rhs == this) return *this;
        delete []content;
        content = new char[strlen(rhs.content) + 1];
        strcpy(content, rhs.content);
        return *this;
    }
};
```

其中值得注意的是，我们对 `&rhs == this` 的情况进行了特判，否则将可能导致访问被释放的空间。

9.3 析构函数

在对象被销毁时，我们应释放其申请的动态空间，防止内存泄漏。

```
class String
{
private:
```

```
    char* content;
...
public:
...
    ~String()
    {
        delete []content;
    }
};
```

例：Exercise 1 of Chapter 13 again using dynamic memory allocation

```
#include <bits/stdc++.h>
using namespace std;
class Cd
{
private:
    char *performers;
    char *label;
    int selections;
    double playtime;
public:
    Cd(const char *s1, const char *s2, int n, double x);
    Cd(const Cd &d);
    Cd();
    virtual ~Cd();
    virtual void Report() const;
    Cd &operator=(const Cd &d);
};

class Classic : public Cd
{
private:
    char *primaryWork;
public:
    Classic(const char *prim = "", const char *s1 = "", const char *s2 = "", int
          n = 0, double x = 0) : Cd(s1, s2, n, x)
    {
        if (prim == NULL)
            prim = "";
        primaryWork = new char[strlen(prim) + 1];
```

```
29          strcpy(primaryWork, prim);
30      }
31      Classic(const Classic &d) : Cd(d)
32      {
33          primaryWork = new char[strlen(d.primaryWork) + 1];
34          strcpy(primaryWork, d.primaryWork);
35      }
36      Classic &operator= (const Classic &d)
37      {
38          if (&d == this)
39              return *this;
40          delete[] primaryWork;
41          primaryWork = new char[strlen(d.primaryWork) + 1];
42          strcpy(primaryWork, d.primaryWork);
43          Cd::operator=(d);
44          return *this;
45      }
46      void Report() const
47      {
48          cout << primaryWork << ",";
49          Cd::Report();
50      }
51      ~Classic()
52      {
53          delete[] primaryWork;
54      }
55  };
56
57  void Bravo(const Cd &disk);
58
59  int main()
60  {
61      Cd c1("Beatles", "Capitol", 14, 35.5);
62      Classic c2 = Classic("Piano Sonata in B flat, Fantasia in C", "Alfred
            Brendel", "Philips", 2, 57.17);
63      Cd *pcd = &c1;
64      cout << "Using object directly:\n";
65      c1.Report();
66      c2.Report();
67      cout << "Using type cd * pointer to objects:\n";
68      pcd->Report();
69      pcd = &c2;
70      pcd->Report();
```

```
71      cout << "Calling a function with a Cd reference argument:\n";
72      Bravo(c1);
73      Bravo(c2);
74      cout << "Testing assignment: ";
75      Classic copy;
76      copy = c2;
77      copy.Report();
78      return 0;
79  }
80
81  void Bravo(const Cd &disk)
82  {
83      disk.Report();
84  }
85
86  Cd::Cd(const char *s1, const char *s2, int n, double x)
87  {
88      if (s1 == NULL)
89          s1 = "";
90      if (s2 == NULL)
91          s2 = "";
92      performers = new char[strlen(s1) + 1];
93      strcpy(performers, s1);
94      label = new char[strlen(s2) + 1];
95      strcpy(label, s2);
96      selections = n;
97      playtime = x;
98  }
99  Cd::Cd(const Cd &d)
100 {
101     performers = new char[strlen(d.performers) + 1];
102     strcpy(performers, d.performers);
103     label = new char[strlen(d.label) + 1];
104     strcpy(label, d.label);
105     selections = d.selections;
106     playtime = d.playtime;
107 }
108 Cd::Cd()
109 {
110     Cd("", "", 0, 0);
111 }
112 Cd::~Cd()
113 {
```

```
114     delete[] performers;
115     delete[] label;
116 }
117 void Cd::Report() const
118 {
119     cout << performers << "," << label << "," << selections << "," << playtime
            << endl;
120 }
121 Cd &Cd::operator= (const Cd &d)
122 {
123     if (&d == this)
124        return *this;
125     delete[] performers;
126     performers = new char[strlen(d.performers) + 1];
127     strcpy(performers, d.performers);
128     delete[] label;
129     label = new char[strlen(d.label) + 1];
130     strcpy(label, d.label);
131     selections = d.selections;
132     playtime = d.playtime;
133     return *this;
134 }
```