

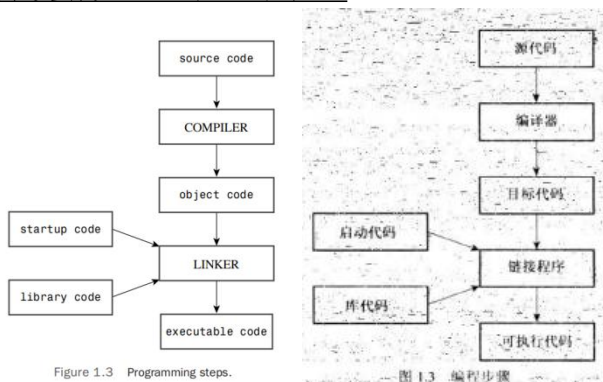
C++总提纲(2024. 7)

*该提纲使用的是 C++11 标准，删改了部分内容。

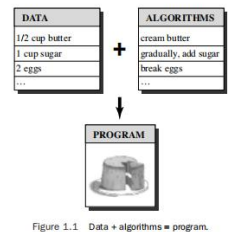
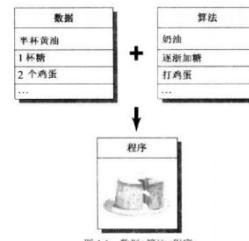
第一部分 入门基础

一、C++简介

1. 融合的编程方式：C 语言代表的**过程性语言**、**面向对象 (OOP)** 编程、C++模板支持的**泛型编程**。
2. **汇编语言**依赖于计算机内部的**机器语言**（都属于**低级语言**，直接操作硬件。汇编语言是机器语言的助记符），C、C++是**高级语言**。
3. **数据+算法=程序**，C 语言为**结构化编程**（顺序、循环、分支）。
4. C++语言的基本模块：**函数**
5. C++由 Bjarne Stroustrup 于 20 世纪 80 年代在贝尔实验室开发。
6. 在一定程度上来说，C++是 C 的超集。（几乎所有的有效 C 语言程序都是有效的 C++程序）但 C++并没有完全包含 C 语言。



7. 将程序的源代码译为机器语言，形成**目标代码** (object code, .obj); 将**启动代码**与**库代码**和**目标代码**链接在一起，即**链接**，得到**可执行文件**。



二、第一个程序

示例程序: 摘自 C++ PRIMER PLUS (6TH EDITION)

```
// myfirst.cpp--displays a message
```

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    using namespace std;
```

```
    cout << "Come up and C++ me some time.";
```

```
    cout << endl;
```

```
    cout << "You won't regret it!" << endl;
```

```
// If the output window closes before you can read it,
```

```
// add the following code:
```

```
    // cout << "Press any key to continue." << endl;
```

```
    // cin.get();
```

```
    return 0;
```

```
}
```

```
// a PREPROCESSOR directive
```

```
// function header
```

```
// start of function body
```

```
// make definitions visible
```

```
// message
```

```
// start a new line
```

```
// more output
```

```
// terminate main()
```

```
// end of function body
```

7. 单行注释：// 开头；多行注释：/* */ 开头。编译器会忽略注释。

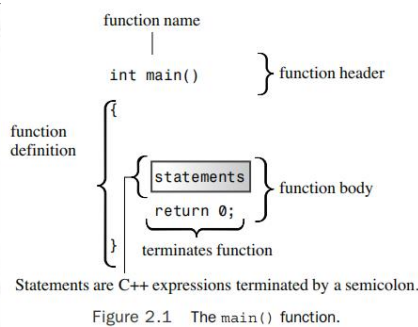
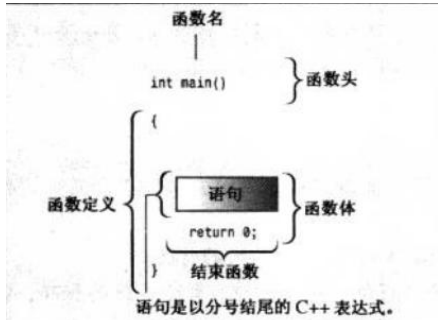
8. ▲C++对大小写敏感，Cout 与 cout 并不一样。

9. C++源文件的扩展名：**.cpp**

10. main() 函数

main() 是默认的主函数。函数定义由花括号及其里面的语句组成，int main() 叫做**函数头**，花括号中包括的语句叫**函数体**，指出了函数具体做什么。return 0; 是**返回语句**，结束该函数，返回 0 表示程序执行成功并退出。

11. 每条完整的指令称为**语句**，以分号结束。



12. 位于函数名前面的部分叫做**函数返回类型**，函数名后括号里的部分叫做**形式参数列表**（简称**形参列表**）。

13. `main()` 函数被**启动代码**调用。

14. C++ 函数给调用参数返回的值是**返回值**。此处 `int` 表示整数值。空括号意味 `main()` 不接受参数。

15. `int main(void)` 是一种变体，**`void` 关键字**指出函数不接受任何参数，与 `int main()` 等价。

16. `#include<iostream>` 将 `iostream` 头文件引入。`#include<标准头文件>` 是**预处理指令**，以 `#` 开头，会在源代码编译之前把头文件内容添加到程序中。`iostream` 是标准输入输出流，`io` 表示输入输出。

17. 像 `iostream` 这样的文件叫**包含文件**，也叫**头文件**。

C 语言头文件的后缀为 `.h`，C++ 的头文件没有后缀。C++ 可以使用 C 语言标准头文件，需去掉 `.h` 后缀并加上 `c` 前缀。如：`math.h` -> `cmath`

18. `using namespace std;` 是**using 编译指令**，表示使用 **`std`**（即**标准名称空间**）名称空间。

如果名称空间 `a` 中有函数 `b()`，则调用时为 `a::b()`。

19. `using std::cout;` 表示只使用 `std::cout` 这个名称，而不是将整个 `std` 都导入。

20. `cout<<表达式 0<<表达式 1<<.....;` 可以把表达式插入到输出流中输出。

21. `cout << "Come up and C++ me some time.";` 其中的**字符串**以 “ ” 扩起。（注意是英文标点符号）

22. **控制符** `endl`: 进行换行，在名称空间 `std` 中被定义。其等价于**换行符** ‘`\n`’（注意：字符串用 “ ” 引起，**字符**用 ‘ ’ 引起），其被视作一个字符。在字符串中包含它可以换行。

23. **空格、制表符与回车**统称**空白**。

24. **声明语句**声明变量：类型名称 变量名(=值);如 `int total;`或 `int total=0;`

25. 变量名标识了一个**内存单元**，因其可以修改，称作**变量**。

26. 程序中的声明语句叫**声明定义语句**，简称为**定义**。会导致编译器为其分配**内存空间**。

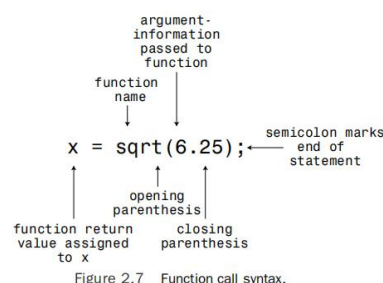
27. C++ 中还有**引用声明**，**声明不一定是定义**，但**定义一定是声明**。

28. `=`是**赋值运算符**，使用方法：变量名=值; 其从右向左执行。可以连续使用赋值运算符。

29. `cout` 可以输出变量，插入时输入变量名。

30. 使用 `cin`: `cin` 可以进行输入，用法 `cin>>变量 0>>.....;`输入多个值用空格回车分隔。

31. `cmath` 中的 `sqrt()` 计算算术平方根。`sqrt(25)` 值为 5. `sqrt(25)` 叫**函数调用**，被调用的函数叫做**被调用函数**（**被调函数**，**called function**），包含函数调用的函数叫**调用函数**（**主调函数**，**calling function**）。



32. 函数调用圆括号的值是**实际参数**（**实参**）。

33. 函数原型：`sqrt()` 的函数原型：

`double sqrt(double);`

`double` 表示双精度浮点数（小数）

函数原型只描述**函数接口**。

34. 在创建变量时赋值叫**初始化**。

35. 用户定义函数：语法：

返回类型 函数名(形参列表)

```
{
    语句
}
```

原型：返回类型 函数名(形参列表或形参类型的列表)；

在形参列表中，每个参数以“类型 参数名”的形式出现，中间以逗号分隔。

▲每个函数都是独立的，不可嵌套。

例：int solve(int,double); int solve(int a,double b){...}

无返回值的函数返回类型为 void.

36. ▲变量名和函数名的命名规则：

- (1) 不能使用关键字作为名称
- (2) 不能在名称中带有空格
- (3) 在同一作用域下，一些名称不可重复（函数重载除外）
- (4) 名称以字母、下划线开头，以字母、下划线与数字组成。

37. 在函数外使用 using 指令：对所有函数有效，在函数里使用 using 指令：对本函数有用。

总结：简单程序的结构

```
#include<标准头文件>
//...引入若干个头文件
using namespace std;
int main() //或 int main(void)
{
    //一些语句
    return 0; //程序成功执行后退出
}
```

三、基本数据类型

38. 整型：没有小数部分的数字。分为**符号整型**与**无符号整型**。符号整型有正数、0、负数。而无符号整型没有负数。

(1) 符号整型 **short**、**int**、**long**、**long long**

①. short 至少 16 位，int 至少与 short 一样长，long 至少 32 位，且至少与 int 一样长，long long 至少 64 位，且至少与 long 一样长。

②. 大部分系统使用的长度：

short 为 16 位 int 和 long 为 32 位 long long 为 64 位

③. 在头文件 climits 中表示各种限制的符号名称，如下：

climits 中的符号常量	
符号常量	表示
CHAR_BIT	char 的位数
CHAR_MAX	char 的最大值
CHAR_MIN	char 的最小值
SCHAR_MAX	signed char 的最大值
SCHAR_MIN	signed char 的最小值
UCHAR_MAX	unsigned char 的最大值
SHRT_MAX	short 的最大值
SHRT_MIN	short 的最小值
USHRT_MAX	unsigned short 的最大值

符号常量	表示
INT_MAX	int 的最大值
INT_MIN	int 的最小值
UINT_MAX	unsigned int 的最大值
LONG_MAX	long 的最大值
LONG_MIN	long 的最小值
ULONG_MAX	unsigned long 的最大值
LLONG_MAX	long long 的最大值
LLONG_MIN	long long 的最小值
ULLONG_MAX	unsigned long long 的最大值

Table 3.1 Symbolic Constants from climits

Symbolic Constant	Represents
CHAR_BIT	Number of bits in a char
CHAR_MAX	Maximum char value
CHAR_MIN	Minimum char value
SCHAR_MAX	Maximum signed char value
SCHAR_MIN	Minimum signed char value
UCHAR_MAX	Maximum unsigned char value
SHRT_MAX	Maximum short value
SHRT_MIN	Minimum short value
USHRT_MAX	Maximum unsigned short value
INT_MAX	Maximum int value
INT_MIN	Minimum int value
UINT_MAX	Maximum unsigned int value
LONG_MAX	Maximum long value
LONG_MIN	Minimum long value
ULONG_MAX	Maximum unsigned long value
LLONG_MAX	Maximum long long value
LLONG_MIN	Minimum long long value
ULLONG_MAX	Maximum unsigned long long value

④. **符号常量**以#define 定义，#define 是预处理指令，用法：

#define 名称 表达式（表达式可省略，只定义一个名称）

这些定义也叫**宏定义**。在预处理时会进行**文本替换**。

⑤. 初始化：可以将变量初始化为另一个变量（已定义）、表达式、**字面值常量**。

⑥. C++的初始化方法：类型名 变量名(表达式)；

将**大括号初始化器**用于单值变量：类型名 变量名={表达式}；

***C++11 初始化方式：将大括号初始化器用于单值变量：类型名 变量名{表达式}；

大括号中可以不包含任何信息，变量会初始化为 0

类型名 变量{}；或 类型名 变量={}； 其中变量的值为 0.

(2) 无符号类型：

unsigned short、unsigned int(unsigned)、unsigned long、unsigned long long

(3) 整型字面量:

①C++整型的三种表示: 十进制、八进制、十六进制

如果第一位为 1~9, 基数为 10; 如果第一位为 0, 第二位为 1~7, 基数为 8; 如果前两位为 0x 或 0X, 基数为 16. (基数为 R 表示 R 进制数)

▲▲计算机中的数据都以二进制存储。

②cout 的控制符 hex 表示输出 16 进制整数, oct 表示输出 8 进制整数, dec 表示输出 10 进制整数。默认为 10 进制整数。

③C++将整型常量存储为 int 类型

④其他类型常量的表示:

前后缀	表示的类型	前后缀	表示的类型
l 或 L 后缀	long	u 或 U 后缀	unsigned int
ul 后缀 (不区分大小写与顺序)			unsigned long
ll\LL 后缀	long long	ull\Ull\uLL\ULL 后缀	unsigned long long

(4) char 类型: 字符、小整数

①char 只有 8 位, 常用于处理字符

▲cout 将 char 类型数据打印为字符, int 类型打印为整数。

例: char ch;cin>>ch;此时 cout<<++ch;打印字符 (++ch 的值为 char), 而 ch+1 为整数 cout.put(ch);打印字符 ch

▲必记 ASCII 码: "a" 为 97, "A" 为 65, " " (空格) 为 32

②转义序列:

表 3.2 C++转义序列的编码

字符名称	ASCII 符号	C++代码	十进制 ASCII 码	十六进制 ASCII 码
换行符	NL (LF)	\n	10	0xA
水平制表符	HT	\t	9	0x9
垂直制表符	VT	\v	11	0xB
退格	BS	\b	8	0x8
回车	CR	\r	13	0xD
振铃	BEL	\a	7	0x7
反斜杠	\	\\	92	0x5C
问号	?	\?	63	0x3F
单引号	'	\'	39	0x27
双引号	"	\"	34	0x22

Table 3.2 C++ Escape Sequence Codes

Character Name	ASCII Symbol	C++ Code	ASCII Decimal Code	ASCII Hex Code
Newline	NL (LF)	\n	10	0xA
Horizontal tab	HT	\t	9	0x9
Vertical tab	VT	\v	11	0xB
Backspace	BS	\b	8	0x8
Carriage return	CR	\r	13	0xD
Alert	BEL	\a	7	0x7
Backslash	\	\\	92	0x5C
Question mark	?	\?	63	0x3F
Single quote	'	\'	39	0x27
Double quote	"	\"	34	0x22

基于八进制和十六进制的转义序列: 如 ' \032' 或 ' \0x1a' (Ctrl+Z)

*通用字符名: 独立于任意的键盘, 用于表示特殊字符

以 \u 或 \U 开头, \u 后面为 4 个二进制位, \U 后面为 8 个二进制位, 其参照 ISO 10646 码点可在标识符和字符串中使用通用字符名。

*signed char、unsigned char

char 在默认情况下既不是没有符号, 也不是有符号

*wchar_t(宽字符类型)

wcin 和 wcout 可用于处理 wchar_t 流。前缀 L 指示宽字符常量和宽字符串

***C++11 新增: char16_t 和 char32_t: 都是无符号的, 一个 16 位, 一个 32 位。前缀 u 表示 char16_t 字符常量与字符串, 前缀 U 表示 char32_t 字符常量与字符串。

(5) bool 类型: 变量值为 true(非 0)和 false(0). true 为真, false 为假。

39. ▲const 限定符: 格式:

const 类型 变量名=值;

该限定符指明变量是只读的, 在初始化之后不可修改(声明时就要初始化)

const 值可以用来声明数组值。

40. 浮点数: 带有小数部分的数字。

(1) 表示法: ①常规表示法: 即使小数部分为 0 也视作浮点数

②E 表示法 (类似于科学计数法, E 或 e 都可以视作 E 表示法)



3. 54E6 表示 3.56×10^6 ，6 叫做**指数**，3.54 叫做**尾数**。

(2) 浮点类型: float、double、long double:

有效位是数字中有意义的位，float 至少 32 位，double 至少 48 位，不少于 float，long double 为 80~96~128 位。指数范围至少为 -37~37。

在 cfloat 中有。如下:

```
// the following are the minimum number of significant
digits
#define DBL_DIG 15 // double
#define FLT_DIG 6 // float
#define LDBL_DIG 18 // long double
// the following are the number of bits used to represent the mantissa
#define DBL_MANT_DIG 53
#define FLT_MANT_DIG 24
#define LDBL_MANT_DIG 64
// the following are the maximum and minimum exponent values
#define DBL_MAX_EXP +308
#define FLT_MAX_EXP +38
#define LDBL_MAX_EXP +4932
#define DBL_MIN_EXP -307
#define FLT_MIN_EXP -37
#define LDBL_MIN_EXP -4931
```

cout.setf(ios_base::fixed, ios_base::floatfield); 防止程序把较大的值切换为 E 表示法。

通常 cout 会删除小数结尾的 0。

(3) **浮点常量**: 默认为 double 类型, f 或 F 后缀表示 float 浮点常量, l 或 L 后缀表示 long double 浮点常量。

→ 整数和浮点数统称**算术类型**

四、算数运算

41. **算术运算符** 二元: 加法、减法、乘法、除法、取模 (取余) 一元: 正号、负号 (相反数)

二元: + 运算符执行加法操作, - 运算符执行减法操作, * 运算符执行乘法操作, / 运算符执行除法操作, % 运算符执行取模操作。一元: + 为正号, - 为负号

• ▲ 对于 / 运算符, 若两个操作数都为整数, 则结果为整数 (丢弃小数位), 如: $5/2$ 为 2; 若有一个操作数为浮点数, 则结果也是浮点数。如 $5/2.0$ 为 2.5, $5.0/2$ 为 2.5, $5.0/2.0$ 为 2.5。

• ▲ 对于 % 运算符, 两个操作数必须为整数, 如果其中一个是负数, 则结果的符号应满足如下规则:
(a/b)*b+a%b=a。

• ▲ **运算符的优先级**: 改变 **优先级**: 加上 **括号**, **括号先算**; * / % 最优先, + - 其次, 同级按先后顺序运算。在 C++ 中, **操作数的结合性** 是 **从左到右** 表示如果两个同级运算符被同时应用到一个操作数上时, 首先应用左侧的运算符, **从右到左** 则首先应用右边的运算符。

五、类型转换初步

▲ 42. 类型转换:

* 自动转换:

(i) 将一种算术类型的值赋给另一种算术类型的变量

(ii) 表达式中包含不同的类型

(iii) 将参数传递给函数

* 转换会出现的问题: 精度降低、小数部分丢失, 超出目标类型取值范围, 只复制右边的字节。

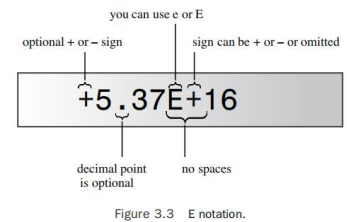
*** C++11: 以 {} 的方式初始化时进行的转换:

使用大括号的初始化叫做 **列表初始化**。列表初始化不允许进行 **缩窄**, 即变量的类型无法表示赋给它的值。

• 表达式的转换:

C++ 将 bool、char、unsigned char、signed char、short 转换为 int (true 为 1, false 为 0) 叫 **整型提升**。在运算涉及两种类型时, 较小的类型被转换为较大的类型。

• ▲ **强制类型转换**: (类型)表达式 或 类型(表达式)



static_cast<类型>(值) 较为严格的转换。

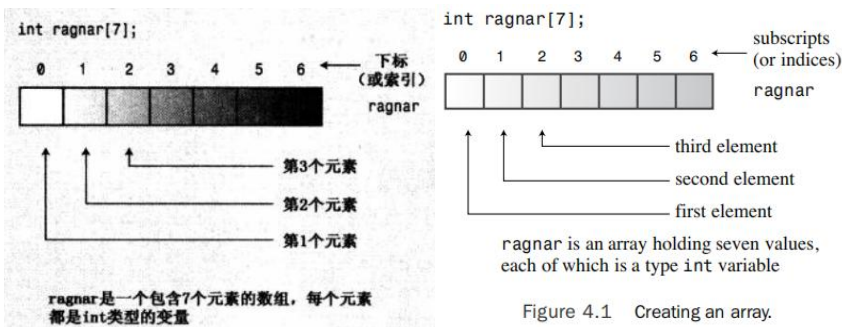
C++11 的 auto 声明: auto 变量名=值; 编译器会自动将变量的类型设置为与变量的初始值相同类型的类型。此为 **自动推断类型**。

六、复合类型

▲43. **数组**: 能存储 **多个同类型** 的值的数据结构。(类似于数学中的集合)

(1) 声明: "**元素类型 数组名[数组长度];**" 其中数组长度为 **整型常数** 或 **const 值** 或 **常量表达式**。

(2) 使用元素: "**数组名[下标(索引)]**" C++ 的数组 **下标从 0 开始**。该形式可以当作单个变量使用。



(3) ▲▲sizeof 运算符:

计算类型或变量或数值占用的字节数。格式:

sizeof 类型名或变量或数值

如: sizeof int 或 sizeof(int)

(4) 将 sizeof 用于数组名时, 得到整个数组的占用字节 (以下称为长度), 用于元素时, 返回元素的长度。

(5) 数组初始化: 只有 **定义时可以初始化**, 此后不可使用, **不能将一个数组赋给另一个数组**。

• 格式: **类型 数组名[数组长度]={初始化列表};** 初始化列表中的每个值以逗号分隔, 并按照列表的顺序分别对每个元素初始化。如:

int arr[4]={1,2,3,4}; 其中 a[0] 初始化为 1, a[1] 初始化为 2, 依此类推。

• **可以只初始化前几个元素, 编译器将把其他元素设为 0;**

• 把数组的元素都初始化为 0: 类型 数组名[数组长度]={0}; 此时显式把第一个元素初始化为 0, 其他元素自动设为 0。

• **如果初始化时方括号为空, 编译器自动计算元素个数**, 如: short arr[]={1,2};

***C++11 数组初始化方法: 可将列表初始化用于任何类型,

- 列表初始化可省略等号
- 可不在大括号里包含任何内容

▲44. 字符串

(1) 定义: 字符串是存储在内存的连续字节的一系列字符。

(2) C 风格字符串 (C-style string): 来自 C 语言

- 可以将字符串存储在 char 数组里
- C 风格字符串 **以空字符结尾**, 写作 **\0, ASCII 码为 0**。例:

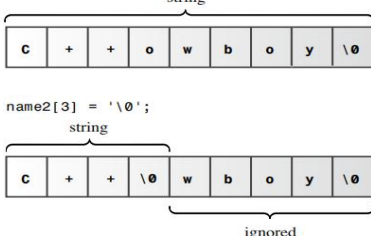
char my_english_name[]={ 'H' , 'A' , 'I' , 'R' , 'S' , 'B' , 'O' , 'Y' , '\0' };

带有空字符的字符数组才是字符串

- 许多处理字符串的函数都以空字符为结束的标志。
- 初始化字符数组: 可以使用字符串, 即: **字符类型 数组名[]=字符串常量;** **字符串常量**用引号引起, 也叫**字符串字面值**。

如: char fruit[]="apple"; 或 char fruit[6]="apple";

```
const int ArSize = 15;
char name2[ArSize] = "C++owboy";
```



▲: 注意 'S' 是字符, 等价于 ASCII 码的 83, "S" 是字符串, 它的值是它所在的地址。

• 拼接字符串常量: 两个字符串常量由空白分开, 其将自动拼成一个。如: "114" "514" 会自动合并为一个字符串。第二个字符串的首字符与第一个字符串的最后一个字符将会直接连在一起。

• 使用字符串:

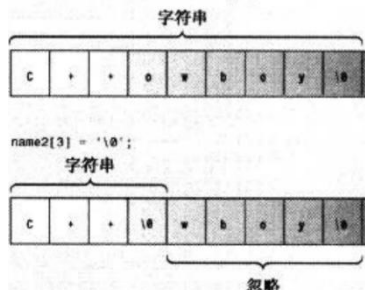
计算字符串的长度: cstring 中的 strlen() 用法: 接受一个字符串 (不

一定是字符数组的长度，是数组中的字符串的字符数)，计算可见的字符，**不包括空字符**。

• 字符串的输入：**cin>>字符数组名**；

cin 使用空白来确定字符串的结束位置。所以 cin 会一次只读取一个单词。输入多个单词会缓存在输入队列里，然后逐一被读取，也无法防止单词过长的问题。

```
const int ArSize = 15;
char name2[ArSize] = "C++owboy";
```



▲每次读取一行的输入：都读取一行输入，直到到达换行符。

① cin.getline(字符数组名, 字符数 (字符数组长度))

本函数丢弃换行符，用空值字符替代，接受两个参数，此处的字符数包括了空值字符，若参数为 20，则最多读取 19 个字符。在读取指定数目字符或遇到换行符时停止读取。

②. cin.get(字符数组名, 字符数 (字符数组长度))

将**换行符**留在输入队列中，其他的与 cin.getline() 相似

③ cin.get() 和 cin.getline() 都返回 cin 对象的引用，便可再次调用其他方法。当 get() 读取空行时设置**失效位 (failbit)**，如果输入字符数

较多，余下的字符留在输入队列中，getline() 设置失效位。

• 混合输入面向行的数字和字符串：在输入数字后，换行符被保留，需要自行丢弃。

常规的 cin 输入返回 cin 对象的引用。如 cin>>year; 返回 istream& 类型 (即 cin 对象的引用)。

(3) string 的初识

• 头文件: string

• 名称空间: std

• 初始化的方式与字符数组相同，允许使用列表初始化 (C++11)。

• 赋值、拼接和附加：

① 可以直接把 **string 对象赋给另一个 string 对象**。

② 可使用 + 运算符把两个 string 对象相加。可以把字符 (串) 与 string 对象合并。即：

string+string, string+char*, char*+string, char+string, string+char 的组合。只要保证**其中有一个操作数为 string 对象即可**。

• C 风格字符串的操作：头文件 cstring 中的函数，用法如下：

strcpy(目标, 源字符串) 复制字符串，

strcat(目标, 源字符串) 附加一个字符串到另一个字符串末尾。

strncat()、strncpy() 前两个参数，第三个参数还要指出最大复制多少字符，更安全

• .size() 和 .length() 表示对象的**长度**，没有被初始化时，长度为 0

• string 的 I/O：可使用 cin 和 cout 来输入输出 string 对象。

输入 string: getline(istream 对象, string 对象); 在使用该函数时，可以这么使用：

getline(cin, string 对象);

(4) 原始字符串 (C++11)

在原始字符串中，字符串表示的就是自己，\n 在原始字符串中不是换行符，可在字符串中使用”。原始字符串以 ” (和) ” 来作为界定，并使用前缀 R。也可以在 ” 和 (之间添加其他字符，结尾的) 和 “ 也是如此。前缀可以混合使用。

▲▲45. 结构体

• 声明结构体：

struct 结构体所代表的类型名

```
{
    结构体成员 (用分号分隔)
};
```

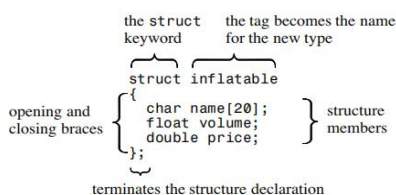
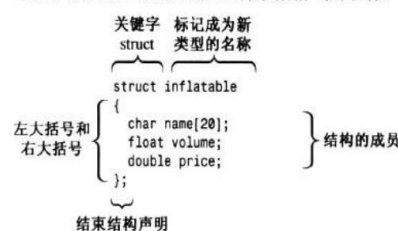


Figure 4.6 Parts of a structure description.



对于结构来说，声明结构体相当于创建自己的类型。所以可以创建结构变量。

• 声明结构变量：类型名称 变量名={初始化列表}; 或 类型名称 变量名;

在声明结构变量时不用像 C 语言一样去在类型名前加上 struct 关键字。

• 访问成员：使用成员运算符. 访问成员。即 结构变量名.成员

该形式可赋值或修改。

▲在 main() 外声明结构(外部声明)有所有的函数共享，内部声明只能被内部的函数使用。

• 初始化方式：

类型名 结构变量名=

```
{
    成员 1 的值,
    成员 2 的值,
    .....
    成员 n 的值
};
```

• C++11 结构初始化：支持列表初始化器，可省略=，空的括号把每个成员设为 0。

• 其他：▲**可以将结构变量赋给同一类型的结构变量**，这样每个成员都一样，叫**成员赋值**。可以同时完成定义结构与变量的工作：

还可以声明没有名称的结构，方法是省略名称，同时定义一个结构类型和一个这种类型的变量。

结构还可以有成员函数。

***结构位字段**：C++ 允许指定占用特定位数的结构成员。

```
struct toggle_register
{
    unsigned int SN : 4;    // 4 bits for SN value
    unsigned int : 4;      // 4 bits unused
    bool goodIn : 1;       // valid input (1 bit)
    bool goodToggle : 1;   // successful toggling
};
```

~46. **共用体（联合）**

• 能够存储不同的数据类型，但只能**同时存储其中的一种类型**，句法与结构类似。

声明格式：

union 联合类型名

```
{
    成员
}
```

共用体的长度为其最大成员的值。与结构一样，也可以有**匿名共用体**。在上面的声明中，id_num 和 id_char 被视为 prize 的两个成员。

▲47. **枚举**

• 格式 **enum 枚举类型名{符号常量（枚举量）};**

如：enum spectrum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};

枚举量的值**从 0 开始**，依次递增。

• 枚举变量的声明：**枚举类型名 枚举变量;**

在不进行强制类型转换时，只能将定义枚举时的枚举量赋给这种枚举类型的变量。

枚举只定义了赋值运算符。可以提升为 int。

• 设定枚举量的值：使用赋值运算符：enum bits{one = 1, two = 2, four = 4, eight = 8};

• 取值范围：比最大枚举值大且相近的 2 的幂减去 1，否则采用与寻找上限相同的方法并加上负号。

▲▲▲48. **指针**

• 概念：存储**地址**的变量或常量，&为**取地址符**，用法：**&变量名**

• 声明：**指向的类型* 指针名=地址;** 或 **指向的类型* 指针名;**

• **解除引用**：取得指针所指向的值，用法 ***地址** 其效果相当于使用原变量。

• **空指针**：值为 0 的指针，标准库中有符号常量 NULL 来表示（C++11 用关键字 **nullptr** 表示）

• 显示地址时，cout 以十六进制表示法显示。提醒：int *m 强调 *m 为 int 变量，int* m 强调 m 为 int* 变量。（以后用 int* 表示指向 int 的指针）int* p1, p2; 会创建一个指针和一个变量，指针变量的长度一致。

警告：在解除引用指针前，要先初始化指针!!!

• ▲**动态分配内存**

①. 方法: **指针=new 要分配的类型;**

new 运算符会根据要分配的类型寻找内存, 并把内存地址返回给指针。(在老的实现里, 内存不足会返回空指针, 新的实现可能引发**异常**)

释放内存: delete 指针;

注意: 一定要把 new 和 delete 配对使用!!!! 否则会发生**内存泄露**。delete 释放指针指向的内存, 不删除指针。不能尝试释放已经释放的内存块, 这是未定义行为。

内存泄露: 指因内存管理不当导致内存并未回收且无法被利用。

只能用 delete 来释放 new 分配的内存, **对于空指针运用 delete 是安全的。**

②. 动态数组

(i) 在编译时给数组分配内存叫**静态联编 (static binding)**

(ii) 在程序运行时选择数组的长度, 叫**动态联编 (dynamic binding)**, 创建的数组叫**动态数组**

(iii) 创建方法: **指针=new 要分配的类型[元素类型];**

返回首元素的地址

释放: **delete[] 指针;**

注意: new[] 与 delete[] 配对使用!!!!

可以把指针当作数组名, 如 *arr 等价于 arr[0]。依此类推。

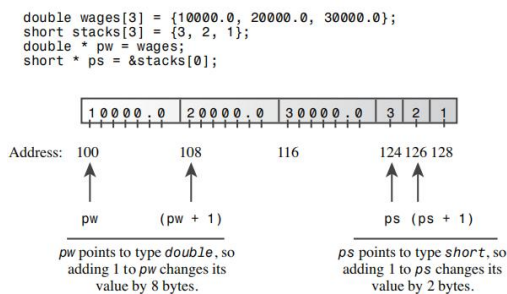
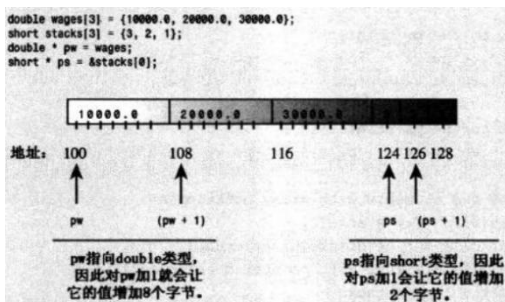
C 与 C++ 的数组与指针基本等价, 其内部使用指针访问数组

• 指针、数组、指针算术

将指针变量加一后, 增加的量相当于它指向的类型的字节数。如 `_Type*` 的变量加一, 指向地址向后偏移 `sizeof(_Type)`。此处的 `_Type` 表示任意类型。

C 和 C++ 将数组名解释为地址。 数组名的地址为数组首元素的地址。

▲ 如果 a 是数组, N 是整数, 则 `*(a+N)` 等价于 `a[N]`, 所以 `*a` 等价于 `a[0]`



▲ 对数组名使用 `sizeof` 得到数组的大小, 而对指针使用 `sizeof` 则为指针的大小, 此时数组名不被视为地址。

▲▲ 易错点: 对数组名取地址时, 数组名不会解释为地址, 而是得到整个数组的地址, 如下:

`short tell[10]; // tell 和 &tell 的指向地址一致。`

此时 `tell` (或说 `&tell[0]`) 是两字节内存块的地址, 对其加一后指针向后偏移两字节, 而 `&tell` 是一块 20 个字节的内存块, 在加一后该指针向后偏移 20 字节。即 `tell` 是 `short&`。而 `&tell` 是 `short(*)[10]`。可以这样声明: `short (*pas)[10]=&tell;` 省略括号 `pas` 会先与 `[10]` 结合, 导致 `pas` 是一个指针数组。(可以理解为 `*pas` 是一个数组, 有括号先算括号)

指针算术还可以获得两个指针的差, 获得一个整数, 是中间间隔的元素个数。

• 当给 `cout` 提供一个字符指针时, `cout` 会从所给地址上的字符开始打印, 直到遇到空字符。

字符数组和字符串常量都是一个地址, 指向第一个字符。

• `strcpy(目标, 源字符串)` 复制字符串。

• `new[]` 的括号中可以使用变量

• 指向结构变量和对象的指针、使用 `new` 创建动态结构

箭头成员运算符: 访问指针里的对象的成员, 用法:

指针->成员 等价于 `*(指针).成员`

- 存储与堆栈

(i) **自动存储**: 在函数内部定义的常规变量使用 **自动存储空间**, 被称为 **自动变量**。他们在所属的函数被调用时自动产生, 在函数结束时消失。

(ii) **静态存储**: 在整个程序执行的期间都存在的存储方式, 创建 **静态变量** 的方式: 在函数外面定义它; 在函数内部使用关键字 static。

(iii) **动态存储**: new 和 delete 管理了内存池, 叫 **自由存储空间 (FREE STORE)** 或 **堆 (HEAP)**, 与静态变量和其他变量分开。

- 类型结合: 创建指针数组 `Type_name* ptr_arr[LENGTH]`

49. 标准模板类初始

array、vector

- 名称空间: std
- 头文件名: array、vector
- 声明的格式:

`std::vector<元素类型> 名称(元素个数(可为变量));`

`std::array<类型, 元素个数(不可变)> 名称;`

- 在 C++11 中, 可使用列表初始化器初始化他们, 可将一个对象赋给同类型对象
- 成员函数: `.at(下标)` 可以在运行期间捕获非法索引, 时间更长。

第一部分总结

1. C++的特性: 是一门面向对象的编译型编程语言, 支持过程化程序设计、面向对象程序设计和泛型程序设计。

2. C++语言的基本模块是函数

3. sizeof 运算符返回类型或变量或数值占用的字节数

4. `cout.setf(ios_base::fixed, ios_base::floatfield);` 防止程序把较大的值切换为 E 表示法。

通常 cout 会删除小数结尾的 0.

5. 字符串的输入输出

方法	说明
<code>std::cin>>字符数组;</code>	<u>使用空白来确定字符串的结束位置</u> , 一次只读取一个单词。多余的在输入队列中, 然后被逐一读取
<code>cin.getline(字符数组名, 字符数(字符数组长度))</code>	读取一行输入, 直到到达换行符。丢弃换行符, 用空值字符替代。在读取指定数目字符或遇到换行符时停止读取。 如果输入字符数较多, 余下的字符留在输入队列中, <code>getline()</code> 设置失效位。
<code>cin.get(字符数组名, 字符数(字符数组长度))</code>	读取一行输入, 直到到达换行符。将 换行符 留在输入队列中。 读取空行时设置 失效位 (failbit)

`cin.get()` 和 `cin.getline()` 都返回 cin 对象的引用, 便可再次调用其他方法。

6. 混合输入面向行的数字和字符串: 在输入数字后, 换行符被保留, 需要自行丢弃。

7. 常规的 cin 输入返回 cin 对象的引用。

8. string

- 头文件: string
- 名称空间: std
- 初始化的方式与字符数组相同, 允许使用列表初始化 (C++11)。
- 赋值、拼接和附加:

方法	说明
<code>string::operator=()</code>	把一个 string 对象赋给另一个 string 对象。
<code>string::operator+()</code>	两个 string 对象相加, 字符(串)与 string 对象合并。 其中有一个操作数为 string 对象

string::size() string::length()	返回长度。没有被初始化时，长度为 0
------------------------------------	--------------------

- string 的 I/O: 输入 string: `getline(istream 对象, string 对象);`

9. C 风格字符串的操作

- 头文件 `cstring` (来自 C)
- 常用方法

`strcpy(目标, 源字符串)` 复制字符串,

`strcat(目标, 源字符串)` 附加一个字符串到另一个字符串末尾。

`strncat()`、`strncpy()` 前两个参数同上, 第三个参数指出最大复制多少字符

第二部分 复合语句: 分支与循环

一、for 循环

- 组成

`for(初始化语句; 测试表达式; 更新表达式)`

语句

C++ 把整个 `for` 看作一条语句, `for` 后可以跟上一条语句或用花括号扩起的代码块
如:

```
for(i=0; i<=5; i++) cout<<i<<endl;
```

```
for(i=0; i<=5; i++) {cout<<" i=" ; cout<<i<<endl;}
```

- `for` 循环的执行过程

- (1) 设置循环变量的初始值
- (2) 进行测试, 检测循环是否应当继续执行
- (3) 执行循环语句
- (4) 更新循环变量

- 控制部分后面的语句叫做 **循环体**。

• **测试表达式** 决定了循环体是否被执行。其通常为 **关系表达式**, 如果值为 `true` 则程序将继续执行循环体

- `for` 是 **入口条件循环**。在每轮循环前会计算测试表达式的值

- **更新表达式** 在每轮循环结束时执行。

二、表达式

▲ • **任何值或任何有效值和运算符的组合都是表达式**, 每个表

达式都有值。 **赋值表达式** 的值定义为左侧成员的值。如 `m=20` 值为 20. 所以可以把几个等于号连用, 如 `a=b=c=0`

- `<` 和 `>` 的优先级比 `<<` 小, 在使用 `cout` 时注意

- 关系运算符的意义: `<` 小于, `>` 大于, `==` 等于, `>=` 大于等于, `<=` 小于等于, `!=` 不等于

关系表达式为的值为 `bool` 类型

• `cout.setf(ios_base::boolalpha);` 用于设置 `cout` 在显示 `bool` 值时使用 `true` 或 `false`, 而不是 0 或 1.

• 为判定表达式 `x=100` 的值, C++ 进行了赋值操作。当判定表达式的值的操作改变了内存中数据的值时, 则该表达式有 **副作用 (side effect)**

- 任何表达式加上分号成为语句, 反过来就不对了。表达式加上分号成为 **表达式语句**

• **返回语句、声明语句和 `for`、`while`、`do-while` 等语句都不是表达式语句, 其没有值。**

如 `int total` 其不是表达式, 没有值。 **判断语句是否为表达式看是否有值**

• 可以在 `for` 循环的初始化部分声明变量。该变量会在离开 `for` 后消失。`for` 循环的第一个语句可以是 **表达式语句**, 也可以是 **声明语句**。

- 更新表达式的更新的量有时也叫做步长

▲▲ • 递增 (`++`) 和递减 (`--`) 运算符: 每次增减 1

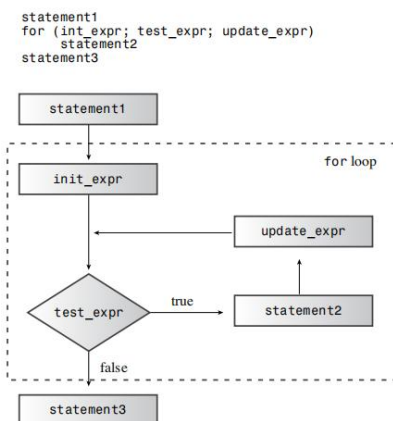


Figure 5.1 The design of `for` loops.

用法：前缀 ++变量 --变量 先修改再使用变量的值（此时已修改）
 后缀 变量-- 变量++ 先使用变量的值再修改变量

警告：在编写程序时，请严格遵守标准，否则程序会出现难以预知的错误。不能在一条语句上运用多次递增递减运算符，如 `x=(++n)+(n++)` 会在不同的系统上生成不同的结果。

• 副作用和顺序点

副作用：计算表达式时对某一些东西进行了修改

顺序点：程序执行过程的一个点，任何完整表达式后都有一个顺序点，在进入下一步前将确保对所有副作用都进行了评估。即在执行下一条语句前，赋值、递增、递减运算符执行的所有修改都必须完成。

完整表达式：不是另一个更大的表达式的子表达式

```
y=(4+x++)+(6+x++);
```

以上语句中 `4+x++` 不是完整表达式，其后没有顺序点，所以其所有的修改无法进行保证
 则 `x` 不一定加 1，所以该表达式会产生未定义行为。

Note:在部分计算机教育中，有一个经典问题：`i=1`，则表达式 `n=(i++)+(++i)` 值为多少，这是与上面的问题相似，在上面已经明确指明此会导致不同编译器输出不同的结果。所以无论如何都不能写这样的程序。且在 C++ 和 C 标准中已经明确指出此为未定义行为！！

*对于用户定义的类型，若定义了递增递减运算符，则前缀格式的效率更高

• 递增递减与指针

前缀格式与解除引用的优先级相同，从右向左结合；后缀格式比前缀格式优先级高，从左向右结合。

`++pt` 表示先将 `++` 应用于 `pt` 后再应用 ``，`++*pt` 意味着先取得 `pt` 指向的值，然后递增，`*pt++` 先应用 `++` 再应用 `*`。

```
#include<iostream>
using namespace std;
int main()
{
    cout<<"-----循环结构递增与解除引用测试-----\n";
    int arr[10]={1,2,3,4,5,6,7,8,9,10};
    int *pt=&arr[0];
    cout<<"*pt="<<(*pt)<<endl
         <<"*++pt="<<(*++pt)<<endl
         <<"*pt++="<<(*pt++)<<endl;
    return 0;
}
```

-----循环结构递增与解除引用测试-----

```
*pt=1
*++pt=2
*pt++=2
```

• **组合赋值运算符：**L 为左操作数，R 为右操作数

<code>+=</code>	将 L+R 赋给 L	<code>-=</code>	将 L-R 赋给 L	<code>*=</code>	将 L*R 赋给 L
<code>/=</code>	将 L/R 赋给 L	<code>%=</code>	将 L%R 赋给 L		

三、复合语句（语句块）

复合语句（代码块）由一对花括号和它包含的语句组成，被视作一条语句。

在代码块中定义的变量只在其中有效。若外部定义了同名变量，则先隐藏，在代码块执行完毕后代码块中的变量消失，外部的变量重新出现。

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



```
int main()
{
    int x=100;
    cout<<"x_out="<<x<<endl;
    {
        int x=114514;
        cout<<"x_in="<<x<<endl;
    }
    cout<<"x_out="<<x<<endl;
    return 0;
}
```

```
x_out=100
x_in=114514
x_out=100
```

▲ • 逗号运算符：表达式的值为右侧（第二部分）的值。

```
a=(114,514);
```

该表达式值为 514.

• bool 值提升为 int: 只有 0(false) 和 1(true) 两种情况

▲▲ • C 风格字符的比较

注意：用引号引起的字符串常量也是其地址，`word=="mate"` 比较的是它们地址是否一致

▲▲▲▲▲ cstring 文件中的 `strcmp()` 函数：接受两个 `char*`

如果两个字符串相同，则返回 0

以下为对其的解释的 2 个版本：

①如果第一个字符串按字母排序排在第二个字符串后面，`strcmp()` 返回正数值。（也说按系统排列顺序）在 ASCII 码中，大写编码都比小写小，大写字母排在前面。

②`strcmp()` 函数是根据 ASCII 码的值来依次比较 `str1` 和 `str2` 的每一个字符直到出现不同的字符，或者到达字符串末尾；`strcmp()` 函数首先将 `s1` 字符串的第一个字符值减去 `s2` 第一个字符，若差值为零则继续比较下去；若差值不为零，则返回差值。

如：“Zoo”比“aviary”排在前面，“Physics”与“physics”不一样

检测相等或排列顺序

以下表达式为 true 的条件：

①`str1` 与 `str2` 相等 `strcmp(str1,str2)==0`

②`str1` 与 `str2` 不相等 `strcmp(str1,str2)!=0`
`strcmp(str1,str2)`

③`str1` 在 `str2` 的前面（此时 `str1<str2`） `strcmp(str1,str2)<0`
//`str1` 中的字符比 `str2` 的字符数值小

④`str1` 在 `str2` 的后面（此时 `str1>str2`） `strcmp(str1,str2)>0`
//`str1` 中的字符比 `str2` 的字符数值小

```
#include<iostream>
#include<cstring>
using namespace std;
int main()
{
    char a[]="Everything";
    char b[]="Every";
    cout.setf(ios_base::boolalpha);
```

```

    cout<<"strcmp(a,b)==0  "<<(strcmp(a,b)==0)<<endl;
    cout<<"strcmp(a,b)>0   "<<(strcmp(a,b)>0)<<endl;
    char c[]="Anything";
    char d[]="anything";
    cout<<"strcmp(c,d)==0  "<<(strcmp(c,d)==0)<<endl;
    cout<<"strcmp(c,d)>0   "<<(strcmp(c,d)>0)<<endl;
    cout<<"strcmp(a,c)>0   "<<(strcmp(a,c)>0)<<endl;
    return 0;
}

```

```

strcmp(a,b)==0  false
strcmp(a,b)>0   true
strcmp(c,d)==0  false
strcmp(c,d)>0   false
strcmp(a,c)>0   true

```

- 比较 string 字符串：至少有一个操作数为 string

string 重载了所有关系运算符

四、while 循环

- while 循环只有测试条件和循环体：

while (测试条件)

循环体

首先，出现先计算测试条件，如果为 true 执行循环体。执行完循环体后，程序返回测试条件，若条件为 true，继续执行，直到为 false 退出

- while 为 **入口条件循环**

• for 循环可以省略括号中的表达式，只需要两个分号，此时括号中有空表达式（语句）。省略测试表达式测试结果为 true。

即 `for(;;) body` 和 `while(true) body` 都是死循环。

- 实用实例 编写延时循环

头文件：ctime (c 语言中为 time.h)

// waiting.cpp -- using clock() in a time-delay loop

```
#include <iostream>
```

```
#include <ctime> // describes clock() function, clock_t type
```

```
int main()
```

```

{
    using namespace std;
    cout << "Enter the delay time, in seconds: ";
    float secs;
    cin >> secs;
    clock_t delay = secs * CLOCKS_PER_SEC; // convert to clock ticks
    cout << "starting\n";
    clock_t start = clock();
    while (clock() - start < delay )           // wait until time elapses
        ;                                     // note the semicolon
    cout << "done \n";
    // cin.get();
    // cin.get();
    return 0;
}

```

```
}
Enter the delay time, in seconds: 2
starting
Done
```

Process exited after 3.258 seconds with return value 0, 3564 KB mem used.
Press ANY key to exit...

Note: 上面的斜体加横线是输入的字符, 较浅的区块为调试器输出的信息。输入用了 1 秒多

clock() 函数返回程序开始执行后所用的**系统时间**, 返回 clock_t 类型

CLOCKS_PER_SEC 常量是**每秒钟包含的系统时间单位数**, 将系统时间除以该值得到秒数。

即 $\text{系统时间} / \text{每秒钟系统时间单位数 (CLOCKS_PER_SEC)} = \text{秒数}$

所以 $\text{秒数} * \text{CLOCKS_PER_SEC} = \text{系统时间}$, 为以上的实现

类型别名: 关键字 typedef: 格式: `typedef 类型名 类型的别名;`

此时可以用声明的别名来替代原来的类型名使用。如 `typedef char* B_pointer;`

使用#define 时, 若声明了一系列变量, 则在文本替换时会出现一些问题:

```
#define QWORDptr long long*
```

```
QWORDptr a,b;
```

等价于

```
long long* a,b;
```

此时 a 为 long long*, b 为 long long

五、do-while 循环

do-while 循环是**出口条件循环**。循环体先执行再判定测试表达式。当测试表达式为假时, 退出循环。语法:

```
do
```

循环体

```
while(测试表达式);
```

注意不要漏了语句后的分号

六、基于范围的 for 循环 (C++11)

示例:

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};
```

```
for (double x : prices)
```

```
cout << x << std::endl;
```

该循环显示数组中的每一个值 (从 prices[0] 到 prices[4])

能够结合使用初始化列表

```
for (int x : {3, 5, 2, 8, 6})
```

```
    cout << x << " ";
```

```
cout << '\n';
```

七、循环和文本输入

1. 使用原始的 cin

(1) 传统停止输入方法: 使用哨兵字符, 将其作为停止标记。

(2) cin 在读取 char 值时忽略空格和换行符。输入的空格不能回显

(3) cin 的输入被**缓冲**, 按下回车键后才会发送内容。

```
#include<iostream>
using namespace std;
int main()
{
```

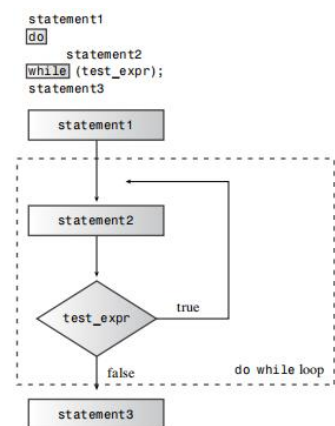


Figure 5.4 The structure of do while loops.

```

char ch=0;
int total=0;
while(ch!='$')
{
    cin>>ch;
    total++;
    cout<<ch;
}
return 0;
}

```

What can I do for you?\$

WhatcanIdoforyou?\$

2. 补救

(1) `cin.get(ch)`; 读取输入中的下一个字符，包括空格、制表符、换行符。其返回 `istream&` 类型的 `cin` (是 `cin` 对象的引用，可以继续调用 `cin` 的方法)。

注: `cin` 所属的 `istream` 类包含了成员 `get(ch)`，其子类都有这个方法。

(2) C++ 中可以有多个同名函数，是 **函数重载**

(3) **文件尾条件**

① 制造文件尾: `Ctrl+Z` (^Z)

② C++ 在检测到 **EOF** (END OF FILE, 即文件尾)，`cin` 将两位 (`eofbit` 和 `failbit`) 设为 1。

`cin.eof()` 检测到 `eofbit` 被设为 1，`cin.eof()` 返回 `true`，否则返回 `false`。

`cin.fail()` 如果 `eofbit` 或 `failbit` 设为 1，则 `fail()` 成员返回 `true`

两个函数报告最近读取的结果

`cin.clear()` 重置输入

③ `istream` 对象在需要 `bool` 值的地方可以转为 `bool` 值 (调用转换函数)。若最后一次读取成功，返回 `true`，失败返回 `false`。其比两个成员函数更通用。

④ 不接受任何参数的 `cin.get()` 成员函数返回输入中的下一个字符。该函数到达 EOF 时返回符号常量 EOF，其值被定义为 -1。

⑤ `cout.put(ch)` 显示一个字符

八、嵌套循环和二维数组

1. 二维数组

- 定义: 每个元素本身就是数组的数组
- 声明方法: 元素类型 二维数组名[元素个数][元素个数];

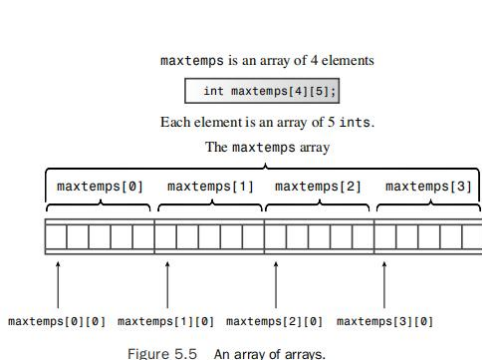


Figure 5.5 An array of arrays.

```
int maxtemps[4][5];
```

The maxtemps array viewed as a table:

		0	1	2	3	4
maxtemps[0]	0	maxtemps[0][0]	maxtemps[0][1]	maxtemps[0][2]	maxtemps[0][3]	maxtemps[0][4]
maxtemps[1]	1	maxtemps[1][0]	maxtemps[1][1]	maxtemps[1][2]	maxtemps[1][3]	maxtemps[1][4]
maxtemps[2]	2	maxtemps[2][0]	maxtemps[2][1]	maxtemps[2][2]	maxtemps[2][3]	maxtemps[2][4]
maxtemps[3]	3	maxtemps[3][0]	maxtemps[3][1]	maxtemps[3][2]	maxtemps[3][3]	maxtemps[3][4]

Figure 5.6 Accessing array elements with subscripts.

如: `int example[2][8]`; 表示 `example` 包含两个元素，是两个数组，每个数组中又有 8 个 `int`

```

#include<iostream>
using namespace std;

```



```

int main()
{
    int marks[5][5]=
    {
        {100,99,100,90,99},
        {120,119,117,101,118},
        {140,135,145,147,150},
        {0,0,1,2,3},
        {28,28,30,35,37}
    };
    for(int i=0;i<5;i++)
    {
        for(int j=0;j<5;j++)
        {
            cout<<marks[i][j]<<" \t";
        }
        cout<<endl;
    }
    return 0;
}

```

100	99	100	90	99
120	119	117	101	118
140	135	145	147	150
0	0	1	2	3
28	28	30	35	37

- 初始化：由一系列逗号分隔的一维数组初始化（用花括号括起）。
- 嵌套 for 循环：算法中常见，有内外层循环。可用于输入、打印高维数组。

示例#1：输入二维数组：

```

for(int i=0;i<5;i++)
    for(int j=0;j<5;j++)
        cin>>arr[i][j];

```

示例#2：计算 $1+(1+2)+(1+2+3)+(1+2+3+4)+\dots+(1+2+3+\dots+20)+5$

```

#include<iostream>
using namespace std;
int main()
{
    int total=0;
    for(int i=1;i<=20;i++)
    {
        for(int j=1;j<=i;j++)
        {
            total+=j;
        }
    }
    total+=5;
    cout<<total;
}

```

```
return 0;
```

```
}
```

输出: 1545_____

九、if 语句、if-else 语句、if-else if-else 结构

1. if 语句

- 语法:

```
if(测试表达式)
```

```
    语句
```

如果测试条件为 true,则程序执行语句,如果测试条件为 false,程序跳过语句。

测试条件会强制转换为 bool 值。

整个 if 语句通常被视为 1 条语句

1. if-else 语句

- 语法:

```
if(测试表达式)
```

```
    语句 1
```

```
else
```

```
    语句 2
```

如果测试条件为 true,程序执行语句 1,如果测试条件为 false 执行语句 2。整个 if-else 语句被视作一条语句。

注:在处理字符时, ch++ 的值仍是 char, ch+1 却是 int

注意:中间如有多条语句,记得使用括号。单独出现 else 会被认为是错误。

2. if-else if-else 结构 (if-else 语句嵌套)

由 if-else 扩展而来。如

```
if (ch == 'A')
    a_grade++; // alternative # 1
else
    if (ch == 'B') // alternative # 2
        b_grade++; // subalternative # 2a
    else
        soso++; // subalternative # 2b
```

如果 ch 不是 'A',则判断 ch 是不是 'B',如

果不是执行最后的 else。

由于 C++ 格式自由,

else 可以直接接 if。

该结构的常用形式和逻辑如下:

```
if(条件 0)
    语句 0
else if(条件 1)
    语句 1
.....
else if(条件 n)
    语句 n
else
    语句 n+1
```

最后一个 else 可以没有。

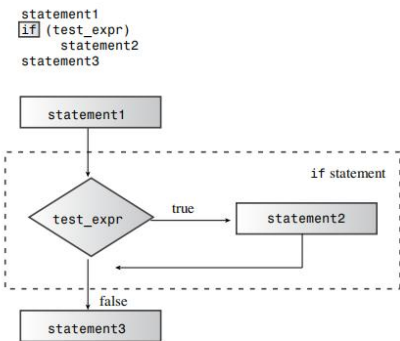


Figure 6.1 The structure of if statements.

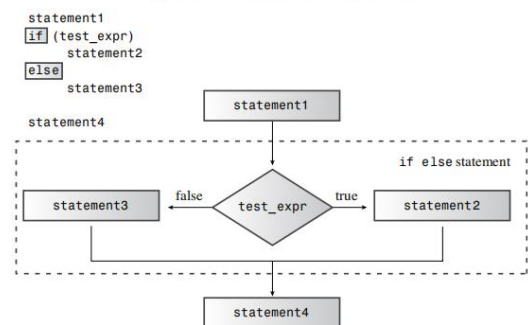
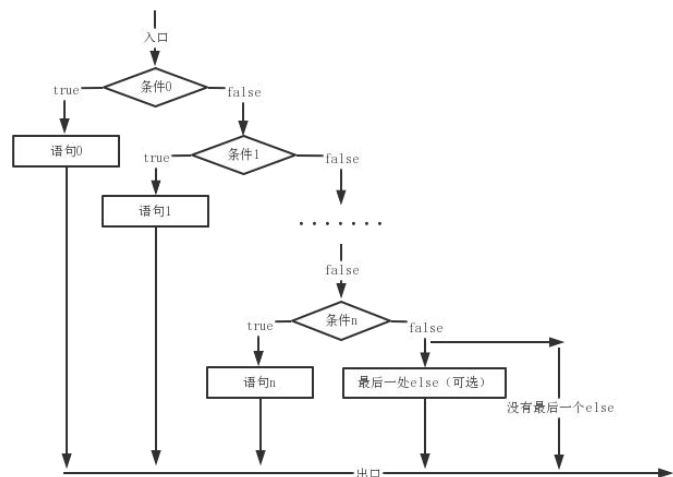


Figure 6.2 The structure of if else statements.



十、▲逻辑表达式

- 三种**逻辑运算符**：或 `or(||)`、且 `and(&&)`、非 `not(!)`

1. `or(||)`

用 `||` 将两个表达式连接在一起时，整个表达式的值为真当且仅当其中有一个表达式为真。

关系运算符的优先级比 `||` 高。

`||` 是一个顺序点。先修改左侧的值再判定右侧的值。（C++11 中说法为左边的子表达式先于右边的子表达式）回顾：分号和逗号也是顺序点。

Table 6.1 The `||` Operator

The Value of <code>expr1 expr2</code>		
	<code>expr1 == true</code>	<code>expr1 == false</code>
<code>expr2 == true</code>	true	true
<code>expr2 == false</code>	true	false

2. `and(&&)`

用 `&&` 将两个表达式连接起来时，仅当原来的两个表达式都为 `true` 得到的表达式才为 `true`。 `&&` 的优先级低于关系运算符，且 `&&` 也是顺序点。首先判定左侧。

Table 6.2 The `&&` Operator

The Value of <code>expr1 && expr2</code>		
	<code>expr1 == true</code>	<code>expr1 == false</code>
<code>expr2 == true</code>	true	false
<code>expr2 == false</code>	false	false

3. 用 `&&` 设置取值范围：例： `1 < x < 10`，在 C++ 中为 `1 < x && x < 10`

注意： `1 < x < 10` 在 C++ 中不是语法错误，但是其等价于 `(1 < x) < 10`，先判断 `1 < x` 并转为 `int`，再判断其是否小于 10。

4. `not(!)`

! 运算符将它后面的表达式的真值取反。即当 `expression` 为 `true`，则 `!expression` 为 `false`，当 `expression` 为 `false`，则 `!expression` 为 `true`。

5. 三种运算符的**运算优先级从小到大排列分别为 !、&&、||**

`or`、`and` 优先级都低于关系运算符，! 的优先级高于任何关系和算术运算符。

note: 这些规定与离散数学中的部分规定相似，逻辑运算符优先级参照了命题联结词。

部分示例：

`x > 5 && x < 10` 判断 `x` 是否在 5~10 之间，`!(x > 5)` 判断是否 `x <= 5`，`!x > 5` 判断 `!x` 是否大于 5。

`age > 30 && age < 45 || weight > 300` 判断 `age > 30 && age < 45` 或者 `weight > 300` 是否成立。

6. 其他表达方式：`&&` 可用 `and` 替换，`||` 可用 `or` 替换，`!` 可用 `not` 替换。`and`、`or`、`not` 都是 C++ 保留字，在 C 语言中要引入 `iso646.h` (C 语言中不是保留字)

十一、字符函数库 CCTYPE

程序示例：右侧是程序输入输出，红框是输入

```
#include<iostream>
#include<cctype>
using namespace std;
int main()
{
    char ch;
    int order=0;
    while(cin.get(ch))
    {
        order++;
        cout<<"char#"<<order<<endl;
        if(isalnum(ch))cout<<"是数字字母"<<endl;
        if(isalpha(ch))cout<<"是字母"<<endl;
        if(isdigit(ch))cout<<"是数字"<<endl;
        if(iscntrl(ch))cout<<"是控制字符"<<endl;
        if(islower(ch))cout<<"小写字母"<<endl;
        if(isupper(ch))cout<<"大写字母"<<endl;
        if(isxdigit(ch))cout<<"是十六进制数字"<<endl;
```

```
Wang 28
char#1
是数字字母
是字母
小写字母
大写字母
char#2
是数字字母
是字母
小写字母
是十六进制数字
char#3
是数字字母
是字母
小写字母
char#4
是数字字母
是字母
小写字母
char#5
是数字字母
是数字
是十六进制数字
char#6
是数字字母
是数字
是十六进制数字
char#7
是数字字母
是数字
是十六进制数字
char#8
是控制字符
```

```

}
return 0;
}

```

表 6.4 cctype 中的字符函数

函数名称	返回值
isalnum()	如果参数是字母数字，即字母或数字，该函数返回 true
isalpha()	如果参数是字母，该函数返回 true
iscntrl()	如果参数是控制字符，该函数返回 true
isdigit()	如果参数是数字（0~9），该函数返回 true
isgraph()	如果参数是除空格之外的打印字符，该函数返回 true
islower()	如果参数是小写字母，该函数返回 true
isprint()	如果参数是打印字符（包括空格），该函数返回 true
ispunct()	如果参数是标点符号，该函数返回 true
isspace()	如果参数是标准空白字符，如空格、进纸、换行符、回车、水平制表符或者垂直制表符，该函数返回 true
isupper()	如果参数是大写字母，该函数返回 true
isxdigit()	如果参数是十六进制数字，即 0~9、a~f 或 A~F，该函数返回 true
tolower()	如果参数是大写字符，则返回其小写，否则返回该参数
toupper()	如果参数是小写字符，则返回其大写，否则返回该参数

Table 6.4 The cctype Character Functions

Function Name	Return Value
isalnum()	This function returns true if the argument is alphanumeric (that is, a letter or a digit).
isalpha()	This function returns true if the argument is alphabetic.
isblank()	This function returns true if the argument is a space or a horizontal tab.
iscntrl()	This function returns true if the argument is a control character.
isdigit()	This function returns true if the argument is a decimal digit (0–9).
isgraph()	This function returns true if the argument is any printing character other than a space.
islower()	This function returns true if the argument is a lowercase letter.
isprint()	This function returns true if the argument is any printing character, including a space.
ispunct()	This function returns true if the argument is a punctuation character.
isspace()	This function returns true if the argument is a standard white-space character (that is, a space, formfeed, newline, carriage return, horizontal tab, vertical tab).
isupper()	This function returns true if the argument is an uppercase letter.

Table 6.4 The cctype Character Functions

Function Name	Return Value
isxdigit()	This function returns true if the argument is a hexadecimal digit character (that is, 0–9, a–f, or A–F).
tolower()	If the argument is an uppercase character, tolower() returns the lowercase version of that character; otherwise, it returns the argument unaltered.
toupper()	If the argument is a lowercase character, toupper() returns the uppercase version of that character; otherwise, it returns the argument unaltered.

十二、?:运算符:?:被称作条件运算符,是唯一需要三个操作数的运算符(三目运算符)
通用格式:

条件? 表达式 1:表达式 2

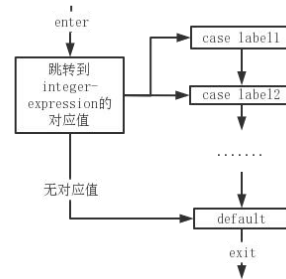
当条件为真时,整个表达式的值为表达式 1;条件为假时,整个表达式的值为表达式 2。

该运算符常嵌套进其他语句,可以用来简化 if else

十三、switch 语句

switch(integer-expression)

```
{
    case label1:statement(s)
    case label2:statement(s)
    ...
    default:statement(s)
}
```



执行 switch 语句时,程序跳转到 integer-expression 的值标记的那一行。

注意:①**integer-expression 必须是一个结果为整数值的表达式。**

②**每个标签 (case 后的就是标签) 必须是 int 或 char 常量 (如 128 或 'W') 或是枚举量。**如果其不与任何标签匹配,跳转到 default 处。

③default 标签是可选的,若没有匹配的标签和 default 标签,程序跳转到 switch 语句的下一条语句。

④case 标签只是行标签, **不是选项之间的界限。**跳转到某一处标签后会执行完下面的所有语句。

程序并不会直接在下一个 case 标签停下⑤使用 break 语句可以结束 switch 语句的执行。

note:程序的跳转只是程序跳到了某一位置的语句,然后继续依次执行语句,并不会无缘无故停下。

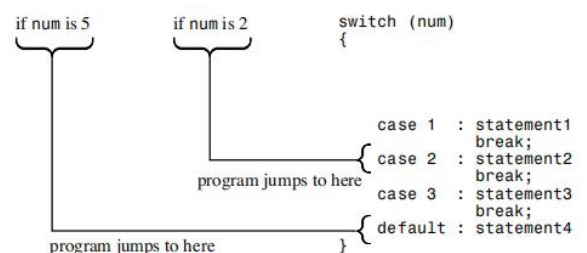
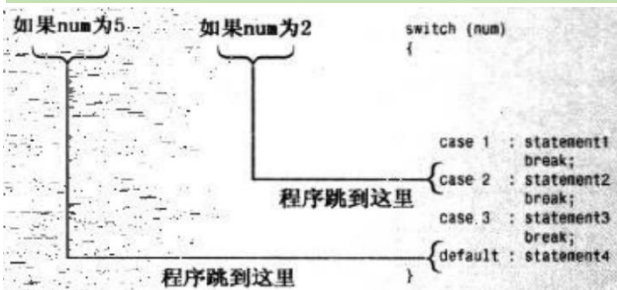


Figure 6.3 The structure of switch statements.

常用用法:①多标签跳转到同一位置 ②常用“菜单”选定格式

switch(integer-expression)

```
{
    case label1a:
    case label1b:
        statement(s)
    case label2a:
    case label2b:
    case .....
        statement(s)
    ....
}
```

switch(integer-expression)

```
{
    case label1:
```

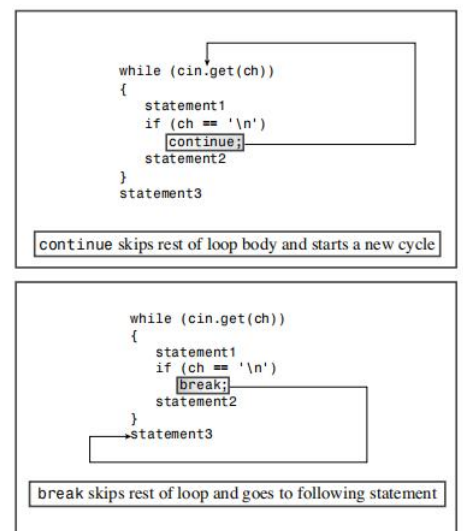


Figure 6.4 The structure of continue and break statements.

```

        statement(s)
        break;
    case label2:
        statement(s)
        break;
    ...
    default:
        statement(s)
        break;
}

```

• 使用匿名枚举（用枚举定义常量）：enum {枚举量 0, 枚举量 1,};

十四、break 和 continue 语句

1. break 语句结束循环

2. continue 语句用于循环当中，跳过循环体余下的代码，开始新一轮循环。

~3. goto 语句：跳转到指定标签。即 goto 标签名;

标签的用法： 标签名: 语句

note: 大部分情况下很少用 goto, 有些人认为这破坏程序的可读性。只有在一次性跳出多重循环时使用。

3. break 和 continue 示例:

// jump.cpp -- using continue and break

```

#include <iostream>
const int ArSize = 80;
int main()
{
    using namespace std;
    char line[ArSize];
    int spaces = 0;
    cout << "Enter a line of text:\n";
    cin.get(line, ArSize);
    cout << "Complete line:\n" << line << endl;
    cout << "Line through first period:\n";
    for (int i = 0; line[i] != '\0'; i++)
    {
        cout << line[i]; // display character
        if (line[i] == '.') // quit if it's a period
        {
            break;
        }
        if (line[i] != ' ') // skip rest of loop
        {
            continue;
        }
        spaces++;
    }
    cout << "\n" << spaces << " spaces\n";
    cout << "Done.\n";
    return 0;
}

```

Enter a line of text:

Let's do lunch today. You can pay!

Complete line:

Let's do lunch today. You can pay!

Line through first period:

Let's do lunch today.

3 spaces

Done.

十五、复合语句应用举例:读取数字的循环

• 输入数字发生类型不匹配时会出现的情况:

- ①变量的值保持不变
- ②不匹配的输入被留在输入队列中
- ③cin 对象中的一个错误标记被设置
- ④对 cin 方法的调用返回 false

十六、简单文件输入输出

1. 使用 cin 输入时, 程序把输入视作一系列**字节**, 每个字节都被解释为**字符编码**。

2. 输入**都是字符数据**, cin 负责将其转换为其他类型, 以达到提供输入的结果。

- ①使用 char 数据: 将输入队列中的第一个字符读取,
- ②使用 int 类型: 不断读取, 直到非数字字符,
- ③使用 double 类型: 不断读取, 直到遇到第一个不属于浮点数的字符
- ④常规 cin 输入 char 数组: 不断读取, 直到空白字符
- ⑤cin.getline() 输入 char 数组: 不断读取, 直到空白字符

3. 输出时, 操作相反。浮点数可能转为 e 表示法

4. 输入一开始为文本, 控制台输入为**文本文件**。

5. 文本文件: 每个字节都存储了一个字符编码的文件

6. **不是所有的文件都是文本**, 数据库、电子表格 (excel) 以数值格式存储数据。

7. 字处理文件 (word) 中除了文本信息, 可能还有其他信息。

8. 控制台 I/O (cmd 是控制台) 仅适用于文本文件。源代码文件属于文本文件。

• 写入文本文件 (类比 cout)

工具	cout	文件输出
头文件	iostream, 定义了一个用于处理输出的 ostream 类。cout 是一个 ostream 对象 (变量)	<ul style="list-style-type: none"> 引入 fstream 需声明 ofstream (由 ostream 派生而来) 对象。方式: ofstream 文件输出流对象名;
名称空间	std	std
使用	使用运算符<<	需要将 ofstream 对象与文件关联起来, 使用的是 open() 方法。使用完文件后使用方法 close() 方法关闭 。可以像 cout 那样使用运算符<<

示例: ofstream fout;

fout<<"31 miss Cai"<<endl<<"pretty "<<"sweet"<<endl;

fout<<28<<"miss W"<<'\\n'<<"fantastic "<<"sing well "<<total<<"boy love her";

关联文件时使用 open() 方法, 该方法接受一个 C 风格字符串作为参数。

boy_which_is_handsome.open("45_PERSONAL.txt");

letter.open("Mr. WuZhehao_to_Miss_Cai.txt");

fout.close();

▲所有可用于 cout 的操作都可用于 ofstream 对象, 包括 setf() 等格式化

在打开文件时, 若文件不存在, 方法 open() 新建文件。打开原有的文件时, 截断该文件, 将其长度变为 0, 丢弃原有的内容, 再输出新的内容。

- 读取文本文件（类比 cin）

工具	cin	文件输出
头文件	iostream, 定义了一个用于处理输出的 istream 类。cin 是一个 istream 对象（变量）	<ul style="list-style-type: none"> • 引入 fstream • 需声明 ifstream（由 istream 派生而来）对象。方式：ifstream 文件输出流对象名；
名称空间	std	std
使用	使用运算符 >>	需要将 ifstream 对象与文件关联起来，使用的是 open() 方法，该方法接受一个 C 风格字符串作为参数。使用完文件后使用方法 close() 方法关闭。可以像 cin 那样使用运算符 >>，可以使用一切 cin 能用的方法，包括 eof()、将其作为测试条件。

示例：ifstream fin; fin >> girlname >> keywords;

检测文件是否成功被打开：使用 is_open() 方法。成功打开返回 true，失败返回 false。

- 函数 exit() 的原型是在头文件 cstdlib 中定义的。exit(返回值) 终止程序。cstdlib 中的 EXIT_FAILURE 表示程序失败的返回值。

- good() 方法也可以检测文件是否打开[包括 eof 和文件是否正常输入]。但是检测的问题没有 is_open() 广。

note: Windows 文本文件每行都以回车字符和换行符结尾，通常会在读取时进行转换。部分文本编辑器不在最后一行末尾加上换行符。

- 程序读取文件不超过 EOF（文件尾）。最后一次读取数据遇到 EOF，方法 eof() 返回 true。最后一次读取数据发生内存类型不匹配的情况，方法 fail() 返回 true（有 EOF 时该方法也返回 true）。最后一次读取出现意外的问题时，方法 bad() 返回 true。good() 方法在没有任何错误时返回 true。

- ★实用方法：用 good() 预防输入意外：在循环前（执行循环测试前）放置输入语句，在下次执行循环测试之前放置另一条输入语句。

```
// standard file-reading loop design
inFile >> value; // get first value
while (inFile.good()) // while input good and not at EOF
{
    // loop body goes here
    inFile >> value; // get next value
}
```

或者如下：

```
while (inFile >> value) // read and test for success
{
    // loop body goes here
    // omit end-of-loop input
}
```

第二部分总结

1. for 循环的格式

for(初始化语句; 测试表达式; 更新表达式)
语句

特别地，for(;;) 用于编写死循环

2. while 循环的格式

while (测试条件)
循环体

特别地, while(true) 或 while(1) 用于编写死循环

3. do-while 循环

do

循环体

while(测试表达式);

注意不要漏了语句后的分号

4. 基于范围的 for 循环(C++11): 示例:

```
for(int x: {3,5,2,8,6}) cout<<x<<" ";
```

5. 表达式的定义: 任何值或任何有效值和运算符的组合, 特点: 每个表达式都有值。

6. 赋值表达式的值定义为左侧成员的值. 应用: 几个等号连用.

7. cout.setf(ios_base::boolalpha); 用于设置 cout 在显示 bool 值时使用 true 或 false, 默认情况下 bool 值显示为数字。

8. 当判定表达式的值的操作改变了内存中数据的值时, 则该表达式有副作用(side effect)

9. 表达式语句的定义: 任何表达式加上分号成为表达式语句, 反过来就不对了。

10. 判断语句是否为表达式看是否有值, 返回语句、声明语句和 for、while、do-while 等语句都不是表达式语句, 因为其没有值。

11. 在 for 循环的初始化部分声明的变量会在离开 for 后消失。for 循环的第一个语句可以是表达式语句, 也可以是声明语句。

12. 顺序点: 程序执行过程的一个点, 任何完整表达式后都有一个顺序点, 在进入下一步前将确保对所有副作用都进行了评估。即在执行下一条语句前, 赋值、递增、递减运算符执行的所有修改都必须完成。

13. 完整表达式: 不是另一个更大的表达式的子表达式

14. 递增递减与指针: 前缀格式与解除引用的优先级相同, 后缀格式比前缀格式优先级高

15. 复合语句(代码块) 由一对花括号和它包含的语句组成, 被视作一条语句。

16. 逗号运算符: 表达式的值为右侧(第二部分)的值。

17. 字符串的比较和 cstring 常用函数的总结:

strcpy(目标, 源字符串) 复制字符串,

strcat(目标, 源字符串) 附加一个字符串到另一个字符串末尾。

strncat()、strncpy() 前两个参数同上, 第三个参数指出最大复制多少字符

检测相等或排列顺序: strcmp()

如果两个字符串相同, 则返回 0

如果第一个字符串按字母排序排在第二个字符串后面, strcmp() 返回正数值。

以下表达式为 true 的条件: (可以从按字母排序、按 ASCII 码排序理解)

① str1 与 str2 相等 strcmp(str1, str2) == 0

② str1 与 str2 不相等 strcmp(str1, str2) != 0
strcmp(str1, str2)

③ str1 在 str2 的前面 (此时 str1 < str2) strcmp(str1, str2) < 0
// str1 中的字符比 str2 的字符数值小

④ str1 在 str2 的后面 (此时 str1 > str2) strcmp(str1, str2) > 0
// str1 中的字符比 str2 的字符数值小

strchr() 的用法:

strchr(字符串, 字符);

返回指定字符在字符串中第一次出现的地址。若没有该字符, 返回 NULL

18. ctime 的用法: 见正文

19. cin 的用法(二)

在检测到 EOF(END OF FILE, 即文件尾)后, cin 将两位 eofbit 和 failbit 设为 1。

方法	说明
<code>cin.eof()</code>	检测到 <code>eofbit</code> 被设为 1, <code>cin.eof()</code> 返回 <code>true</code> , 否则返回 <code>false</code> .
<code>cin.fail()</code>	如果 <code>eofbit</code> 或 <code>failbit</code> 设为 1, 则 <code>fail()</code> 成员返回 <code>true</code>
	两个函数报告最近读取的结果
<code>cin.clear()</code>	重置输入
<code>istream::operator bool()</code>	转为 <code>bool</code> 值. 若最后一次读取成功, 返回 <code>true</code> , 失败返回 <code>false</code> . 其比两个成员函数更通用。
<code>cin.get()</code>	返回输入中的下一个字符. 到达 EOF 时返回符号常量 <code>EOF(-1)</code>
<code>cout.put(char ch)</code>	显示一个字符

20. `cctype`: 见正文

21. `if` 语句

```
if(测试表达式)
    语句
```

22. `if-else` 语句

```
if(测试表达式)
    语句 1
```

```
else
```

```
    语句 2
```

23. `if-else if-else` 结构 (`if-else` 嵌套)

```
if(条件 0)
```

```
    语句 0
```

```
else if(条件 1)
```

```
    语句 1
```

```
.....
```

```
else if(条件 n)
```

```
    语句 n
```

```
else
```

```
    语句 n+1
```

最后一个 `else` 可以没有。

24. `switch` 语句结构

```
switch(integer-expression)
{
    case label1:statement(s)
    case label2:statement(s)
    ...
    default:statement(s)
}
```

25. `break` 语句结束循环

`continue` 语句用于循环当中, 跳过循环体余下的代码, 开始新一轮循环。

25. 文件输入输出: 见正文。

第三部分: 函数、内存与名称空间

一、基本知识

1. 函数包括没有返回值的函数 (`void` 函数)、有返回值的函数。

函数的原型声明格式

返回类型 函数名(形参列表);

函数的定义的格式

返回类型 函数名(形参列表);

2. 没有返回值的函数 (`void` 函数)

通用格式:

```
void 函数名(形参列表)
{
    语句
    return; // 可选
}
```

返回语句是可选的，意味着 void 函数的结束。通常函数在右花括号处结束。

3. 有返回值的函数

通用格式：

返回类型名 函数名(形参列表)

```
{
    语句
    return 返回值;
}
```

有返回值的函数必须使用返回语句，返回值可以是常量、变量或表达式。**返回值不能是数组！**
函数的形参列表可以为空，可以有多条 return 语句

4. 函数原型

(1) 为什么需要原型：函数原型可以指出函数的特征

(2) 原型的语法：函数原型可以不提供变量名，有类型列表足矣。

(3) 原型的功能：

- 使编译器正确处理函数返回值
- 使编译器检查使用的参数数目是否正确
- 使编译器检查使用的参数类型是否正确；若不正确可以进行类型转换。
- 通常原型自动将被传递的参数强制转换为期望的类型。

注意：• 仅当有意义时，**原型化**导致类型转换。

- 在编译阶段进行原型化被称为**静态类型检查**，可以捕获许多在运行期间难以捕获的错误。

note: 在 C++ 中，括号为空与在括号中使用 void 等效。ANSI C 中括号为空意味着不指出参数（参数由定义指出）。C++ 中不指定参数列表用省略号(...)。

二、函数参数、按值传递、函数作用域

1. C++ 通常**按值传递参数**，即将数值参数传递给函数，并赋给新的变量。

e.g. 函数原型是 `double cube(double x)`，则 `cube(3)` 调用时把 x 赋值为 3。此处的 x 是一个新的变量，不影响调用函数。

2. 用于接受传递值的变量称为**形参**。传递给函数的值为**实参**。以上例子 x 为形参，3 为实参。

C++ 标准(以下简称“标准”)使用**参数(argument)**表示实参，使用**参量(parameter)**来表示形参。参数会赋给参量。

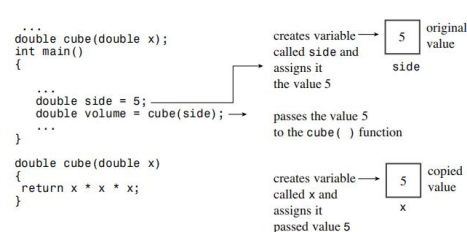
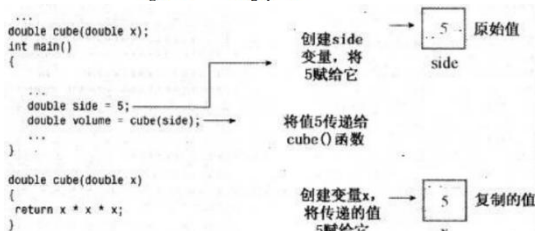


Figure 7.2 Passing by value.



3. 函数的内存是专用的，并且函数内的变量不会影响外部的同名变量，在执行完毕后函数释放内存。这些变量叫**局部变量**，也叫**自动变量**，在执行过程中自动分配释放。

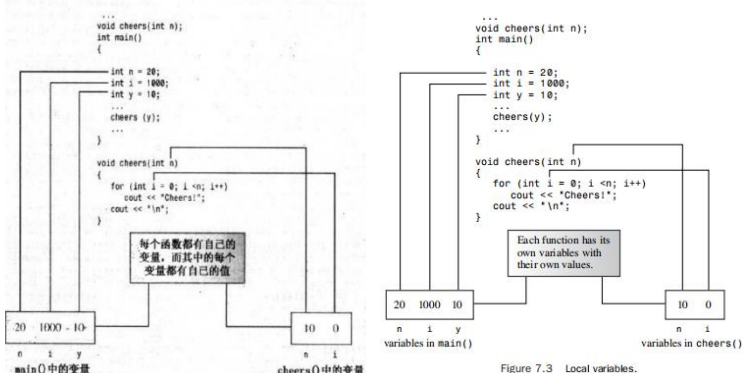


Figure 7.3 Local variables.

4. 形参可以用=指定**默认参数**，注意**按值传递时不允许中间断开**，只能在末尾的参数指定默认参数。

三、函数与数组

1. 示例代码: `int sum(int arr[],int n)`

函数的形参中没有数组, 只有指针。但是在函数中仍然可以像使用数组一样使用它。在示例中 arr 是一个指针, 绝非数组。

当(且仅当)在函数头或函数原型中, `int *arr` 和 `int arr[]` 的含义相同。

注意: 传递数组 **传递了数组的地址**, 也是按值传递, 不过使用的仍然是原来的数组。

在以上实例中, 若 arr1 是一个 8 个元素的 int 数组, sizeof arr1 的值为 32. 而 arr1 作为参数传递给 sum 后, sizeof arr 为 4 (32 位系统) 或 8 (64 位系统)。

★★★★★2. [易混淆点] 指针与 const

①. 指针可以指向 **const 值 (Pointers-to-const)**, 此时指针指向的地址可以修改, 但是指向的内容不可以修改; 指针变量本身也可以是 **const 值 (const Pointers, const 指针)**, 此时指针指向的地址不能修改, 指向的内容可以修改; **const 指针指向 const 值时, 指针和指向的值都不能修改。**

注意: 要区别是指针的值还是指向的对象的价值能修改。

② 指向常量的指针 pt

```
int age=39;
const int *pt=&age;
```

以上的声明指出, pt 指向 const int (此处为 39), 不能用 pt 修改 age (因为 *pt 理应是 const int), 但是可以用 age 来修改自己的值。在此处的声明不一定意味着 pt 指向的值一定是 const, 而是相对于 pt 而言, 它指向的值是 const。

③ 可以把常规变量赋给常规指针, 可以把常规变量赋给指向 const 的指针, 可以把 const 变量的地址赋给指向 const 的指针, 但是不可以将 const 变量的地址赋给常规指针。

④ 涉及一级间接关系时可以将非 const 指针赋给 const 指针。(一级间接关系类似 `Type_ *p;`)

note: 要在函数的形参中尽可能使用 const

⑤ 禁止将 const 数组的地址赋给非 pointers-to-const 指针。

⑥. 无法修改指针值的指针

```
int sloth=3;
const int *ps=&sloth;
int * const finger=&sloth;
```

以上代码中 finger 只能指向 sloth, 但允许使用 finger 修改 sloth 的值。不允许使用 ps 来修改 sloth 的值, 但允许 ps 指向另一个值。finger 和 *ps 都是 const, *finger 和 ps 不是。

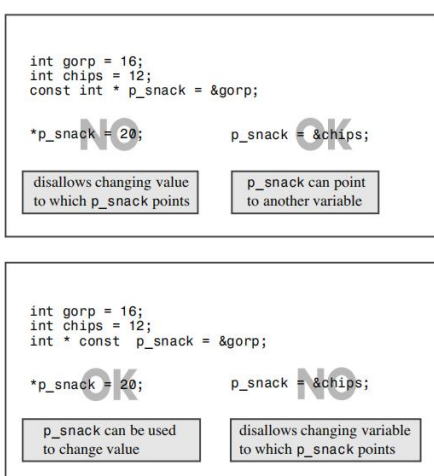


Figure 7.5 Pointers-to-const and const pointers.

```
double trouble=2.0E30;
```

```
const double * const stick=&trouble;
```

在此处 stick 和 *stick 都是 const. stick 只能指向 trouble, stick 也无法修改 trouble.

⑦ 总结:

指向 const 的指针: **const 指向类型 *指针名=值;**

***指针名指向的类型**

const 指针: **指向类型 * const 指针名=值;**

说明是一个指针

指向 const 的 const 指针:

const 指向类型 *const 指针名=值;

3. 函数与二维数组

```
int data[3][4]={ {1,2,3,4}, {9,8,7,6}, {2,4,6,8} };
int total=sum(data,3);
```

以上代码中 data 是一个数组, 第一个元素是由 4 个 int 构成的数组。

则 sum() 的原型如下:

```
int sum(int (*ar2)[4],int size);
```

其中的括号必不可少。int *ar2[4]; 声明一个由 4 个指向 int 的指针组成的数组。加上括号则声明一个指向由 4 个 int 组成的数组的指针

note: 数组与指针的一些声明该如何看?

数组声明 Type_ arrname[length];

此时先看前面的 Type_ arrname, 这里指出了数组元素的类型是 Type_ 和 arrname 是一个数组。

指针声明 Type_ *pointername;

此时 *pointername 是一个 Type_ 类型的数据, 则 pointername 应该是一个指向 Type_ 类型的指针。

同理当声明是 Type_ **p; 时, **p 是 Type_ 类型, *p 是 Type_* 类型, **p 为 Type_**.

当声明是 Type_ (*p_name)[length]; 时, 由于小括号的优先级大于中括号, 则应当把 (*p_name) 看作一个整体。此时 (*p_name) 是一个有 length 个 Type_ 组成的数组, 则 p_name 就是指向数组的指针了。

警告: 在实际工作中请使用简洁明了的声明, 不复杂化声明。

在函数原型中也可以写成这样: int sum(int ar2[][4],int size); 可读性更好

在函数的定义中可以把 ar2 看作数组的名称。由于 ar2 指向数组, ar2+r 表示二维数组的第 r 行, 所以 ar2[r][c] 的表达方式可以用在 ar2 上。

5. 函数与 C 风格字符串: 注意字符串可能需要使用 const 关键字定义函数。

接受字符串时不需要传递长度, 因为有 '\0'。'\0' 的 ASCII 码为 0, 则可以用循环条件判断它。函数不能返回字符串, 但是可以返回其地址。

四、函数与其他类型

- 数学库 (cmath) 的 atan2(y,x) 可以根据 x 和 y 计算角度。(tan α = y/x, 此处的库函数在已知 y 和 x 的情况下求 α)。atan() 不能区分 180 之内和之外的角度。

- 传递结构可以用直接按值传递和传递结构的地址。传递地址节省时间空间。

五、递归

- 函数调用自己叫 **递归** (类似于数学中的递归定义), C++ 中 main() 不能调用自己

- 包含一个递归调用的递归: void 函数的递归的一般形式如下:

void 函数名(形参)

```
{
    statements1
    if(text)
        函数名(实参);
    statements2
}
```

递归要有退出的条件, 让调用链断开。在每次递归调用时, 控制权交给新调用的函数, 而后调用链断开时逐渐返回。

note: 函数调用时会返回地址和函数实参压入栈中 (与汇编 call 与 ret 指令有一定关系)。栈空间是有限的, 切勿进行过多递归调用。

- 包含多个递归调用的递归

在需要将一件工作不断分为两项较小的、类似的工作时, 使用递归, 可以将同样的操作作用于每一部分, 有时也叫 **分而治之策略**。

实例: (摘自 C++ Primer Plus)

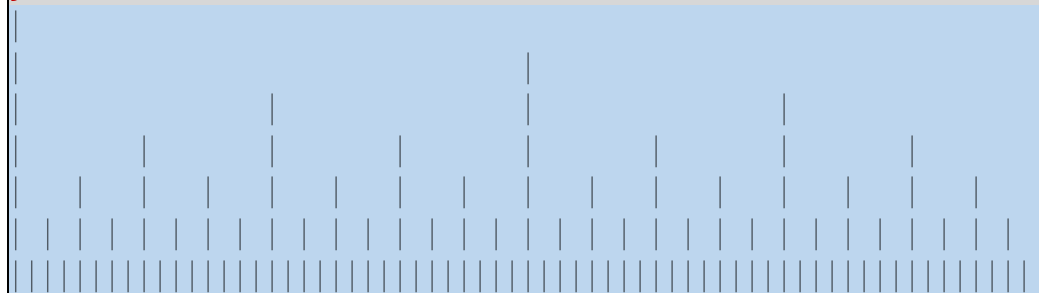
```
// ruler.cpp -- using recursion to subdivide a ruler
#include <iostream>
const int Len = 66;
const int Divs = 6;
void subdivide(char ar[], int low, int high, int level);
```

```

int main()
{
    char ruler[Len];
    int i;
    for (i = 1; i < Len - 2; i++) ruler[i] = ' ';
    ruler[Len - 1] = '\0';
    int max = Len - 2;
    int min = 0;
    ruler[min] = ruler[max] = '|';
    std::cout << ruler << std::endl;
    for (i = 1; i <= Divs; i++)
    {
        subdivide(ruler,min,max, i);
        std::cout << ruler << std::endl;
        for (int j = 1; j < Len - 2; j++)
            ruler[j] = ' '; // reset to blank ruler
    }
    // std::cin.get();
    return 0;
}

void subdivide(char ar[], int low, int high, int level)
{
    if (level == 0) return;
    int mid = (high + low) / 2;
    ar[mid] = '|';
    subdivide(ar, low, mid, level - 1);
    subdivide(ar, mid, high, level - 1);
}

```



六、函数指针 (指向函数的指针, 简称函数指针, Pointers to Functions)

1. 函数的地址是存储其机器语言代码的内存的开始地址。
2. 函数名 (后面不跟参数) 就是函数的地址。如 `a()` 是函数, `a` 就是它的地址。
即 函数名 (参数列表) 是函数调用, 而 函数名 是其地址。

```

function1(a());
function2(a);

```

以上的代码会先调用函数 `a()`, 将其返回值传递给 `function1()` 函数。而函数 `a()` 的地址 `a` 被直接传递给 `function2()`

3. 声明函数指针: 注意函数指针要包含函数的返回类型和函数的特征标 (参数列表)。

若函数原型是 `double pam(int);`

则函数指针的声明如下: `double (*pf)(int);`

这里的(*pf)与 pam 都是函数，则 pf 就是函数指针了。

注意：必须使用括号将*pf 括起。括号的优先级比*高，则 `double *pf(int);` 意味 pf() 是一个返回指针的函数，(*pf)(int) 意味着 pf 是一个指向函数的指针。

此时可以将相应的函数的地址赋给 pf，**特征标和返回类型都必须相同**，若不相同，编译器会拒绝这种赋值。

4. 使用函数指针调用函数：在以上实例中(*pf)相当于函数名，所以在它后面加上参数列表可以调用函数。也可以使用 pf 来进行调用。

```
double pam(int);
double (*pf)(int);
pf=pam;
double x=pam(4);
double y=(*pf)(5); //两种格式可以互换
y=pf(5);
```

note:pf 与(*pf)等价的原因：一种说法是由于 pf 是函数指针，*pf 是函数，因此(*pf)() 是函数调用；另一种说法是函数名是指向该函数的指针，指向函数的指针应与函数名相似，所以 pf()；可以作为函数调用。C++允许两种写法。

5. 总结：函数指针的声明：

返回类型 (*指针名)(形参类型列表); //可以使用 auto 初始化以达到简化的效果

示例：冒泡排序 Bubble_Sort

```
#include<iostream>
using namespace std;
void bubble_sort(int*,bool (*)(int,int));
bool greaterthan(int,int);
bool smallerthan(int,int);
int main()
{
    int arr1[10];
    int arr2[10];
    cout<<"键入 10 个数据以进行排序:\n";
    for(int i=0;i<10;i++)
    {
        cin>>arr1[i];
        arr2[i]=arr1[i];
    }
    bubble_sort(arr1,greaterthan);
    bubble_sort(arr2,smallerthan);
    cout<<"从小到大:";
    for(int i=0;i<10;i++){cout<<arr1[i]<<" ";}
    cout<<endl;
    cout<<"从大到小:";
    for(int i=0;i<10;i++){cout<<arr2[i]<<" ";}
    cout<<endl;
    return 0;
}
void bubble_sort(int arr[10],bool (*condition)(int,int))
{
```

```

for(int i=9;i>0;i--)
{
    for(int j=0;j<i;j++)
    {
        if(condition(arr[j],arr[j+1]))
        {
            int temp=arr[j+1];
            arr[j+1]=arr[j];
            arr[j]=temp;
        }
    }
}
}

bool greaterthan(int x,int y){return x>y;}
bool smallerthan(int x,int y){return x<y;}

```

键入 10 个数据以进行排序:

12 254 2323 -34 0 44 -23 0 10 1

从小到大:-34 -23 0 0 1 10 12 44 254 2323

从大到小:2323 254 44 12 10 1 0 0 -23 -34

6. 深入函数指针: 部分函数的原型:

```

const double * f1(const double ar[],int n);
const double * f2(const double [],int);
const double * f3(const double *,int);

```

以上函数的特征标相同。

若要声明一个指针指向这 3 个函数, 指针名为 p1, 此时将目标函数名换为 (*p1)

得到: `const double *(*p1)(const double*,int);` 因为 (*p1) 是函数, 则 p1 是函数指针
此时仅仅只是 **单值**, 所以 **可以用 auto, 当要初始化数组时, 有初始化列表, 不能用 auto。**

若要声明一个包含三个函数指针的数组, 声明如下:

```
const double *(*pa[3])(const double*,int)={f1,f2,f3};
```

声明数组时需要使用 pa[3], 其他部分指明了数组的元素特征, 运算符 [] 的优先级高于 *, 因此 *pa[3] 表明 pa 是一个包含三个指针的数组。所以, pa 是包含 3 个指针的数组, 每个指针都指向以 const double* 和 int 作为参数, 并返回 const double* 的函数。

若要声明 pd 指向 pa, 则声明如下:

```
const double *(*(*pd)[3])(const double*,int)=&pa;
```

由于 (*pd) 是一个数组, 则 pd 就是一个指向数组的指针。

note: 复杂声明拆解:

先从整体看。以上声明若是第一次遇见, 则要先大致看。此处可以明确声明包括了函数指针, 此时 (*(*pd)[3]) 是函数, 则 ((*pd)[3]) 是函数指针, (*pd) 是一个有 3 个元素的数组, 每个元素都是函数指针, 则 pd 就是指向这个数组的指针。总结一下, pd 是指向一个有三个元素, 每个元素都是指向接受 const double* 和 int 返回 const double* 的函数的函数指针。

原书程序:

```

#include <iostream>
const double * f1(const double ar[], int n);
const double * f2(const double [], int);
const double * f3(const double *, int);
int main()

```



```

{
    using namespace std;
    double av[3] = {1112.3, 1542.6, 2227.9};
    const double *(*p1)(const double *, int) = f1;
    auto p2 = f2;
    cout << "Using pointers to functions:\n";
    cout << " Address Value\n";
    cout << (*p1)(av,3) << ": " << (*p1)(av,3) << endl;
    cout << p2(av,3) << ": " << *p2(av,3) << endl;
    const double *(*pa[3])(const double *, int) = {f1,f2,f3};
    auto pb = pa;
    cout << "\nUsing an array of pointers to functions:\n";
    cout << " Address Value\n";
    for (int i = 0; i < 3; i++)
        cout << pa[i](av,3) << ": " << *pa[i](av,3) << endl;
    cout << "\nUsing a pointer to a pointer to a function:\n";
    cout << " Address Value\n";
    for (int i = 0; i < 3; i++)
        cout << pb[i](av,3) << ": " << *pb[i](av,3) << endl;
    cout << "\nUsing pointers to an array of pointers:\n";
    cout << " Address Value\n";
    auto pc = &pa;
    cout << (*pc)[0](av,3) << ": " << *(*pc)[0](av,3) << endl;
    const double *(*pd)[3])(const double *, int) = &pa;
    const double * pdb = (*pd)[1](av,3);
    cout << pdb << ": " << *pdb << endl;
    cout << *(*pd)[2](av,3) << ": " << *(*pd)[2](av,3) << endl;
    return 0;
}

const double * f1(const double * ar, int n){return ar;}
const double * f2(const double ar[], int n){return ar+1;}
const double * f3(const double ar[], int n){return ar+2;}

```

Using pointers to functions:

Address Value

002AF9E0: 1112.3

002AF9E8: 1542.6

Using an array of pointers to functions:

Address Value

002AF9E0: 1112.3

002AF9E8: 1542.6

002AF9F0: 2227.9

Using a pointer to a pointer to a function:

Address Value

002AF9E0: 1112.3

002AF9E8: 1542.6

002AF9F0: 2227.9

Using pointers to an array of pointers:

Address Value

002AF9E0: 1112.3

002AF9E8: 1542.6

002AF9F0: 2227.9

- 函数指针数组的通用套路：返回类型 (*指针名[元素个数]) (形参类型列表)={函数列表};
- 使用 typedef 简化声明：将别名当作标识符进行声明，在开头使用关键字 typedef. 如：

```
typedef const double *(*p_fun)(const double *, int); // p_fun now a type name
p_fun p1 = f1; // p1 points to the f1() function
```

注意：这里的 p_fun 是新的类型，而不是变量。

三、内联函数

在声明和定义前加上 inline，一般省略声明，在 main() 函数前直接定义。

编译器用内联代码替换函数调用。因此，内联函数不可递归。

若函数简短且十分常用，可以使用内联函数。

note: 内联与宏(#define)

由于宏是通过文本替换实现的，则#define SQUARE(X) X*X 在替换 SQUARE(4.5+7.5) 时会出现问题，此时加上括号，变成#define SQUARE(X) ((X)*(X))，但是 SQUARE(C++) 并不会先让 C++ 递增再平方，而是替换为 (C++)*(C++)，显然不符合预期。所以要考虑把 C 宏换位内联函数。

四、引用变量

1. 引用是乙定义的变量的别名。

2. 创建：引用的类型 &引用名=被引用的变量；

如 int &a=b; 其中 int &指的是指向 int 的引用。引用与被引用的变量指向相同的值和内存单元。

(变量名其实是对内存单元的标识，一个内存单元可以对应多个标识，所以所有的标识等效)

• 必须在声明时将引用初始化，一旦与某个变量（被引用的变量）关联起来，就一直效忠于它，不能用赋值运算符修改。（在初始化后赋值运算符只能像常规变量一样使用）

3. 将引用作为函数参数（按引用传递）

• 将引用作为函数参数叫做按引用传递。

• 只有按引用传递才能修改调用函数中变量的值。如交换变量的函数需要引用。

• 注意：传递引用会改变原变量的值，常规类型变量最好用按值传递。在传递结构变量和类对象时，用引用比较好。

• 如果函数形参是引用，则应该传入一个变量，而不是一个常量或含有多项的表达式（如 1+a）部分情况下，程序会创建无名变量，将其初始化为传入的表达式，再去引用该变量。

• 临时变量、引用参数和 const

(1) 左值：可以被引用的数据对象。变量、数组元素、结构成员、引用和解除引用的指针都是左值。非左值（右值）包括字面常量（用引号括起的字符串常量除外，其由其地址表示）和包含多项的表达式。左值可用地址访问，包括常规变量（可修改的左值）和 const 变量（不可修改左值）。



(2) 编译器创建临时变量的情况：当引用参数为 const 时

实参的类型正确，但不是左值

实参的类型不正确，但可以转换为正确的类型

例子：

```
double refsqu(const double&);
int a=3;
double b=refsqu(a); // 会创建临时变量
```

```
double c=refsqu(a+1);//会创建临时变量
```

note: 最好尽可能使用 `const` 作为形参

(*) C++11 新增了右值引用，可指向右值，用 `&&` 声明。用 `&` 声明的时左值引用。

4. 引用与结构、类对象

返回引用：传统的返回计算关键字 `return` 后的表达式的值，然后把值复制到一个位置以返回，然后调用函数再使用它。返回引用时无需拷贝值（此时函数相当于被引用变量的别名）。

注意：要避免返回指向函数终止时不在存在的变量的引用（指针）。可以返回一个作为参数传递给函数的引用或使用 `new` 关键字。

- 不能通过地址访问的值叫做右值，这些表达式只能出现在赋值语句右边
- 可以将 C 风格字符串用作 `string` 对象引用参数，因为 `string` 类定义了 `char*` 到 `string` 的转换功能。**注意：要使用 `const string&` 形参。**

5. 对象继承和引用

- 使得能够将特性从一个类传递给另一个类的语言特性叫做**继承**。
- `ostream` 是**基类**，`ofstream` 是**派生类**，派生类继承了基类的方法，可以使用基类的特性
- 基类引用可以指向派生类对象，无需强制类型转换。如：将参数类型为 `ostream&` 的函数可以接受 `ostream` 对象和 `ofstream` 对象。
- **`ostream` 中的方法 `setf()` 可以设置格式化状态。**其返回调用它之前所有有效的格式化设置，用 `ios_base::fmtflags` 类型来存储。
- `setf(ios_base::fixed)` 设置定点表示法，方法 `precision(位数)` 在定点模式下指定显示多少位小数。`setf(ios_base::showpoint)` 显示小数点，即使小数部分为 0。这些设置在下次调用方法设置前一直保持不变。方法 `width(宽度)` 设置下一次输出操作使用的字段宽度，该设置在显示下一次值时有效，然后将恢复默认设置。
- 给 `setf()` 传入之前保存的设置 (`ios_base::fmtflags` 类型的变量) 可以恢复原有设置，如：

```
ios_base::fmtflags init_setting=cout.setf(ios_base::boolalpha);
.....
cout.setf(init_setting);
```

五、默认参数

- 默认参数指的是当函数调用中省略了实参时自动使用的一个值。在调用函数时可以选择覆盖默认值或直接使用默认值。
- 使用：通过函数原型设置默认值。如：

```
char *left(const char*str,int n=1);
```

- 对于带参数列表的函数，必须从右向左添加默认值。若要为某个参数设置默认值，则必须为它右边的所有参数提供默认值：

```
int a(int n,int m=0,int k=0);//可以
int b(int n,int k=0,int s);//不可以
int c(int n,int k,int s=0);//可以
```

六、函数重载

- **多态**：有多种形式
- **函数重载（多态）**：可以有多个同名的函数，重载的函数特征标应该不同。
- **函数特征标**：函数的参数列表（函数参数数目和类型，参数的排列顺序）**与变量名、返回类型无关**。
- 编译器根据所采取的用法使用有相应特征标的原型。
- 若没有匹配的原型，会尝试进行强制类型转换以匹配。若匹配到多个函数（或有多种转换方式），编译器会拒绝这种函数调用。
- 编译器检查函数特征标时，把类型引用和类型本身视作同一特征标。
- 匹配函数时，会区分 `const` 和非 `const` 变量。

```
void a(const int &);//#1
```

```
void a(int &); // #2
```

此时有函数调用 `a(b)`; 若 `b` 为 `const int`, 则调用 #1 版本的函数, 若 `b` 为 `int`, 调用 #2 版本的函数。编译器根据实参是否为 `const` 决定使用那个原型。

```
void staff(double & rs); // matches modifiable lvalue
void staff(const double & rcs); // matches rvalue, const lvalue
void stove(double & r1); // matches modifiable lvalue
void stove(const double & r2); // matches const lvalue
void stove(double && r3); // matches rvalue
```

左值引用参数 `r1` 与可修改的左值参数匹配, `const` 左值引用参数 `r2` 与可修改的左值参数、`const` 左值参数和右值参数匹配; 右值引用参数 `r3` 与右值匹配。此时调用 `stove()` 函数时会调用最匹配的版本。若没有定义 `stove(double &&)`, `stove(x+y)` 将调用函数 `stove(const double&)`。

- 重载示例: 实用示例 8-1

任务 1: 分离数字各位

```
#include<iostream>
using namespace std;
int main()
{
    int n; // 输入一个数字 n 进行操作
    cin >> n;
    while(n != 0) // 核心部分
    {
        cout << n % 10 << " ";
        n /= 10;
    }
    return 0;
}
```

1428579

9 7 5 8 2 4 1

分析: 此程序的原理可以类比十进制转换为二进制, 十进制转换为二进制用除 2 取余法, 此时可以知道每一位分别是什么。那么, “将十进制转换为十进制” 时, 除 10 取余可以看出每一位是什么。输入 1428579, 首先计算 $1428579 \% 10$, 得到最后一位 9, 然后 $1428579 / 10$, 得到 142857, 消去了一位, 计算 $142857 \% 10$, 得到最后一位 7, 依次类推, 在计算到 $1 / 10$ 时, 变量 `n=0`, 循环结束。

任务 2: 计算整数位数: 类比任务 1, 每次除以 10 去掉一位。

任务 3: 编写 `left()` 函数, 返回字符串前 `n` 位或整数前 `n` 位。

1. `left()` 实现:

2. 组成程序。

note: 编译器处理函数重载时使用 **名称修饰 (矫正)**, 根据形参类型标注函数。

七、函数模板

- **函数模板** 是通用的函数描述。

—> 函数模板用 **泛型** 来定义函数, 其中的泛型可以用具体的类型替换。

—> 通过将类型作为参数传递给模板, 可使编译器生成该类型的函数。

- 模板特性也叫 **参数化类型**。

示例：交换变量的模板

```
template <typename AnyType> // template typename are keywords
void Swap(AnyType &a, AnyType &b)
{
    AnyType temp;
    temp=a;
    a=b;
    b=temp;
}
```

第一行的 template 指出要创建一个模板，关键字 typename 指出 AnyType 是一个类型，类型名可任意选择，可用关键字 class 代替 typename.（两个关键字等价）

声明模板的格式：template<typename 类型 0,...> 函数

▲模板不会创建任何函数，而是指出如何定义函数。

在以上例子中，若要交换 int 值，编译器把 AnyType 用 int 替换。

- 在最终的代码中不包含任何函数定义，只包含为了程序生成的实际函数。

1. **模板重载**：函数特征标必须不同。并非所有的模板参数都必须是模板参数类型

2. **显式具体化 (explicit specialization)**

- **显式具体化**是为特定类型提供的具体化函数定义。
- 当编译器找到与函数调用匹配的具体化定义时，将使用该定义，而不再寻找模板。

▲▲标准：第三代具体化

- 对于给定的函数名，可以有非模板函数、模板函数和显式具体化模板函数以及它们的重载版本
- 显示具体化的原型定义应以 template<> 打头，并通过名称来指出类型
- 具体化优先于常规模板，非模板函数优先于具体化和常规模板
(优先顺序：非模板函数、具体化模板函数、常规模板函数)

下面是用于交换 job 结构的具体化：

template <> void Swap<job>(job&, job&); 其中的<job>可以省略。

格式：template<> 返回类型 函数名<具体化类型>(形参列表);

或者 template<> 返回类型 函数名(形参列表);

3. **实例化和具体化**

• 编译器使用模板为特定类型生成函数定义时，得到的是**模板实例 (instantiation)**。模板并非函数定义，模板实例是函数定义。

- 函数调用使编译器产生实例的实例化方法是**隐式实例化 (implicit instantiation)**。

- **显式实例化 (explicit instantiation)**：直接命令编译器创建特定的实例。语法：

template 返回类型 函数名<实例类型>(形参类型列表);

模板函数名<实例类型>() 表示模板函数的特定实例类型的函数调用。如 Swap<int>(a,b);

此处 Swap<int>可以赋给相应的函数指针。

- 可以使用函数来创建显示实例化，语法：**模板函数名<实例类型>(实参列表);**

此时可以将实参强制转换为实例类型。如 Add<int>(x,y); 要求传入的两个参数的类型相同，其中 x 为 short 类型，y 为 int 类型，则会把 x 转为 int 类型以匹配。

- **隐式实例化、显式实例化、显式具体化**统称**具体化 (specialization)**

note: 显示具体化是针对特殊（特定）的类型，定制出不同于通用（常规）模板的方案，两种实例化虽然也针对特定的类型，但却使用通用（常规）的模板生成。

4. 编译器选择函数

编译器选择函数的过程：**重载解析**：步骤：

(1) 创建候选函数列表（所有同名函数、模板） (2) 创建可行函数列表（参数数目正确） (3) 确定是否有最佳的可行函数，有则使用，没有则出错

确定最佳的可行函数时，可能进行转换，从最优到最后的顺序如下：

(1) 完全匹配 (2) 提升转换 (如整型提升) (3) 标准转换 (4) 用户定义的转换

- 完全匹配与最佳匹配

Table 8.1 Trivial Conversions Allowed for an Exact Match

From an Actual Argument	To a Formal Argument
Type	Type &
Type &	Type
Type []	* Type
Type (argument-list)	Type (*) (argument-list)
Type	const Type
Type	volatile Type
Type *	const Type *
Type *	volatile Type *

进行完全匹配时，允许某些无关紧要的转换，下表中 Type 为任意类型，Type (argument-list) 表示用作实参的函数名与用作形参的指针只要返回类型和参数列表相同，就是匹配的。

有时候两个函数完全匹配也可以完成重载解析。

▲指向非 const 的指针 (引用) 优先与非 const 指针 (引用) 匹配，const 与非 const 之间的区别只适用于指针和引用，

否则会导致二义性。

- 如果两个完全匹配的函数都是模板函数，则较具体的模板函数优先。(如显示具体化优于隐式生成的具体化)：

```
template <class T>void a(T t); // #1
template <> void a<string>(string& t); // #2
string x;
a(x); // 调用 #2
```

- **最具体** (most specialized): 编译器推断使用那种类型时执行的转换最少。

```
template <class Type> void recycle (Type t); // #1
template <class Type> void recycle (Type * t); // #2
struct blot {int a; char b[10];};
blot ink = {25, "spots"};
recycle(&ink); // address of a structure
```

此时调用的应该是 #2。

- 用于找出最具体的模板的规则叫做函数模板的部分排序规则。

```
原书示例: // tempover.cpp --- template overloading
#include <iostream>
template <typename T>           // template A
void ShowArray(T arr[], int n);
template <typename T>           // template B
void ShowArray(T * arr[], int n);
struct debts
{
    char name[50];
    double amount;
};
int main()
{
    using namespace std;
    int things[6] = {13, 31, 103, 301, 310, 130};
    struct debts mr_E[3] =
    {
        {"Ima Wolfe", 2400.0},
        {"Ura Foxe", 1300.0},
        {"Iby Stout", 1800.0}
    };
};
```

```

double * pd[3];
for (int i = 0; i < 3; i++) pd[i] = &mr_E[i].amount;
cout << "Listing Mr. E's counts of things:\n";
ShowArray(things, 6); // uses template A
cout << "Listing Mr. E's debts:\n";
ShowArray(pd, 3); // uses template B (more specialized)
return 0;
}

template <typename T>
void ShowArray(T arr[], int n)
{
    using namespace std;
    cout << "template A\n";
    for (int i = 0; i < n; i++) cout << arr[i] << ' ';
    cout << endl;
}

template <typename T>
void ShowArray(T * arr[], int n)
{
    using namespace std;
    cout << "template B\n";
    for (int i = 0; i < n; i++) cout << *arr[i] << ' ';
    cout << endl;
}

```

Listing Mr. E's counts of things:

template A

13 31 103 301 310 130

Listing Mr. E's debts:

template B

2400 1300 1800

`ShowArray(pd, 3);` 该函数调用与模板 A、B 都匹配，模板 B 更具体，指出了数组内容是指针，因此而被使用。

- 创建自定义选择

使用 `函数名<>(参数列表);` 的形式时，编译器选择模板函数，而不是非模板函数。

注意：即使去掉<>后有相应的非模板函数，编译器在发现<>后也不使用。

5. C++11: 类型推导关键字 `decltype`

```

int x;
decltype(x) y;

```

则此时的 `y` 与 `x` 的类型相同。可以给 `decltype()` 提供表达式，推导表达式结果的类型。

若有以下声明 `decltype(expr) var;` 则编译器会这样核对类型：

step 1: 若 `expr` 是没有用括号括起的标识符，`var` 的类型应该与该标识符类型相同，包括 `const` 等限定符

step 2: 若 `expr` 是一个函数调用，则 `var` 类型与函数返回类型相同

step 3: 若 `expr` 是以一个左值，则 `var` 为指向其类型的引用。（注意：只有不符合前面所有步骤才进入第三步，则当 `expr` 是用括号括起的标识符会进入第三步）

```
double xx=2.8;
```

```
decltype((xx)) r2=xx; //r2 is double &
decltype(xx) w=xx; //w is double
```

▲括号不会改变表达式的值和左值性。

step 4: 如果前面都不满足, var 类型与 expr 类型相同

6. C++11 后置返回类型

语法: **auto 函数名(形参列表) -> 后置返回类型;**

不能把 decltype() 放在前面, 因为此时参数还未声明, 不在作用域中

以下两行代码等价:

```
double h(int x, float y);
auto h(int x, float y) -> double;
```

此时 double 是**后置返回类型**, auto 是**占位符**, 表示后置返回类型提供的类型。

示例:

```
template<class T1, class T2>
auto Sub(T1 a, T2 b) -> decltype(a-b)
{
    return abs(a-b);
}
```

八、单独编译

大型程序的源码可分为 3 部分:

头文件: 结构、类声明和函数原型

源代码文件 A: 包含有关函数的实现

源代码文件 B: 调用这些函数、main() 函数的所在文件

1. 引入头文件:

#include "自定义头文件"

用于引入**自定义**的头文件, 优先从**使用头文件的源代码所在目录查找**

#include <标准头文件>

优先从编译器存放头文件的位置查找, 引入标准头文件 (编译器扩展头文件也可以)

警告: 不要用#include 包含源代码文件

2. 管理头文件的预处理指令

#ifndef 宏名称

预处理指令 #ifndef (if not defined) 用于检测某个宏是否已被 #define 定义。其与 #endif 一同使用, 如果宏没有定义, 则在编译时才会使用它们之间的代码, 否则不使用。

```
#ifndef HEADER_1
```

```
int a=0;
```

```
#endif
```

如果 HEADER_1 宏被定义, 则 int a=0; 这条声明被忽略, 若 HEADER_1 没有定义这条声明有效。

#define 宏名称

#define 可以只定义一个符号而不定义值, 如 #define HEADER_1

头文件的常用结构

```
#ifndef 宏
```

```
#define 宏
```

```
//代码
```

```
#endif
```

以上的做法依据如下: 如果宏被定义, 忽略代码和 #define; 若没有定义宏, 则立马定义它, 并使用 #ifndef 和 #endif 之间的代码, 在下次该文件再次被包含时就会忽略代码和 #define, 防止代码重复。

九、存储持续性、作用域和链接性

C++的存储持续性

- **自动存储持续性**：在函数定义中声明的变量和函数参数为自动存储持续性。在程序执行其所属函数（代码块）时创建，函数（代码块）执行完毕时释放。
- **静态存储持续性**：在函数定义外定义的变量、使用 `static` 关键字定义的变量为此类型。在程序的整个运行周期都存在。
- *****线程存储持续性(C++11)**
- **动态存储持续性**：用 `new` 分配的内存。生命周期知道 `delete` 运算符释放内存。

1. 作用域：描述名称在多大的范围内可见

- 作用域为局部的变量仅仅只在定义它的代码块可用
- 作用域为全局的变量在定义位置到文件结尾都可用
- 自动变量的作用域为局部
- **函数原型作用域**中使用的名称只在包含参数列表的括号可见。
- 类中声明的成员的作用域为整个类
- 在名称空间中声明的变量的作用域为整个名称空间
- C++函数作用域为整个类（结构）或整个名称空间（包括全局）

2. 链接性：描述文件如何在不同单元间共享

- 链接性为外部——可在文间共享
- 链接性为内部——只能由一个文件中的函数共享
- 自动变量没有链接性，无法共享

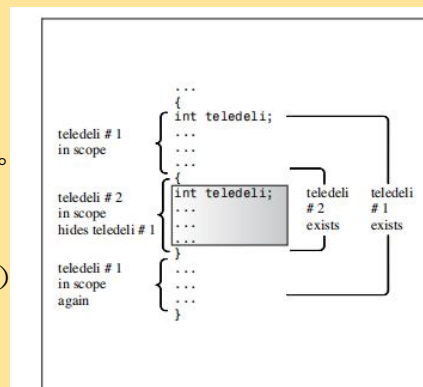


Figure 9.2 Blocks and scope.

3. 自动存储持续性

note: 早期版本的 C++ 中 `auto` 关键字显示指出变量为自动存储。

当函数调用时，自动变量被加入栈中，栈顶指针移到变量后的下一个内存单元。函数结束时，栈顶指针重置为函数被调用前的值。

栈是 LIFO(后进先出, late in first out) 的，函数调用将参数压入栈中。

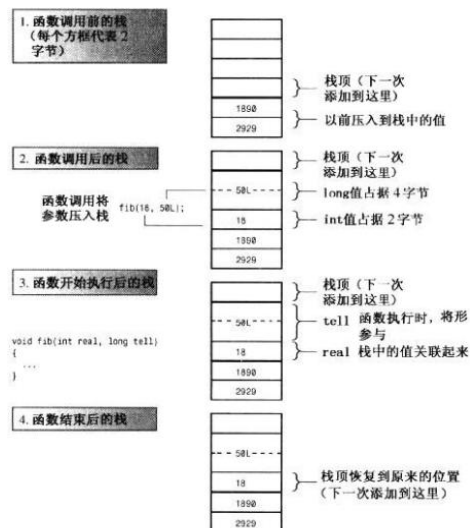


图 9.3 使用栈传递参数

• 寄存器变量：

关键字 `register` 建议编译器使用 CPU 寄存器存储自动变量。C++11 中变为显示指出变量是自动的。

4. 静态持续变量（3 种链接性）

- 编译器会分配固定内存块存储静态变量
- 没有显式初始化静态变量时，编译器将其设为 0；静态数组与结构每个元素（成员）会设为 0。
- 创建静态持续变量：
 - 链接性为外部：在代码块外声明即可（常规语法）
 - 链接性为内部：在代码块外声明并在前面加上 `static`。
- 语法： `static 类型 名称=值;`
- 无链接性：在代码块内声明并在前面加上 `static`。

• 未被初始化的静态变量所有位都设为 0，这种变量被称为 **零初始化的**。

• 静态变量的初始化

→ 静态变量还能进行 **常量表达式初始化和动态初始化**（运行阶段初始化）
 → **零初始化+常量表达式初始化=静态初始化**（编译阶段初始化）

→ 所有的静态变量都先被零初始化，然后若用常量表达式初始变量，且编译器可根据文件内容计算表达式的值，编译器就会进行常量表

存储描述	持续性	作用域	链接性	如何声明
自动	自动	代码块	无	在代码块中
寄存器	自动	代码块	无	在代码块中，使用关键字 <code>register</code>
静态，无链接性	静态	代码块	无	在代码块中，使用关键字 <code>static</code>
静态，外部链接性	静态	文件	外部	不在任何函数内
静态，内部链接性	静态	文件	内部	不在任何函数内，使用关键字 <code>static</code>

达式初始化。没有足够的信息（如遇到库函数）时，变量会被动态初始化。

常量表达式还可以使用 `sizeof` 运算符。

5. 静态持续性&外部链接性

- 外部变量：链接性为外部，存储持续性为静态，作用域为整个文件，在函数外部定义，也叫全局变量。

- 单定义规则：变量只能有一次定义

- C++的两种变量声明：

—>定义声明：简称定义，给变量分配存储空间。

—>引用声明：简称声明，不给变量分配存储空间，引用已有的变量

- 引用声明使用关键字 `extern`，且不进行初始化。若初始化，会成为定义并分配空间

引用外部变量 `extern 类型 变量名; // 不许初始化`

创建外部变量 `extern 类型 变量名=值;`

只需在一个文件中包含该变量的定义，其他玩家都必须使用关键字 `extern` 声明它。

```
// file1.cpp
#include <iostream>
using namespace std;

// function prototypes
#include "mystuff.h"

// defining an external variable
int process_status = 0;

void promise ();
int main()
{
    ...
}

void promise ()
{
    ...
}
```

这个文件定义变量
`process_status`，使得
编译器为它分配空间。

```
// file2.cpp
#include <iostream>
using namespace std;

// function prototypes
#include "mystuff.h"

// referencing an external variable
extern int process_status;

int manipulate(int n)
{
    ...
}

char * remark(char * str)
{
    ...
}
```

这个文件用`extern`指示
程序使用另一个文件中定义
的变量`process_status`。

注意：单定义规则不代表不能有多个变量的名称相同。
每个变量都是独立的，只有一个声明。

- 在定义与全局变量相同的局部变量后，局部变量隐藏全局变量。

- 访问变量的全局版本：`::变量名`

`::`叫做作用域解析运算符

在使用外部的函数时，只需声明原型

6. 静态持续性&内部链接性

文件中的静态变量(`static` 变量)会隐藏常规外部变量。

原书示例：

twofile1.cpp

```
// twofile1.cpp -- variables with external and internal linkage
#include <iostream> // to be compiled with two file2.cpp
int tom = 3; // external variable definition
int dick = 30; // external variable definition
static int harry = 300; // static, internal linkage
// function prototype
void remote_access();
int main()
{
    using namespace std;
    cout << "main() reports the following addresses:\n";
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n";
    remote_access();
    // cin.get();
    return 0;
}
```

twofile2.cpp

```
// twofile2.cpp -- variables with internal and external linkage
#include <iostream>
extern int tom; // tom defined elsewhere
static int dick = 10; // overrides external dick
```



```
int harry = 200;           // external variable definition,
                           // no conflict with twofile1 harry

void remote_access()
{
    using namespace std;
    cout << "remote_access() reports the following addresses:\n";
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n";
}

main() reports the following addresses:
0x403010 = &tom, 0x403014 = &dick, 0x403018 = &harry
remote_access() reports the following addresses:
0x403010 = &tom, 0x403020 = &dick, 0x403024 = &harry
```

7. 静态存储持续性&无链接性

无链接性的局部变量声明：在局部变量声明前加上 `static`. 只在其所在的代码块中可用，但是其一直存在。若其在函数中，两次函数调用之间静态局部变量的值保持不变。如果初始化了静态局部变量，则程序只在启动时进行一次初始化，以后在调用时，不会再进行初始化。

```
#include<iostream>
using namespace std;
int times(void)
{
    static int calledTimes=1;
    return calledTimes++;
}

int main()
{
    cout<<"第一次"<<times()<<"\n 第二次"<<times();
    return 0;
}

第一次 1
第二次 2
```

8. 说明符和限定符

(1) 存储说明符

- `auto` (C++11 之前)
- `register`
- `static`
- `extern`
- `thread_local` (C++11)
- `mutable`

(2) `const` 和 `volatile` (cv-限定符)

`volatile` 表示即使程序代码没有对内存单元进行修改，其值也可能发生变化。

- 默认情况下全局变量的链接性为外部，`const` 全局变量链接性为内部。
- 若要使得 `const` 链接性变为外部，加上 `extern`

(3) `mutable`

用来指出即使结构(类)变量为 `const`, 某个成员也可以被修改。

9. 函数的链接性

- 默认情况下函数存储持续性为静态，函数的链接性为外部的。
- 可在函数原型中使用关键字 `extern` 指出函数在另一个文件定义（可选）
- 可以使用关键字 `static` 将函数的链接性设置为内部的，使之只能在一个文件中使用（必须在函数的定义和原型中都加上）
- 静态函数将覆盖外部定义。
- 内联函数不受单定义规则的约束。同一函数的所有内联定义必须相同

10. 语言链接性

- 链接程序要求每个不同的函数都有不同的符号名
- **C 语言链接性**：一个名称对应一个函数，不会对函数名做过多的修饰
- **C++语言链接性**：同一名称对应多个函数，C++编译器进行名称矫正（修饰），为重载函数生成不同的符号名称。
- C++语言链接性与 C 语言链接性可能导致同一函数名链接时名称不同。
- 在 C++ 中使用 C 库预编译的函数：`extern "C" 返回类型 函数名(形参类型列表);`
- `extern 语言 返回类型 函数名(形参类型列表);`

以上的方法可以指定语言链接性，标准中规定了"C"、"C++"，不指出语言默认为 C++。

十、存储方案与动态分配、定位 new 运算符

- 动态内存由运算符 `new` 和 `delete` 控制。

1. 为**标量类型**分配存储空间并初始化：

```
int pi* = new int (6); // *pi 设为 6
double *pd=new double(99.99) // *pd 设为 99.99
```

概括：设 T 为标量类型，则分配存储空间并初始化语法如下：

T *指针名=new T(初始值);

2. 初始化常规结构、数组（C++11 大括号列表初始化器）

```
struct where{double x;double y;double z;};
where *one=new where{31.13,26.00,82.2228};
int *ar=new int[4]{2,4,6,8};
```

该方法也可用于单值变量

3. new 失败时引发**异常** `std::bad_alloc`

4. new: 运算符、函数和替换函数

运算符 `new` 和 `new[]` 分别调用以下函数

```
void *operator new(std::size_t);
void *operator new[](std::size_t);
```

以上函数是**分配函数**，位于全局名称空间中。

运算符 `delete`、`delete[]` 调用**释放函数**：

```
void operator delete(void *);
void operator delete[](void *);
```

`std::size_t` 为 typedef。

`int *pi=new int;` 被转换为 `int *pi=new(sizeof(int));`

`int *pa=new int[40];` 被转换为 `int *pa=new(40*sizeof(int));`

C++ 中这些函数是**可替换的**，可进行定制（定制后使用定制的版本）。

5. **定位 new 运算符**

头文件：`new`

用法：`new(用于分配空间的地址) 类型;`

`new(用于分配空间的地址) 类型[分配的个数];`

```
struct chaff
{
```

```

    char dross[20];
    int slag;
};
char buffer1[50];
char buffer2[500];
chaff *p1, *p2;
int *p3, *p4;
p1 = new chaff; // place structure in heap
p3 = new int[20]; // place int array in heap
p2 = new (buffer1) chaff; // place structure in buffer1
p4 = new (buffer2) int[20]; // place int array in buffer2

```

以上代码中 new 在 buffer1 中划出空间给一个 chaff 结构, 在 buffer2 中划出空间给 20 个 int. 定位 new 不需要 delete, 定位 new 可以重载

- 其他形式的定位 new

标准定位 new 调用一个接受两个参数的 new() 函数

```

int * pi = new int; // invokes new(sizeof(int))
int * p2 = new(buffer) int; // invokes new(sizeof(int), buffer)
int * p3 = new(buffer) int[40]; // invokes new(40*sizeof(int), buffer)

```

定位 new 运算符不可替换, 其至少接受两个参数, 第一个总为 std::size_t, 指定了请求的字节数。这些重载函数都被叫做 **定义 new**。

十一、名称空间

- **声明区域**: 可以在其中进行声明的区域。全局变量的声明区域为其声明所在的文件, 函数中声明的变量声明区域为其声明所在的代码块。
- **潜在作用域**: 变量的潜在作用域从声明点开始, 到声明区域的结尾。
- 变量对程序而言可见的范围叫 **作用域**。
- 声明 **名称空间** (使用 namespace)

```

namespace 名称空间名
{
    //各类声明
}

```

名称空间可以是全局的, 也可以位于另一个名称空间 (名称空间可以嵌套), 不能位于代码块中。默认情况下名称空间中的名称链接性为外部。

- **全局名称空间**: 对应文件级声明区域, 全局变量位于全局名称空间中

任何名称空间中的名称都不会与其他名称空间中的名称发生冲突。

可以这样使用名称空间:

```

namespace grade_9
{
    namespace segment_B
    {
        namespace class_15
        {
            void clean();
            .....
        }
        .....
    }
    .....
}

```

```

}
namespace grade_7
{
    .....
}
.....
namespace grade_8
{
    namespace segment_B
    {
        namespace class_15
        {
            void clean()
            {
                .....
            }
        }
    }
}

```

- 访问名称空间：使用作用域解析运算符::

如：`grade_9::segment_B`、`grade_9::segment_B::class_15::clean()` 即 `名称空间名::名称` 未被装饰的名称(`clean()`)叫做**未限定的名称**，包含名称空间的名称叫**限定的名称**。

1. using 声明：格式 `using 限定的名称;`

using 声明将特定的名称添加到它所属的声明区域中。完成声明后，就可以用未限定的名称代替限定的名称。如 `using std::cout;` 可以让 `cout` 代替 `std::cout`

2. using 编译指令：格式 `using namespace 名称空间;`

using 编译指令使名称空间中所有的名称都可用。

3. 无论是否使用 using, 都可以使用限定的名称。

4. 可以在其他名称空间中使用 using.

```

namespace myth
{
    using Jill::fetch;
    using namespace elements;
    using std::cout;
    using std::cin;
}

```

在以上示例中，导入名称空间 `myth` 的同时也会导入 `elements`, `fetch` 即在名称空间 `Jill` 中也在名称空间 `myth` 中。

5. 给名称空间创建别名

namespace 别名=原名称空间名;

该方法可以用来简化对嵌套名称空间的使用：

```

namespace jjyz_g9c15=grade_9::segment_B::class_15;
jjyz_g9c15::introuduce();

```

此时 `jjyz_g9c15` 是 `grade_9::segment_B::class_15` 的别名。

6. 未命名名称空间：效果与链接性为内部的静态变量相似。潜在作用域为从声明点到该声明区域末尾。如：

```
static int counts;
```

```
namespace
{
    int counts;
}
```

两种方式效果等价。

第四部分 OOP 和类设计、类的应用 第一小节

零、绪论：OOP 的重要特性：**抽象**、**封装**和**数据隐藏**、**多态**、**继承**、代码的可**重用性**

1. **用户定义类型**：实现抽象接口的类设计

2. 指定类型需要完成的工作：

- 决定数据对象需要的内存数量
- 决定内存中每一位的含义（作用）
- 决定对数据对象可执行的操作、方法。

3. **类规范**的组成：

- **类声明**：**数据成员**描述数据部分，**成员函数（方法）**描述**公有接口**，一般在头文件中
- **类方法定义**：类成员函数的实现，一般在源代码文件中。

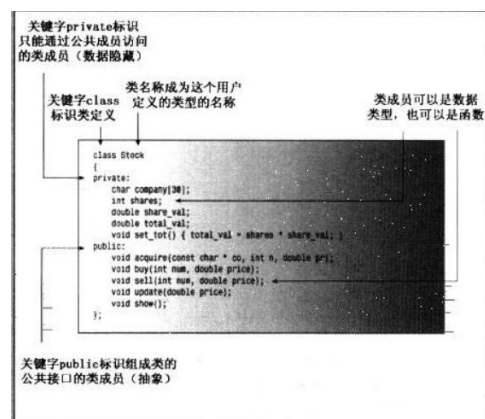
一、类的声明：

```
class 类名
{
    private:
        私有成员
    public:
        公有成员
};
```

在此处不能用 typename 代替 class.class 指出要声明类。

例子：

```
class student
{
    private:
        int class_;
        int grade;
        string name;
        string school;
    public:
        void self_introduce();
        student get_info();
};
```



此时可以创建 student 类型的变量，叫做**类对象**或**实例**。（如 student A;）

成员函数可以就地定义，也可以用原型表示。

4. **访问控制**：使用类对象的程序都可以直接访问公有部分(public 标签下的部分), 私有成员(private 标签下的部分)只能用公有的成员函数访问，外部无法直接访问，是数据隐藏。

5. **类对象的默认访问控制是 private**, 因此可以省略 private, 而**结构**也可以拥有成员函数, 其**默认访问类型是 public**.

6. 实现类成员函数：使用作用域解析运算符。

类成员函数的限定名 **所属类名::成员函数名**

如: `student::self_introduce()` 表示 `student` 类中的 `self_introduce()` 成员函数
成员函数的函数定义:

```
返回类型 限定名(形参列表)//或 返回类型 所属类名::成员函数名(形参列表)
{
    定义
}
```

此时 `self_introduce()` 具有**类作用域**, 其他成员函数可以直接访问它。

成员函数可以访问类的私有成员。

7. **定义位于类声明中的函数自动成为内联函数。**在类声明之外定义内联成员函数只需在实现部分中定义函数时使用 `inline` 限定符即可。每个使用内联函数的文件中都要对其进行定义。

8. **改写规则:** 在类的声明中定义方法等同于用原型替换方法定义并在类声明后定义改写为内联函数。

9. 使用对象

创建对象 类名 对象名;

使用成员 对象名. 成员名;

调用成员函数时, 它将会使用被用来调用它的对象的数据成员。

每个对象都有自己的存储空间, 存储内部变量和类成员, **同一类所有的类对象共享同一组方法**, 每种方法只有一个。

所有结构能做的事情类都能做, 如把一个对象赋给同类型的另一个对象。

通常数据成员放在私有部分, 成员函数(方法)放在公有部分。

二、构造函数、析构函数

1. **构造函数**在创建对象时调用, 可以自动初始化对象

2. 构造函数名是类的名称, 没有返回值(不是 `void`)

3. `cerr`(在 `iostream` 中)流输出错误信息。用法同 `cout`。

4. 调用构造函数:

类名 对象名=构造函数名(参数); //显示

类名 对象名(参数); //隐式

在使用 `new` 时调用 类名* 指针=new 类名(构造函数参数);

注意: 不能将类成员名用作构造函数的参数名。

构造函数用来创建对象, 不能通过对象调用。

5. **默认构造函数:** 构造函数名() {定义} 当且仅当没有初始值时调用

当且仅当未定义构造函数时编译器才会提供默认的构造函数(隐式版本)。即

构造函数名() {} 自动生成, 不做任何事情。

在为类定义非默认构造函数后, 而没有定义默认构造函数时, 不提供初始值会出错。

可以有多个构造函数, 只有一个默认构造函数。

隐式调用默认构造函数时, 不要使用圆括号!

6. **析构函数:** 对象过期时, 程序自动调用析构函数进行清理工作。其在类名前加上 `~`, 没有参数, 没有返回值, 若没有提供析构函数, 编译器提供默认析构函数。

7. `const` 成员函数:

声明类只描述了对象的形式, 并没有创建对象, 所以不能直接使用 `const` 在类定义中声明常量。若一个成员函数不会改变数据, 声明如下:

返回类型 函数名(形参列表) `const`;

原书实例:

stock10.h

```
// stock10.h Stock class declaration with constructors, destructor added
#ifndef STOCK1_H_
```

```

#define STOCK1_H_
#include <string>
class Stock
{
private:
    std::string company;
    long shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    Stock();           // default constructor
    Stock(const std::string & co, long n = 0, double pr = 0.0);
    ~Stock();          // noisy destructor
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show();
};
#endif

```

stock10.cpp

```

// stock1.cpp Stock class implementation with constructors, destructor added
#include <iostream>
#include "stock10.h"
// constructors (verbose versions)
Stock::Stock()           // default constructor
{
    std::cout << "Default constructor called\n";
    company = "no name";
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}
Stock::Stock(const std::string & co, long n, double pr)
{
    std::cout << "Constructor using " << co << " called\n";
    company = co;

    if (n < 0)
    {
        std::cout << "Number of shares can't be negative; "
                    << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
}

```

```
    share_val = pr;
    set_tot();
}
// class destructor
Stock::~Stock()           // verbose class destructor
{
    std::cout << "Bye, " << company << "!\n";
}
// other methods
void Stock::buy(long num, double price)
{
    if (num < 0)
    {
        std::cout << "Number of shares purchased can't be negative. "
                    << "Transaction is aborted.\n";
    }
    else
    {
        shares += num;
        share_val = price;
        set_tot();
    }
}
void Stock::sell(long num, double price)
{
    using std::cout;
    if (num < 0)
    {
        cout << "Number of shares sold can't be negative. "
              << "Transaction is aborted.\n";
    }
    else if (num > shares)
    {
        cout << "You can't sell more than you have! "
              << "Transaction is aborted.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}
void Stock::update(double price)
{
    share_val = price;
```

```

    set_tot();
}
void Stock::show()
{
    using std::cout;
    using std::ios_base;
    // set format to #.###
    ios_base::fmtflags orig =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    std::streamsize prec = cout.precision(3);
    cout << "Company: " << company
        << " Shares: " << shares << '\n';
    cout << " Share Price: $" << share_val;
    // set format to #.##
    cout.precision(2);
    cout << " Total Worth: $" << total_val << '\n';
    // restore original format
    cout.setf(orig, ios_base::floatfield);
    cout.precision(prec);
}

```

usestok1.cpp

```

// usestok1.cpp -- using the Stock class
// compile with stock10.cpp
#include <iostream>
#include "stock10.h"
int main()
{
    {
        using std::cout;
        cout << "Using constructors to create new objects\n";
        Stock stock1("NanoSmart", 12, 20.0);           // syntax 1
        stock1.show();
        Stock stock2 = Stock ("Boffo Objects", 2, 2.0); // syntax 2
        stock2.show();

        cout << "Assigning stock1 to stock2:\n";
        stock2 = stock1;
        cout << "Listing stock1 and stock2:\n";
        stock1.show();
        stock2.show();

        cout << "Using a constructor to reset an object\n";
        stock1 = Stock("Nifty Foods", 10, 50.0);      // temp object
        cout << "Revised stock1:\n";
        stock1.show();
        cout << "Done\n";
    }
}

```

```

}
// std::cin.get();
return 0;
}

```

Using constructors to create new objects

Constructor using NanoSmart called

Company: NanoSmart Shares: 12

Share Price: \$20.00 Total Worth: \$240.00

Constructor using Boffo Objects called

Bye, Boffo Objects!

<<---部分编译器可能输出这行

Company: Boffo Objects Shares: 2

Share Price: \$2.00 Total Worth: \$4.00

Assigning stock1 to stock2:

Listing stock1 and stock2:

Company: NanoSmart Shares: 12

Share Price: \$20.00 Total Worth: \$240.00

Company: NanoSmart Shares: 12

Share Price: \$20.00 Total Worth: \$240.00

Using a constructor to reset an object

Constructor using Nifty Foods called

Bye, Nifty Foods!

Revised stock1:

Company: Nifty Foods Shares: 10

Share Price: \$50.00 Total Worth: \$500.00

Done

Bye, NanoSmart!

Bye, Nifty Foods!

• `std::cout` 的 `precision()` 方法返回一个 `std::streamsize` 类型的值，为修改前显示小数位的值。随后可以再次传入 `precision()` 方法恢复原来的设置。

此程序中有一对大括号，确保析构函数在 `main()` 结束前调用。

• `Stock stock2=Stock("Boffo Objects",2,2.0);`

该语句可能与初始化的另一种语法完全相同（只操作 `stock2`，不生成其他对象），也可能调用构造函数构造一个临时对象，再复制给 `stock2`。

在默认的情况下，给类对象赋值时，将把一个对象的成员复制给另一个。

构造函数也可以赋新值给对象：

`stock1=Stock("Nifty Foods",10,50.0);`

由于 `stock1` 已存在，所以此时创建一个 `Stock` 临时对象后赋给 `stock1`。

• 自动变量放在栈中，最后创建的对象最先删除，最先创建的变量最后删除。

note: 在进行初始化时，最好选择 `Stock stock2("Boffo Objects",2,2.0);` 的方式，因为 `Stock stock2=Stock("Boffo Objects",2,2.0);` 可能产生临时对象（或不产生）。

• 可以将 C++11 列表初始化的语法运用到类，即：

类名 对象名={初始化列表}; 或 类名 对象名{初始化列表};

初始化列表相当于原来的参数列表，空括号表示使用默认构造函数。

• 使用 `const` 对象时只能调用 `const` 成员函数，在函数原型和定义中都要在括号后加上 `const`

三、`this` 指针：1. 每一个类方法都有一个 `this` 指针，指向调用它的对象的地址。

2. `this` 指针被作为隐藏参数传递给方法。

3. ->运算符同样适用于访问类成员。

4. 对象数组：与普通数组差别不大，可以使用大括号并调用构造函数使用。

四、类作用域

- 在类中定义的名称作用域都为整个类。

1. 作用域为类的常量：由于 const 变量需要初始化，而声明类只是描述了对象的形式，没有创建对象。因此在创建对象前，没有用于存储 const 的空间。

(1) 声明为枚举 格式：enum{符号=值(整数)}；

声明枚举不会创建类数据成员，因为此处没有枚举类型名。

编译器在遇到符号时替换为对应的值。

(2) 使用 static const 类型 变量名=值；

相当于静态常量，与其他静态变量存储在一起

[C++11]2. 作用域内枚举：作用域为类

(1) 传统枚举中两个枚举定义的枚举量可能发生冲突

(2) 声明格式：enum class 枚举类型{枚举量}； 可以用 struct 代替 class

(3) 无论如何该方式都要用枚举名来限定枚举量，格式 枚举名::枚举量

(4) 枚举的底层类型默认为 int, C++11 可以指定底层类型，语法：

enum class:类型 枚举名{枚举量}；

(5) 作用域内枚举不能隐式转换为整型。

五、运算符重载：使用运算符函数，属于一种形式的 C++ 多态

1. 格式 operator 运算符(形参列表)；

2. 只能重载有效的 C++ 运算符，不能虚构。

3. 当其作为一个成员函数时，运算符前是调用对象，运算符后是运算符函数的参数。

如：sid+sara; 种 sid 对象所属的类有重载+，则它等价于 sid.operator+(sara);

拓展：将两个以上的对象相加：t4=t1+t2+t3; +是从左向右的运算符

此时该调用是如下的形式：t4=t1.operator+(t2.operator+(t3)); 所以该操作可行。

4. 重载运算符时至少有一个操作数是用户定义的类型

5. 使用时不违反原句法规则，如不改变操作数个数

6. 不能改变运算符优先级，不能创建新的运算符

7. 可以重载的运算符：(不一定要通过成员函数重载运算符)

Table 11.1 Operators That Can Be Overloaded

+	-	*	/	%	^
&		~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	>>=
<<=	==	!=	<=	>=	&&
	++	--	,	->*	->
()	[]	new	delete	new []	delete []

只能通过成员函数重载的运算符：

= 赋值 () 括号 [] 中括号 -> 间接成员访问运算符

8. 在类外使用运算符函数时，要有多个参数，运算符前后都会作为参数。

如：c=a+b; 语句种 a 和 b 和 c 均为对象，在当前的作用域中(不在类中)定义的 operator+() 接受两个参数。

六、友元

1. 友元包括：友元函数、友元类、友元成员函数

note: 为什么需要友元

在为类重载二元运算符时，由于左侧操作数是调用对象，若左侧为内置的类型右侧为用户定义类型，则无法进行调用。非成员函数不由对象调用，它使用的所有值都是显式调用的。非成员运算符函数的第一个参数为运算符左侧操作数，第二个参数为运算符右侧操作数。

2. 通过让函数成为类的友元，可以赋予其与类成员函数相同的访问权限，它不是成员函数（不作为类的一部分），不可以成员运算符调用，但可以访问私有组件。
3. 创建：在类中声明原型并在原型前加上 friend，定义时无需加上 friend，且无需使用类的限定方法。
4. 非成员友元函数的应用：为类重载运算符时将非类的项作为其第一个操作数时反转操作数顺序。

▲△实用方法：重载<<运算符

5. ostream 重载的<<运算符要求左边是一个 ostream 对象，因此，ostream::operator<<() 实现返回一个 ostream&。cout 是 ostream 对象，cout<<x 返回指向 cout 的引用，使得 cout<<x<<y; 能够使用。

6. 类的继承属性可以让 ostream& 指向 ostream 和 ofstream

7. cerr 是 ostream 对象，将输出发送到标准错误流（默认为显示器）

8. 调用时应使用 cout 本身（或其他 ostream 及其子类的对象），所以该运算符函数要接受引用，且要保证可以像 cout 一样可以连续调用，则万用模板如下：

```
ostream& operator<<(ostream& os, 要输出的类型& Object)
{
    //输出
    return os;
}
```

注意：该函数不是 ostream 的友元，因为它整体的使用 ostream 对象，不能访问私有成员。

9. 状态成员：描述对象所处的状态，一般用枚举来表示
 10. cmath 数学库：atan(y,x) 反正切函数（已知正切值求角度）、sin(A) 正弦、cos(A) 余弦、atan2(y,x) 反正切函数
 11. cstdlib 通用库：rand() 函数返回一个从 0 到某个值之间的随机数（伪随机数）
srand() 覆盖默认的种子值，通常用当前时间来覆盖。
 12. ctime 时间库：time() 函数接收一个 time_t 的地址，将时间放到该变量里，并返回它。
- NULL 宏表示空指针（C++11 有关键字 nullptr 表示空指针）
- 随机数种子的生成 srand(time(NULL));
- 随机数的生成 rand()%需要的最大随机数加一；
- 如：生成 0~7 的随机数 a=rand()%8;

七、类的自动转换和强制类型转换

1. C++ 会自动转换兼容的类型，不兼容的类型可以使用强制类型转换。
 2. 接受一个参数的构造函数为类型转换提供了途径，如 Stonewt Thing=114514.1; 此时会调用 Stonewt::Stonewt(double) 进行隐式转换。
 3. 将用户定义类型的对象初始化为其他类型、将其他类型的值赋给用户定义类型的对象、将其他类型的值传递给接受用户定义类型参数的函数、返回用户定义类型的函数试图返回其他类型时可进行隐式转换。若用户定义类型 CLA 提供了转换 T 类型到 CLA 类型的构造函数，且 AT 是可以转换为 T 类型的类型，当且仅当转换不存在二义性时才会将 AT 转为 T 再转为 CLA（进行二步转换）。
如：Stonewt a=10; 此时调用 Stonewt::Stonewt(double)，若声明了 Stonewt::Stonewt(long)，则转换存在二义性，编译器会拒绝这样的转换。（因为 int 可被转为 long 或 double）
 4. 防止意外转换：在前面加上 explicit，关闭隐式转换，此时可以使用强制类型转换
 5. 转换函数：格式 operator 类型();
 - 构造函数只用于从某种类型到类的转换。
 - 要把类转换为某种类型需要用转换函数，它属于特殊的运算符函数。
 - 该函数必须为类方法，不能指定返回类型，不能有参数。
 - 在上下文无法指明隐式转换的类型且转换存在二义性是，编译器无法接受这些语句。
- 如：cout<<thing<<endl; 此时 thing 是 Thing 类型的对象，且定义了 Thing::operator X() 和

Thing::operator Y(), 则编译器不知道要转换为类型 X 还是 Y。如果只有 Thing::operator X(), 则 thing 被转为 X 类型。

- 可以使用显式强制类型转换来指出使用那个转换函数
- C++11: 可以将转换函数声明为显式的。即在前面加上 explicit. 此时只能由强制转换调用转换函数, 而不能自动转换。

6. 转换函数与友元函数:

定义转换函数 Thing::operator double() 后, 声明 double 值 a, Thing 类对象 thing, Thing 类有构造函数 Thing::Thing(double), 则此时执行 b=a+thing; 时并不会把 a 转为 Thing 类型, 而是把 thing 转为 double 再相加。若此时定义友元函数可以把 a 作为参数, 既可以让其被隐式转换, 也可以让其直接作为参数传递。

八、动态内存分配和类

原书实例: StringBad 类

strngbad.h

```
// strngbad.h -- flawed string class definition
#include <iostream>
#ifndef STRNGBAD_H_
#define STRNGBAD_H_
class StringBad
{
private:
    char * str;           // pointer to string
    int len;              // length of string
    static int num_strings; // number of objects
public:
    StringBad(const char * s); // constructor
    StringBad();               // default constructor
    ~StringBad();              // destructor
// friend function
    friend std::ostream & operator<<(std::ostream & os,
                                     const StringBad & st);
};
#endif
```

strngbad.cpp

```
// strngbad.cpp -- StringBad class methods
#include <cstring>           // string.h for some
#include "strngbad.h"
using std::cout;
// initializing static class member
int StringBad::num_strings = 0;
// class methods
// construct StringBad from C string
StringBad::StringBad(const char * s)
{
    len = std::strlen(s);           // set size
    str = new char[len + 1];        // allot storage
    std::strcpy(str, s);             // initialize pointer
```

```

    num_strings++; // set object count
    cout << num_strings << ": \"" << str
         << "\" object created\n"; // For Your Information
}
StringBad::StringBad() // default constructor
{
    len = 4;
    str = new char[4];
    std::strcpy(str, "C++"); // default string
    num_strings++;
    cout << num_strings << ": \"" << str
         << "\" default object created\n"; // FYI
}
StringBad::~StringBad() // necessary destructor
{
    cout << "\"" << str << "\" object deleted, "; // FYI
    --num_strings; // required
    cout << num_strings << " left\n"; // FYI
    delete [] str; // required
}
std::ostream & operator<<(std::ostream & os, const StringBad & st)
{
    os << st.str;
    return os;
}

```

语句 `int StringBad::num_strings = 0;` 的作用是将静态成员 `num_strings` 的值初始化为零。不能在类声明中初始化 **静态成员变量**，因为声明描述了如何分配内存，但并不分配内存。访问类静态成员的方法 `类名::静态成员名`

静态类成员单独存储，不是类的组成部分。初始化语句使用了作用域运算符，指出了类型，并没有使用关键字 `static`。如果静态成员是 `const` 整型或枚举型，则可以在类声明中初始化。

vegnews.cpp

```

// vegnews.cpp -- using new and delete with classes
// compile with strngbad.cpp
#include <iostream>
using std::cout;
#include "strngbad.h"
void callme1(StringBad &); // pass by reference
void callme2(StringBad); // pass by value
int main()
{
    using std::endl;
    {
        cout << "Starting an inner block.\n";
        StringBad headline1("Celery Stalks at Midnight");
        StringBad headline2("Lettuce Prey");
        StringBad sports("Spinach Leaves Bowl for Dollars");
    }
}

```

```

    cout << "headline1: " << headline1 << endl;
    cout << "headline2: " << headline2 << endl;
    cout << "sports: " << sports << endl;
    callme1(headline1);
    cout << "headline1: " << headline1 << endl;
    callme2(headline2);
    cout << "headline2: " << headline2 << endl;
    cout << "Initialize one object to another:\n";
    StringBad sailor = sports;
    cout << "sailor: " << sailor << endl;
    cout << "Assign one object to another:\n";
    StringBad knot;
    knot = headline1;
    cout << "knot: " << knot << endl;
    cout << "Exiting the block.\n";
}
cout << "End of main()\n";
// std::cin.get();
return 0;
}

void callme1(StringBad & rsb)
{
    cout << "String passed by reference:\n";
    cout << "    \"" << rsb << "\"\n";
}

void callme2(StringBad sb)
{
    cout << "String passed by value:\n";
    cout << "    \"" << sb << "\"\n";
}

```

控制台窗口<stdin><stdout>

```

Starting an inner block.
1: "Celery Stalks at Midnight" object created
2: "Lettuce Prey" object created
3: "Spinach Leaves Bowl for Dollars" object created
headline1: Celery Stalks at Midnight
headline2: Lettuce Prey
sports: Spinach Leaves Bowl for Dollars
String passed by reference:
    "Celery Stalks at Midnight"
headline1: Celery Stalks at Midnight
String passed by value:
    "Lettuce Prey"
"Lettuce Prey" object deleted, 2 left
headline2: @紕_x0019_
Initialize one object to another:

```



```
sailor: Spinach Leaves Bowl for Dollars
Assign one object to another:
3: "C++" default object created
knot: Celery Stalks at Midnight
Exiting the block.
"Celery Stalks at Midnight" object deleted, 2 left
"Spinach Leaves Bowl for Dollars" object deleted, 1 left
"0_紕_x0019_" object deleted, 0 left
```

该程序修改了已释放的内存导致了 Windows Defender 报毒。

• 按值传递、用一个对象初始化另一个对象可能会将构造函数调用，对象过期后被析构，从而导致指针内容被删除（同一个指针被多个对象（包括临时对象）引用）

▲▲ • C++ 中的 **特殊成员函数**：若没有定义默认提供

默认构造函数（没有定义构造函数时提供）

默认析构函数

复制构造函数

赋值运算符

地址运算符

上述程序的问题由隐式赋值运算符和隐式赋值构造函数引起

• 默认构造函数

不接受任何参数，不执行任何操作。带参数的构造函数也可以作为默认构造函数，需要所有参数都有默认值。只允许由一个默认构造函数。

• 复制构造函数 原型 类名(const 类名&);

用途：将一个对象复制到新创建的对象中（用于初始化过程和按值传递，不是常规赋值操作）

调用条件

① 新建一个对象并初始化为现有对象

当程序生成对象副本时

② 函数按值传递

③ 返回对象

④ 编译器使用临时对象

功能：逐个复制非静态成员（**浅复制**，按值复制），如果该成员为类对象，调用其所属类的复制构造函数

使用时应当进行自定义并使用 **深度复制**。

深度复制：将数据全新地拷贝到其他地方，并将旧的指针（引用）进行更新，防止对旧数据进行误操作。在动态内存分配类中，这可以防止数据被篡改。

• **默认赋值运算符**：原型：

类名& 类名::operator =(const 类名&);

使用：将已有的对象赋给另外的对象，初始化并不一定使用。

赋值运算符的隐式实现也逐个赋值成员，进行浅复制。如果该成员为类对象，调用其所属类的赋值运算符。赋值的实现：释放目标对象以前分配的数据、避免自己给自己赋值、返回指向调用对象的引用。

• 静态成员函数：声明包含 static，不能通过对象调用，无 this，若该成员函数公有，在类外要用限定名访问。静态成员函数不与特定的对象相关联，因此只能使用静态数据成员。

• ▲ 中括号的重载

通常 C++ 的二元运算符（带两个操作数的运算符）位于两个操作数之间。中括号运算符一个操作数

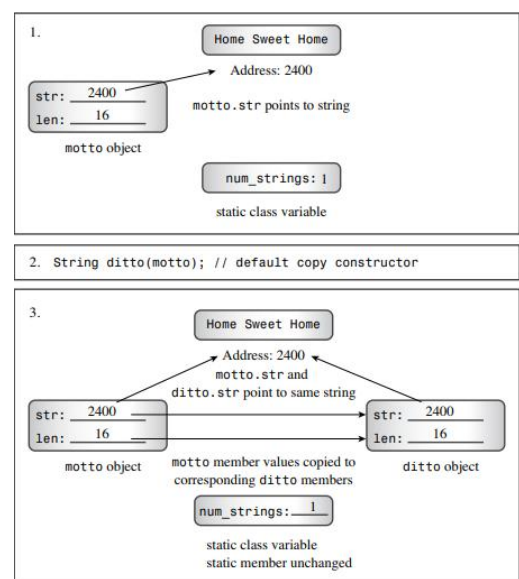


Figure 12.2 An inside look at memberwise copying.

位于左中括号前面，另一个操作数位于中括号内。

重载时可以传入一个下标访问对象的数据，同时为了方便赋值，需要返回引用。如：

```
char& String::operator(int i);
```

若声明了 const 对象，以上的代码将无法使用。重载时 C++ 将区分常量与非常量函数的特征标，则可以提供供 const 对象使用的版本，如：`const char& String::operator[](int i) const;`

- 在构造函数使用 new 的注意事项

-> new 与 delete 配套使用，new[] 与 delete[] 配套使用

-> 应定义复制构造函数和赋值运算符，以进行深度复制

-> 包含类对象成员的类的逐成员复制时，将会调用该成员所属类的复制构造函数进行复制。类中类对象成员进行赋值操作时调用赋值运算符。

- 返回对象

-> 返回指向 const 对象的引用

引用指向的对象应该在调用函数前存在，不能指向函数结束后被删除的变量。

形参出现了 const，注意返回时使用 const

-> 返回指向非 const 对象的引用

重载赋值运算符时要确保能够连续赋值(因为非 const 可转为 const, const 不可转为非 const)

-> 返回对象：

对象是被调用函数中的局部变量，则不应该返回引用。因为局部变量在函数执行完毕后会调用对象的析构函数，导致变量被析构，引用指向它无法使用。

常见应用：算术运算符函数

-> 返回 const 对象

以上的算术运算符函数由于返回对象，调用了复制构造函数创造了临时对象表示返回值。则此时可能会使如 `force1+force2=force3` 的语句成立(3 个变量均为对象，所属类定义了 `operator+`)，当类有定义 `operator==()` 时，少写一个 `=` 便会导致意外。

-> 总结

返回局部对象应返回对象，此时会使用复制构造函数生成返回的对象。

返回没有公有复制构造函数的类(如 `ostream`)，必须返回指向对象的引用

- 使用指向对象的指针：使用 new 初始化对象：

若 `Class_name` 是类，`value` 的类型为 `Type_name`，则

```
Class_name * pclass=new Class_name(value);
```

调用如下构造函数：`Class_name(Type_name);`

不存在二义性时发生由原型匹配导致的转换。下面的初始化使用默认构造函数：

```
Class_name *ptr=new Class_name;
```

- 析构函数调用条件

1. 动态变量：当执行完定义该对象的程序块时，将调用该对象的析构函数。
2. 静态变量：程序结束时调用析构函数
3. new 创建的对象：显式使用 delete 删除对象时调用。

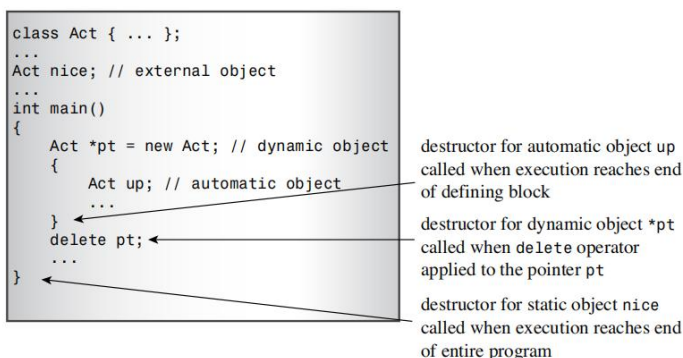


Figure 12.4 Calling destructors.

- 定位 new 与类对象

1. 定位 new 运算符分配空间是根据指定的地址来分配的，若两次分配都用相同的地址分配，会导致第二次创建的对象覆盖第一次创建的对象。

解决：向后偏移相应的内存单元个数。

注：1 个内存单元对应 1Byte, char 类型占用空间正好为 1Byte.

2. 定位 new 运算符不能保证对象析构函数的

调用，需要显式调用！

警告：必须自行负责管理定位 new 运算符分配的对象，显式调用析构函数!!!!

九、抽象数据类型 (Abstract Data Type, ADT)

- 基本常识

栈：在同一段添加和删除，后进先出 (LIFO, last-in, first-out)

队列：在队尾添加项目，在队首删除项目，先进先出 (FIFO, first-in, first-out)

链表：链表由节点序列构成。每一个节点种都包含要保存到链表的信息和指向下一个节点的指针。
链表节点结构示例：（这是一个单向链表）

```
struct Node
{
    Item item;
    struct Node *next;
}
```

单向链表的每个节点都只包含一个指向其他节点的指针，知道第一个节点的地址就可以知道每个节点的地址。最后一个节点指向的地址会被设为 NULL(0)，指出后面没有节点。

1. 嵌套类 (结构)

嵌套类的作用域为类，若 Node 结构在 Queue 的公共部分声明，则在类外面可以声明 Queue::Node 类型的类。

▲2. 成员初始化列表

- 调用构造函数时，对象在括号中的代码执行之前被创建
- const 成员在创建对象时必须初始化，成员初始化列表可以在创建对象时初始化成员。
- 语法：

c_name::c_name(形参列表):成员 0(值 0),成员 1(值 1),.....

每个**初始化器**由逗号分隔，若数据成员的名称为 member，要初始化为 val，则初始化器为 member(val)

初始化的值可以是常量或构造函数参数列表的参数，并且可以初始化 const 成员。

- 注意：只有构造函数可以使用该语法，对于 const 和引用成员必须使用该语法初始化
- 数据成员被初始化的顺序与他们出现在类声明中的顺序相同，与初始化器的排列顺序无关。

3. (C++11) 类内初始化

```
class A
{
    int mem1=10;
    const int mem2=20;
};
```

与在构造函数中使用成员初始化列表等价。实际的列表可以覆盖这些默认值。

A::A(int n):mem1(n);

4. 如果不想将类对象进行复制，可以定义伪私有方法，即把复制构造函数和赋值运算符声明为私有。可以避免自动生成的默认方法定义，且不能被广泛使用。

- RAND_MAX 是 rand() 函数可返回的最大值，0 是最小值。

十、类的公有继承、is-a

- 基本概念

1. 定义：从已有的类派生出新的类，**派生类**继承了原有类（**基类**）的特征。这种方法叫做**类继承**。派生类也叫**子类**，基类也叫**父类**。

2. 公有派生

声明：

```
class 派生类:public 公有基类
{
    .....
}
```

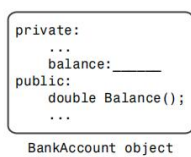
特点:

- 派生类对象包含基类对象
- **基类的公有成员将成为派生类的公有成员**
- **基类的私有部分会成为基类的一部分，但只能通过基类的公有和保护方法访问。**

->派生类继承了基类的实现,派生类对象可以使用基类的方法

使用:

- 派生类需要自己的构造函数,需要添加额外的数据成员和成员函数。
- 派生类的构造函数需要给继承的成员提供数据。



```
class Overdraft : public BankAccount {...};
```

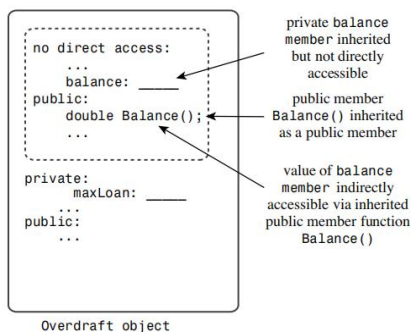


Figure 13.1 Base-class and derived-class objects.

①. 派生类的构造函数

- 创建派生类对象时,程序先创建基类对象
- 基类对象在程序进入派生类构造函数时前创建,派生类构造函数必须通过成员初始化列表将基类信息传递给基类构造函数。

如:A 是 OBJ 的子类,则要初始化 OBJ 中定义的成员时,应这样写:A(...):OBJ(...){...}

初始化器中要调用父类的构造函数,如果不在初始化器中调用父类构造函数,则调用父类的默认构造函数。

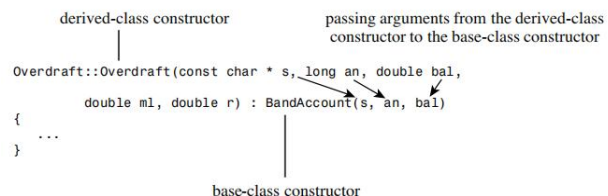


Figure 13.2 Passing arguments through to a base-class constructor.

注意: 必须首先创建基类对象,不调用基类的构造函数时程序将使用默认构造函数。

- 派生类构造函数应初始化新的成员。
- * • 除虚基类以外类只能将值传递回相邻基类的构造函数

②派生类的析构函数

派生类对象过期时,先调用派生类的析构函数,再调用基类的构造函数。

③指针、引用的特殊关系

基类指针可以在不进行显式类型转换的情况下指向派生类对象

基类引用可以在不进行显式类型转换的情况下指向派生类对象

例子:

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
TableTennisPlayer & rt = rplayer;
TableTennisPlayer * pt = &rplayer;
rt.Name(); // invoke Name() with reference
pt->Name(); // invoke Name() with pointer
```

*此处 RatedPlayer 是 TableTennisPlayer 的派生类。

基类指针或引用只能用于调用基类方法!

要注意:这是单向的,不能够将基类对象和地址赋给派生类引用和指针。

④. 可以将派生类对象赋给基类对象,此时调用隐式的重载赋值运算符。因为 operator=() 接受一个基类的引用,可以把派生类的基类部分复制到别的对象。

3. is-a 关系的继承

• C++的 3 种继承关系: **公有继承**、**保护继承**、**私有继承**

• 公有继承建立 is-a 关系(实际上为 is-a-kind-of 关系). 其内容如下

派生类也**是一个**基类对象, 可对基类对象执行的任何操作, 也可以对派生类执行。

• has-a 关系: 离散数学可能用到代数, 但是不能因为这个原因将离散数学认作是代数的分支。通常只会引用代数中的方法, 而不是直接纳入其体系。公有继承不建立 has-a 关系

• 公有继承不建立 is-like-a 关系。继承可以在基类的基础上添加属性, 但不可以删除基类的属性

• 公有继承不建立 is-implemented-as-a(作为……来实现)关系, 如不能从数组类派生出栈类, 虽然数组可以实现栈

• 公有继承不建立 uses-a 关系

十一、多态公有继承

• 若一个方法在基类和派生类中不同, 方法的行为取决于调用方法的对象, 则该行为称作多态。实现多态有两种方法: 在派生类中重新定义基类的方法、使用虚方法

• is-a 关系通常不可逆

• 若基类为 base_c, 派生类为 depend_c, 方法 method() 在两个类中有不同的定义, 则基类中的限定名为 base_c::method(), 派生类中的为 depend_c::method()。在两个类中行为一致的方法, 只需在基类中声明。

1. 虚方法(使用 virtual)

格式 virtual 函数原型;

(1) 没有使用关键字 virtual 时, 程序根据引用或指针类型选择方法, 如

```
base_c b();
depend_c d();
base_c& rb=b;
base_c& rd=d;
base_c* pb=&b;
base_c* pd=&d;
rb.method();
rd.method();
```

当 method() 方法不是虚方法时, 两个调用都使用 base_c::method();

若使用了 virtual 关键字, 程序根据指针或引用指向的对象的类型来选择方法, 这里为 rb.method() 调用 base_c::method(), rd.method() 调用 depend_c::method()

(2) 经常将派生类中要重新定义的方法声明为虚方法。在基类中声明虚方法后, 该方法在派生类中自动成为虚方法(亦即派生类中可以不使用 virtual 关键字)。

(3) 基类中需声明虚析构函数, 确保释放对象时, 按正确的顺序调用析构函数。

(4) 关键字 virtual 只用于类声明的方法原型中, 而不是方法定义中

(5) 派生类调用基类方法: 若是虚的, 要加限定名, 使用作用域解析运算符调用, 否则——将会调用自己的方法。若不是虚的, 派生类没有重新定义, 直接调用。

(6) 需要虚析构函数的原因: 若析构函数不是虚的, 则只调用对应于指针类型的析构函数。若没有声明虚析构函数, 则只会有基类的析构函数调用。只有在基类有虚析构函数时, 才可以让对应的对象执行对应派生类的析构函数, 再自动执行基类的析构函数

十二、静态联编和动态联编

1. 将源代码中的函数调用解释为执行特定的函数代码块称为 **函数名联编(binding)**。

2. 在编译过程中进行联编称为 **静态联编(static binding)**, 亦即 **早期联编, early binding**

3. 虚函数定义会让使用哪个函数无法确定, 须在程序运行时选择正确的虚函数, 该方式为 **动态联编(dynamic binding)**, 亦即 **晚期联编, late binding**。

4. 将派生类引用或指针转换为基类引用或指针被称为 **向上强制转换(upcasting)**, 向上强制转换是

可传递的。相反的过程为**向下强制转换 (downcasting)**,在不使用显式类型转换时,向下强制转换是不允许的。

5. 对于使用基类引用或指针作为参数的函数调用,将进行向上转换。

note:编译器对虚方法使用动态联编,静态联编是默认的联编方式。动态联编会产生其他开销,静态联编则不用其他开销,所以静态联编的效率更高。当没有虚方法时,使用静态联编更合理。

***C++的指导原则其一:不要为不使用的特性付出代价**

十三、▲虚函数的工作原理

处理方法:给每个对象添加一个隐藏成员,其中保存了一个指向函数地址数组的指针。该数组称作**虚函数表 (virtual function table, VTBL)**,其中存储了该类所有虚函数的地址。在基类对象中,虚函数表仅仅包括基类的虚函数的地址,而在派生类对象中,有一个独立于基类虚函数表的虚函数表。若派生类提供虚函数的新定义,该虚函数表将保存新函数的地址,若没有重新定义,将保存函数原始版本的地址。若在派生类中定义了新的函数,则把函数地址添加到虚函数表中。

注意:无论表多大,隐藏成员只是指针,不会因为虚函数太多导致对象太大。

调用:程序查看隐藏成员,找到虚函数表,找到函数。

注意事项:

- 在基类声明的虚函数在其所有的派生类中(包括从派生类中派生出来的类)都是虚的。
- 构造函数不能是虚的
- 析构函数在类作为基类时应声明为虚的。
- 友元不能是虚函数,友元不是类成员,只有类成员才可以是虚函数。
- 如果派生类没有重新定义函数,将使用该函数的基类版本。如果派生类位于派生链中,则使用最新的虚函数版本,除非基类版本的是隐藏的。
- 在派生类中重新定义函数,并不会生成函数的重载版本,而是隐藏基类定义的版本。重新定义继承的方法不是重载,无论参数列表是否相同,所有同名的基类

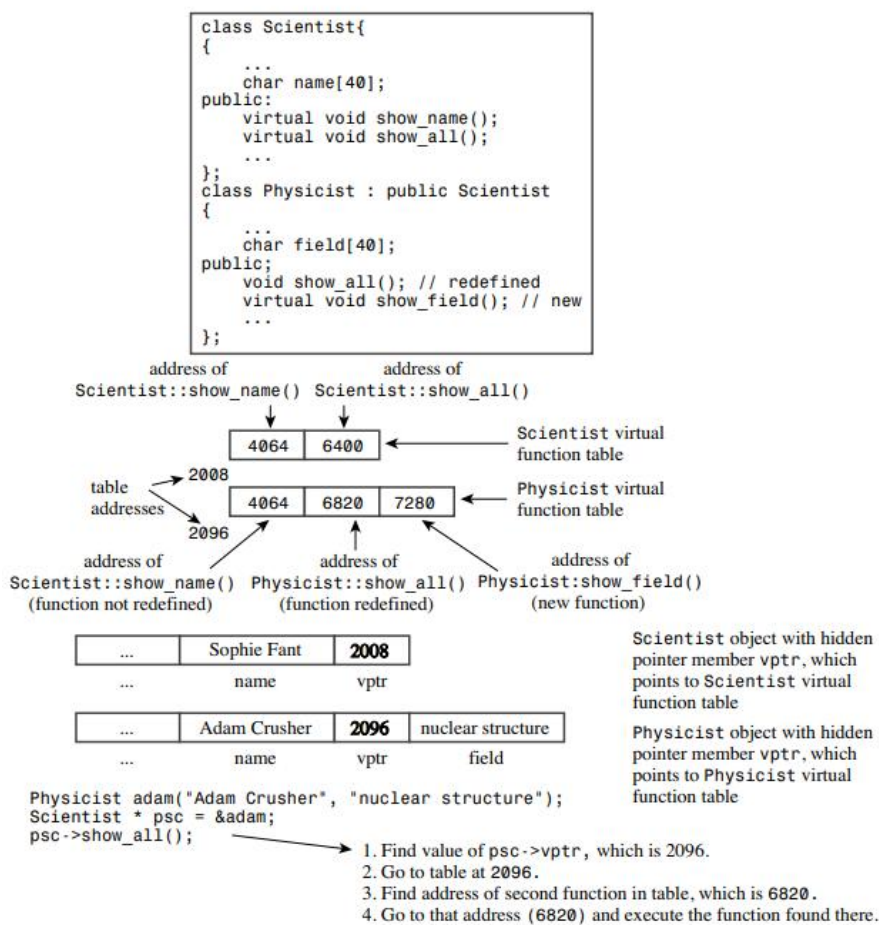


Figure 13.5 A virtual function mechanism.

方法都被隐藏。

总结

- 重新定义继承的方法,应该确保与原来的原型一致。
 - 如果返回类型是基类引用或指针,则可以修改为指向派生类的引用或指针。该特性叫做**返回类型协变 (covariance of return type)**,此时允许返回类型随类的类型变化而变化。
 - 如果基类声明被重载,应在派生类中重新定义所有的基类版本。若只重新定义一个版本,其他的版本被隐藏,派生类对象将无法使用它们。如果不需要修改,新定义可以只调用基类版本。
- 如:基类 overloaded() 函数有 45 个版本,则在派生类中定义 28 个后,其他的只需在函数中调用基类的版本:

```

class Father
{
public:
.....
    virtual void overloaded(int);//version 1
    virtual void overloaded(double,Father&);//version 2
    .....
    virtual Father& overloaded(int,Father[]);//version 44
    virtual void overloaded();//version 45
};
class Son:public Father
...

```

则子类中可以将 version 44 的函数修改为 `virtual Son& overloaded(int,Father[]);`;
若 version 45 没有修改的必要, 则可以这样写:

```
void Son::overloaded() {Father::overloaded();}
```

十四、保护访问控制 protected

- protected 标签用来声明保护成员, 语法像 public 和 private 一样
- 在类外只能用公有成员访问 protected 成员, 派生类的成员可以直接访问保护成员。

十五、抽象基类 (abstract base class, ABC)

- 模型: 从不同的类中抽象出共性, 让一个类拥有这些特性, 并使其作为基类。

如: 正方形、矩形、菱形和平行四边形都是四边形, 则可以定义一个类, 表示四边形。由于特殊四边形可以有自己不同的特性, 矩形的对角线相同, 但是其他四边形不一定, 就需要分别指明两条对角线长, 这就造成了一些冗余。此时定义抽象基类可以解决一些问题。

- 在抽象基类中, **纯虚函数 (pure virtual function)** 提供未实现的函数。纯虚函数原型结尾处为 =0。声明格式如下:

```
virtual 返回类型 函数名(形参列表)=0;
```

- 一个抽象基类至少有一个纯虚函数, 不能创建抽象基类的对象。
- 可以给纯虚函数提供定义
- 从抽象基类派生的类叫做 **具体类 (concreat class)**。

十六、继承和动态内存分配

1. 派生类不使用 new

不需要为为派生类定义显示析构函数、复制构造函数、赋值运算符

- 没有定义析构函数时, 默认提供一个析构函数, 在执行完自身的代码之后调用基类的析构函数。
- 派生类的默认复制构造函数、默认赋值运算符会调用基类的复制构造函数、赋值运算符来操作对象的基类部分。

2. 派生类使用 new

必需要为为派生类定义显示析构函数、复制构造函数、赋值运算符

- 派生类析构函数在执行完自己的代码后自动执行基类的析构函数析构对象的基类部分
- 派生类复制构造函数只能访问派生类成员, 要显式地用初始化成员列表调用基类的复制构造函数来处理对象的基类部分数据。
- 派生类赋值运算符要显式调用基类赋值运算符(通过作用域解析运算符和函数表示法完成)

note: 友元与继承

当基类和派生类都使用了友元时, 派生类无法访问基类的私有成员。此时可以借助基类的友元, 但是要注意类型转换。

创建一个自动内存分配类的步骤

1. 基本框架的设计编写
2. 增加实用常用接口
3. 增加运算符重载
4. 编写复制构造函数、默认赋值运算符、修改特殊成员函数。
5. 调试对象的使用和返回等行为
6. *考虑定位 new 和其他因素

重载<<和>>运算符

```
ostream &operator<<(ostream & os, const class_name&obj)
{
    os<<....
    return os;
}

istream &operator>>(istream &is, class_name&obj)
{
    is>>....
    return is;
}
```

类型转换:

explicit 可以防止隐式转换

从其他到类: 构造函数转换 `c_name(type_name value);`

从类到其他: `operator type_name();`

构造函数使用 new:

- 在构造函数中使用 new 在析构函数中就要使用 delete!
- new 和 delete 配对使用, new[] 和 delete[] 配对使用
- 构造函数应当把指针初始化为 new 分配的地址或空指针 (nullptr, NULL, (void*)0), 不能让析构函数的 delete 释放野指针。
- 定义一个分配内存的复制构造函数, 使得程序能够将一个类对象初始化为另一个类对象。

原型: `className(const className&)`

- 重载赋值运算符 (此处假定用 new[] 分配空间, c_pointer 是 c_name 的成员, 其类型为指向 type_name 的指针)

```
c_name& c_name::operator=(const c_name& cn)
{
    if(this==&cn)
        return *this; //防止自己给自己赋值
    delete[] c_pointer;
    //销毁旧数据, 复制新数据
    c_pointer=new type_name[size];
    //拷贝其他数据
    ...
    return *this;
}
```

1. 编译器生成的成员函数 (特殊成员函数)

- 默认构造函数

默认构造函数没有参数或所有的参数都有默认值。

没有定义构造函数时, 编译器定义默认构造函数。

默认构造函数会自动调用基类的默认构造函数以及本身是对象的成员所属类的默认构造函数
派生类构造函数的成员初始化列表中没有显式调用基类的构造函数时,编译器使用基类的默认构造函数构造对象的基类部分

在定义了某种构造函数后,编译器将不会定义默认构造函数

• 复制构造函数

使用构造函数的情况:

->将新对象初始化为同类对象

->按值将对象传递给函数

->函数按值返回对象

->编译器生成临时对象

在程序没有使用复制构造函数时,编译器提供原型,不提供函数定义

在使用时,新对象的每个成员都被初始化为原始对象相应成员的值。

如果成员为类对象,则初始化该成员时,将使用相应类的复制构造函数

• 赋值运算符

默认赋值为成员赋值。若成员为类对象,会使用所属类的赋值运算符。

编译器不生成将一种类型赋给另一个类型的赋值运算符。

2. 其他类方法

• 构造函数

构造函数创建新的对象,其他类方法由对象调用,所以构造函数不可以继承。

• 析构函数

若一个类要作为基类,则析构函数应声明为虚函数

• 转换

->从参数类型到类类型:调用转换构造函数

在原型中使用 `explicit` 可以禁止隐式转换

->将类类型转换为其他类型:转换函数(如 `operator 类型()`)

转换函数可以是没有参数的类成员函数和返回类型为目标类型的成员函数

[C++11]在转换函数原型中使用 `explicit` 可以禁止隐式转换

• 按值、按引用传递

通常应按引用传递,可以提高效率。按值传递对象会产生临时对象,调用复制构造函数和析构函数。

在函数不修改对象时,应声明为 `const`

• 返回对象、返回引用

返回对象时,会生成副本,较为耗时。返回引用节省时间和内存。

函数不能返回在函数中创建的对象引用。函数在返回函数中创建的临时变量时,要返回对象而不是引用。

函数在返回通过引用或指针传递的对象时,应返回引用。

• ▲使用 `const`

`const` 用来保证方法不修改参数、不修改调用它的对象、引用或指针返回值不能用于修改数据。只有在其他的函数保证不会修改参数时,才能将参数声明为指向 `const` 的指针和引用的函数的参数传递给其他的函数。

3. 公有继承的考虑

• 不可继承的有构造函数、析构函数、赋值运算符。当派生类构造函数没有使用成员初始化列表调用基类构造函数,将使用基类的默认构造函数。

• 赋值运算符

当程序将一个对象付给同类的另一个对象,它将自动为这个类提供一个赋值运算符。

该运算符的默认版本采用成员赋值

如果对象属于派生类,编译器将使用基类赋值运算符来处理派生类对象中的基类部分的赋值

如果成员为另一个对象的类的对象，则使用其所属类的赋值运算符。
可以将派生类对象赋给基类对象

<-->小结

下表中 op=表示+=、*=等格式的赋值运算符

Table 13.1 Member Function Properties

Function	Inherited	Member or Friend	Generated by Default	Can Be Virtual	Can Have a Return Type
Constructor	No	Member	Yes	No	No
Destructor	No	Member	Yes	Yes	No
=	No	Member	Yes	Yes	Yes
&	Yes	Either	Yes	Yes	Yes
Conversion	Yes	Member	No	Yes	No
()	Yes	Member	No	Yes	Yes
[]	Yes	Member	No	Yes	Yes
->	Yes	Member	No	Yes	Yes
op=	Yes	Either	No	Yes	Yes

Table 13.1 Member Function Properties

Function	Inherited	Member or Friend	Generated by Default	Can Be Virtual	Can Have a Return Type
new	Yes	Static member	No	No	void *
delete	Yes	Static member	No	No	void
Other operators	Yes	Either	No	Yes	Yes
Other members	Yes	Member	No	Yes	Yes
Friends	No	Friend	No	No	Yes

第四部分 OOP 和类设计、类的应用 第二小节

零、1. valarray 简介

- 头文件 valarray
- 名称空间 std
- 功能:用于处理数值(或与数值类似)的类,是一个数组。
- 初始化方式:该类为**模板类**,需使用以下语法:

valarray<元素类型> 对象名;

- 方法:

(1)默认构造函数:构造长度为 0 的 valarray 数组

(2)其他构造函数:

valarray<T>::valarray(size_t 长度) 构造指定长度的数组

valarray<T>::valarray(size_t 长度,T 初始值) 构造指定长度的数组,并将初始值设为指定的值

valarray<T>::valarray(数组名,指定长度) 取指定数组的前面几个元素,取的长度为指定长度

valarray<元素类型> 对象名={初始化列表}; //C++11 支持

(3)常用方法

operator[]()	用下标访问元素
size()	返回元素个数(长度)
sum()	求和并返回
max()	最大值
min()	最小值

一、包含**成员对象**的类

1. 不继承接口是 has-a 关系的一个特性

2. 对于继承的对象,在使用成员初始化列表中使用类名调用特定的基类构造函数。对于成员对象则

使用对象名来调用所属类的构造函数。

3. 在构建其他对象之前先构建继承对象的所有成员对象。若省略初始化列表，则会使用成员对象所属类的默认构造函数。

*4. **初始化顺序**: 在初始化列表中, 先被声明的先初始化

二、**私有继承**(可用于实现 has-a 关系)

1. 使用私有继承时, **基类的公有成员和保护成员都变为派生类的私有成员**

2. **包含**和私有继承的区别: (被添加的对象叫**子对象**(subobject))

包含: 将对象作为一个有名称的对象加入类中

私有继承: 将对象作为一个未命名的继承对象添加到类中。

3. 私有继承的声明: 使用关键字 private (不指出关键字时默认为 private)

class 类名: private 基类{ };

4. 可以使用多个基类, 该方式叫做**多重继承(multiple inheritance, MI)**

class 类名: 继承方式 基类#1, 继承方式 基类#2{ };

5. 初始化基类组件

与包含相比, 初始化列表中使用类名来初始化子对象、

6 访问基类的方法: 使用类名和作用域解析运算符

7. 访问基类对象: 使用强制类型转换

8. 访问基类的友元函数: 使用强制类型转换

9. 在私有继承中, 未进行显式类型转换的派生类引用或指针, 无法赋值给基类的引用或指针

10. 私有继承和包含的选用: 通常使用包含, 若需要访问私有成员或重定义虚函数用保护继承。

11. **保护继承**: 使用关键字 protected. 私有继承的变体, 基类的公有和保护成员都将成为派生类的保护成员。

12. 继承总结: 如下表。

Table 14.1 Varieties of Inheritance

Property	Public Inheritance	Protected Inheritance	Private Inheritance
Public members become	Public members of the derived class	Protected members of the derived class	Private members of the derived class
Protected members become	Protected members of the derived class	Protected members of the derived class	Private members of the derived class
Private members become	Accessible only through the base-class interface	Accessible only through the base-class interface	Accessible only through the base-class interface
Implicit upcasting	Yes	Yes (but only the derived class) within	No

其中**隐式向上转换**意味着无需进行显示类型转换, 就可以将基类指针或引用指向派生类对象。

13. 使用 using 重定义访问权限: 在公有部分使用: using 函数限定名; 来达到效果。

如: `using std::valarray<double>::min;`

三、多重继承

1. MI 的主要问题: 从基类中继承同名方法、继承同一祖先类的多个实例

2. 将继承同一祖先的派生类的引用(指针)赋给祖先类引用(指针)时, 由于包含多个实例, 便会导致二义性。

e. g. Student 和 Worker 都派生自 job 类, Student_worker 派生自 Student 和 Worker, 则将 Student_worker 赋给 job 类时要这样操作:

job *a=(*Student) &Student_worker; //将 Student 子对象的 job 对象的地址赋给 a

job *b=(*Worker) &Student_worker; //将 Worker 子对象的 job 对象的地址赋给 b

3. **虚基类**:

- 声明: 将关键字 virtual 与 public 放在一起, 不分顺序

- 作用: 使多个有相同基类派生的类派生出的对象**只有一个相同的基类**

• 派生类的构造函数的使用

对于非虚基类,唯一可以出现在初始化列表的构造函数为基类构造函数。

在基类是虚的时候,不能通过中间类将信息传给基类。若在初始化列表中没有将信息传给基类,则使用基类的默认构造函数。

e. g. Student 和 Worker 虚继承自 job, Student_worker 继承自 Student 和 Worker,则在 Student_worker 类的构造函数中可以直接调用 job 的构造函数。

注意:如果类有间接的虚基类,除非只要使用虚基类的默认构造函数,否则应该显式调用该虚基类默认构造函数。即有间接的虚基类的类的构造函数应该用以下方式编写:

类名(形参列表): 基类(...),...,虚基类(...),...,成员(...),...

4. 类方法的冲突与解决

(1) 对于**单继承**,没有重新定义的函数使用最近祖先的定义

(2) 若多继承中最近多个祖先都有同名函数,在调用时会出现二义性。

(3) 指定要调用的同名方法;

对象. 类名::方法(实参列表);

在类的内部,可以使用作用域解析运算符调用基类的方法。

5. 混合使用虚基类和非虚基类

当类通过虚途径和非虚途径继承某个特定的基类时,该类会包含一个表示所有的虚途径的基类子对象和分别表示各条非虚途径的多个基类子对象。

6. 使用非虚基时,若类从不同的类继承了多个同名成员,则在使用时不使用类名限定将导致二义性。

7. **虚二义性规则**:在使用虚基类时,若某个名称优先于其他所有名称,则使用时,即使不使用限定名,也不会导致二义性。(派生类中的名称优先于直接或间接祖先类中的相同名称,虚二义性规则与访问规则无关)

四、类模板(常用于容器类)

1. 定义:在类声明前的开头:

template <typename Type>//typename 可换为 class

其中的 Type 为**类型说明符**,实例化时可用具体类型替代

在编写成员函数时,也要以此为开头,并将限定名改为 类型名<Type>::

2. 模板说明了如何生成类和成员函数定义,模板的具体实现叫做**实例化**或**具体化**。不可将模板成员函数放在独立的实现文件中。(不是函数,不能单独编译,需与实例化请求搭配使用)

3. 使用:类名<**类型参数列表**> 对象名(构造函数实参列表);

必须提供类型参数列表

4. 模板的**非类型(表达式参数)**:如 template<class T,int n>

其中的 int n 为表达式参数。表达式参数可以是整型、枚举、引用或指针。

模板代码不可修改参数值,也无法取地址(这是右值)

5. 默认类型模板参数

类模板可以为类型参数设置默认值,即:(defaultType1 为默认值)

template<class T1,class T2,class T3=defaultType1>

6. 模板的具体化

(1) **具体化**:包括**显式实例化**、**隐式实例化**、**显式具体化**

(2) **隐式实例化**:创建模板类对象

(3) **显式实例化**:让编译器显式生成实例(即使不用),格式:

template class 类名<类型参数列表>

(4) **显式具体化**:为特定类型专门定义,如果该类型需要特殊处理要使用。

格式: template<> class 类型名<专门类型> {...};

(5) **部分具体化**:给部分类型参数指定类型

格式: template<未具体化的类型> class 类型名<已经具体化的类型>{...};

7. 成员模板:

成员函数可以是模板函数, 当类为模板时, 编写函数实现就要出现两层 template, 不能合在一起。

8. 将模板用作参数: 举例:

```
template<template<typename T> class Thing> class Crab
```

其中模板参数是 `template<typename T> class Thing`, `template<typename T> class` 是类型, `Thing` 为参数。Thing 用于匹配一个模板类, 模板参数的形式应该与指定的一致。

在类 `Crab<Type>` 中, 若使用了 `Thing<int>`, 则会被替换为 `Type<int>`。

9. 模板的静态成员: 每个实例化各自有一个静态成员

10. 模板别名 (C++11)

使用格式 `using 别名=模板; // 也可以用于非模板`

使用实例:

```
template<typename T>
```

```
using arrtype=std::array<T,12>; //arrtype<t>等价于 std::array<t,12>
```

五、友元: 允许访问某一类私有成员的一个实体

分类:

- 友元函数 (允许访问类私有成员的外部函数, 通常用于运算符函数)
- 友元类
- 友元成员 (允许访问私有成员的外部类的成员函数)

1. 模板类和友元: 分类

- 非模板友元
- 约束模板友元: 友元的类型取决于类被实例化的类型
- 非约束模板友元

(1) 模板类的非模板友元

在模板类中将一个常规函数声明为友元, 则它将成为类模板所有实例化的友元
在友元中若提供了模板类参数, 如:

```
template<class T>
class HasFriend
{
    friend void report(HasFriend<T> &);
    ...
};
```

此时在编写 report 的实现时, 必须为特定类型编写。

(2) 模板类的约束模板友元函数: 定义步骤:

- ① 在类声明前声明模板函数
- ② 在函数中将模板声明为友元: 需要在函数名后用 <> 指明模板具体化
注意: 若友元函数没有参数, 需要在 <> 指出其类型, 否则编译器无法推断。
对于模板类特定的实例化, 会唯一对应一个约束模板友元函数的实例。
- ③ 定义友元函数。

(3) 非约束模板友元:

在类内声明模板, 每个函数具体化都是每个类的友元。

2. 友元类的声明: 在类的声明中使用 friend class 友元类;

友元声明可以在公有、私有或保护部分, 位置任意。

3. 友元成员函数: 让另一个类的特定成员函数成为友元, 须在类中添加如下声明:

friend 函数原型; // 注意函数名要使用限定名。

4. 避免类的循环依赖: 使用 前向声明 class 类名;

让整个类成为友元不需要前向声明。

5. 其他友元关系: 可以让两个类互为友元

6. 一个函数可以是多个类共同的友元

六、**嵌套类**: 在另一个类中定义的类

- 在被包含的类外的访问控制: 与普通成员一致。
- 嵌套类在对自生的访问控制是独立的。包含它的类不能直接访问其私有部分, 除非是友元。
- 公有嵌套类的访问: `所在类::嵌套类`

这种方式可以推广到多层公有嵌套类。

注: 在 C++ 中, 结构被进行了增强。结构可以看作一个类, 可以拥有成员函数。其与 `class` 的唯一区别在于默认访问权限是 `public`。

七、**异常**

- 定义: 程序遇到运行时错误而无法继续正常运行。
- 异常处理机制: 在出现异常时, 将控制权从程序的一个部分转移到另一个部分。过程如下:

①抛出异常 ②捕获异常 ③处理异常

1. 抛出异常: `throw 对象;` // 可以是字符串、数字、各种对象来表示异常

`throw;` // 可以重新抛出原有的异常

2. 异常的捕获和处理:

```
try
{
    可能会抛出异常的代码
}
catch(异常对象)
{
    处理异常
}
```

示例:

```
double div(double x, double y) { if(y==0) throw "Divided by zero"; return x/y; }
try { cout << div(a,b); } catch(const char* s) { cerr << "Exception: " << s << endl; }
```

可以出现多个 `catch` 块来捕获不同异常。

3. `abort()` 函数: 位于 `cstdlib` 中, 调用后向标准错误流发送消息, 然后终止程序, 还会返回一个异常值(取决于实现)。也可以使用 `exit()`, 该函数刷新文件缓冲区, 不显示消息

4. `std::exception` 类(C++ 标准类, 为其他异常的基类)

- 头文件: `exception`
- 返回字符串(`const char*`)的虚函数 `what()` 返回出错原因, 可在派生类中重定义。
- 使用后可以不需要多个 `catch` 类, 可以使用引用来捕获异常并显示消息。

5. `std::bad_alloc` 异常(从 `std::exception` 派生)

申请动态变量失败时, 抛出异常 `std::bad_alloc` 异常, 早期 C++ 返回 `NULL`

若不想要 `new` 抛出异常, 应该使用 `new (std::nothrow)` 失败时会返回空指针

6. `throw` 语句的执行: 类似于返回语句, 会终止函数执行, 但其会沿着函数调用序列后退, 直到找到包含 `try` 的函数, 然后便会去匹配相应的 `catch` 块。

若执行好 `try` 块后没有异常, 则会跳过 `catch` 块。

7. 关键字 `noexcept`: 指出函数不会抛出异常, 如:

```
double div(double, double) noexcept;
```

8. **栈解退**

若 `try` 块调用了对引发异常的函数进行调用的函数, 则会一直释放栈直到找到一个位于 `try` 块中的返回地址, 然后转到块尾的异常处理程序, 该过程叫做栈解退。

注意: C++ 函数的调用过程涉及 CPU 工作机制和汇编语言实现, 详情参见汇编语言资料

9. 引发异常时将创建一个临时拷贝，即使 catch 块中指定的是引用(这样做防止函数执行完后删除对象导致引用无效)

10. 在 catch 块中最好指定引用，因为基类引用可以引用派生类对象。

11. 引发的异常对象将被第一个与之匹配的 catch 块捕获。则在有一个异常类结构层次时，catch 块的排列顺序要与继承顺序相反，基类放最后。

12. catch(...) {} 用于捕获任意异常。

13. 在 catch 语句中使用基类对象时，将捕获所有的派生类对象，但会剥去派生特性。

*****14. 异常规范 (C++98 引入，C++11 废弃)

格式: 函数原型 throw(可能出现的异常的列表); // 列表为空表示为不抛异常

15. exception 类的派生类

(1) stdexcept 异常

- 头文件: stdexcept

- 定义的类型: logic_error(逻辑错误)、runtime_error(运行时错误)

两个类型都由 exception 派生而来

- 方法: 两个类的构造函数接受一个 string 对象作为参数, 用于提供方法 what() 返回的字符数据。

- logic_error 系列: 派生自 logic_error, 每个类都有类似于 logci_error 的构造函数

① domain_error(定义域错误)

当使用数学库函数时若给函数传入了超出定义域的值, 则引起定义域错误(如 asin() 传入 114)

② invalid_argument(非法值错误)

给函数传入意料之外的值时可以抛出

③ length_error(长度不足错误)

没有足够的空间完成指定的操作。

④ out_of_bounds(下标/索引错误)

指示容器类中访问了错误的索引(如越界)

- runtime_error 系列: 派生自 runtime_error, 每个类都有类似于 runtime_error 的构造函数

① range_error(范围错误)

② overflow_error(上溢错误)

③ underflow_error(下溢错误)

三者中①用于计算结果不在范围内, ②③用于浮点运算的上溢下溢。

16. 异常的使用注意

- 若异常是在带异常规范的函数中引发的, 则必须与规范列表中的某种异常匹配(类类型与该类及其派生类匹配), 否则为 **意外异常**。

- 如果异常不在函数中引发(或函数没有异常规范), 则必须捕获。若没有捕获, 为 **未捕获异常**。

- 未捕获异常发生后程序调用 terminate() 函数, 默认情况下 terminate() 调用 abort()

可以指定 terminate() 应该调用的函数, 使用 set_terminate() 函数。

set_terminate() 接受一个指向没有参数和返回值的指针, 并返回其地址。若调用了多次, 使用最后一次指定的函数。(函数声明自头文件 exception)

- 异常规范应包含函数调用的其他函数引发的异常, 若发生意外异常, 程序调用 unexpected() 函数, 默认情况下调用 terminate()。set_unexpected() 函数用于修改 unexpected() 调用的函数。

set_unexpected() 的原型与 set_terminate() 类似。(函数声明自头文件 exception)

提供给 set_unexpected() 的函数的限制:

① 通过调用 terminate()、abort() 或 exit() 终止程序

② 引发异常

③ 新异常若和原来的异常规范匹配, 则按正常流程处理(寻找 catch 块)

④ 若不匹配且异常规范没有 std::bad_exception(由 exception 派生) 则调用 terminate()

⑤ 若不匹配且异常规范有 std::bad_exception, 则异常被 std::bad_exception 取代。

八、RTTI(Runtime Type Identification, 运行阶段类型识别):

- 头文件 `typeinfo`

• 适用于包含虚基类的函数

1. typeid 运算符: 获取对象的类型

- 使用 `typeid(类名或结果为对象的表达式)`
- 返回 `type_info&`

2. type_info 类

- `operator==()` 比较类型
- `operator!=()` 比较类型
- `name()` 返回对应的类型名(若是基类指针指向派生类,解引用后显示的是派生类名)

3. dynamic_cast 运算符

- 用法: 安全转换指针或引用
- 语法: `dynamic_cast<要转换的类型的指针或引用>(类对象指针或引用)`

若能安全转换,返回对象的地址(引用),否则返回空指针(转换引用时引发异常 `bad_cast`,由 `exception` 派生)

• 通常如果指向的对象(*pt)类型为 `Type` 或者从 `Type` 派生而来的类型,则以下的表达式将指针 `pt` 转换为 `Type*` 类型。

注意:如果在大量的 `if else` 中使用了 `typeid`,则要考虑改为虚函数和 `dynamic_cast`

九、类型转换运算符

1. const_cast: 改变值为 `const` 或 `volatile`,除了 `const` 或 `volatile` 特征不同外,其他特征要相同,否则转换出错

• 可以用于删除对象的 `const` 属性,但修改 `const` 的结果不确定。当调用函数的形参不是 `const`,实参是 `const` 时,若函数不修改参数,就需要去除 `const` 让函数接受参数。

- 用法: `变量=const_cast<去掉 const 属性后的类型名>(const 对象);`
- 示例: `const int* pt=arr; int *pc=const_cast<int*>(pt);`

2. static_cast

- 编译器支持的隐式转换,比较安全,若不能隐式转换则出错
- 用法: `static<转换后类型> 对象`

3. reinterpret_cast

- 危险的转换,不允许删除 `const`
- 不允许的转换: 不能将指针转为更小的整数浮点型,不能将函数指针转为数据指针。
- 用法: `reinterpret_cast<转换后类型> 对象`

4. dynamic_cast: 在类层次中向上转换

第五部分 标准库的使用

SECTION 1 string 类

- 头文件: `string`
- 名称空间: `std`
- 特性:

1. string 是模板具体化 `basic_string<char>` 的一个 typedef.

2. size_type 是一个依赖实现的整型,位于头文件 `string`

3. string::npos 为字符串的最大长度,通常为 `unsigned int` 的最大值.

- 基础方法

方法	说明
<code>size()</code> 、 <code>length()</code>	返回字符串的字符个数
<code>operator[]()</code>	下标访问法
<code>operator+()</code>	拼接字符串

operator=()	赋值
-------------	----

- 构造函数

构造函数	说明
string(const char *s);	将 string 对象初始化为 C 风格字符串。 用法: string(要初始化为的 C 风格字符串);
string(size_type n,char c);	创建一个包含 n 个 c 字符的字符串。 用法 string(需要的字符数, 填充的字符);
string(const string &str);	复制构造函数
string();	默认构造函数
template<class Iter> string(Iter begin,Iter end);	初始化为区间[begin,end) 之间的字符,其包含 begin 不包含 end.begin 和 end 可以看做是指针(迭代器)。 用法: string(开始位置, 结束位置);
string(const string &str, size_type pos ,size_type n=npos);	初始化为对象 str 中从位置 pos 开始到结尾的字符, 或从位置 pos 开始的 n 个字符.用法: string(被操作 string 对象, 开始位置, 抽取字符数=到达结尾);
string(string &&str) noexcept;	[C++11]初始化为 str,并可能修改(移动构造函数)
string(const char*s, size_type n);	初始化为 s 的前 n 个字符,即使超过'\0' 用法: string(要操作的 C 风格字符串,需要的长度)
string(initializer_list<char> il);	[C++11]初始化为花括号初始化列表里的字符。 Initializer_list 是初始化列表,如 {1,2,3,4,5};

- string 版 getline() 用法:getline(输入流对象, string 对象);

该函数自动调整 string 大小

当读取的字符数达到最大值,设置 failbit

- string 的逻辑运算符均被重载。

- 字符搜索:find() 方法:

- (1) size_type find(const string &str,size_type pos=0) const;
- (2) size_type find(const char *s,size_type pos=0) const;
- (3) size_type find(const char ch,size_type pos=0) const;

从字符串的 pos 位置查找子字符串 str(或 s)或字符 ch 的位置。

若找到,返回其首地址,否则返回 string::npos。

(1) (2) (3) 用法:find(string 字符串或 C 风格字符串或字符, 开始位置=0);

(4)size_type find(const char *s,size_type pos,size_type n) ;从字符串的 pos 位置开始, 查找 s 的前 n 个字符组成的字符串, 返回方式同(1) (2)。

用法:find(待取的 C 风格字符串,查找的开始位置,取的字符数);

- 其他搜索:与 find() 的重载特征都相同

(1) rfind() 查找子字符串最后一次出现的位置。

(2) find_first_of() 查找参数中任何一个字符首次出现的位置

(3) find_last_of() 与 (2) 相仿,但查找最后一个

(4) find_first_not_of() 查找第一个不包含在参数中的字符

(5) find_last_not_of() 与 (4) 相仿,但查找最后一个

- 其他方法:

capacity() 返回当前分配给字符串的内存块的大小

reserve(长度) 请求内存块的最小长度

c_str() 返回对应的 C 风格字符串

****注:**

模板 basic_string 有 4 个具体化,每个具体化都有一个 typedef 名称:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
typedef basic_string<char16_t> u16string; //C++11
typedef basic_string<char32_t> u32string; //C++11
其原型如下:
template<class charT, class traits=char_traits<charT>,
class Allocator=allocator<charT> > basic_string;
charT 是字符类型, traits 是其特征, Allocator 管理内存分配(使用 new 和 delete)
```

SECTION 2 智能指针

• 头文件: **memory**

• 模板: **auto_ptr**(C++98 提出 C++11 废弃)、**shared_ptr**、**weak_ptr**

1. 共同特点: 定义了类似指针的对象, 可以将从 new 获得的地址赋给这种对象。过期后会自动使用 delete 删除。因此不能用于非堆空间。若使用了 new[] 分配内存, 只能用 unique_ptr

2. 创建和使用

auto_ptr 模板的定义如下

```
template<class X> class auto_ptr{public: explicit auto_ptr(X *p=0) throw(); ...};
```

其他两个模板类似。

其不会自动将指针转换为智能指针对象。声明 X 类型的智能指针方法:

智能指针模板名<X> 指针对象名(指针);

智能指针支持常规的指针操作, 能赋给相同类型的常规指针, 还能赋给相同类型的智能指针。

3. 注意事项:

(1) 所有权: 对于 auto_ptr 和 unique_ptr, 赋值会转让所有权, auto_ptr 和 unique_ptr 只有一个智能指针可以拥有特定的对象, 该指针才能删除对象。

(2) shared_ptr 会跟踪引用智能指针, 进行引用计数。仅在最后一个指针过期时才使用 delete。

(3) auto_ptr 转让所有权后不能使用它访问对象, auto 能够直接使用=, 但会带来悬空指针

(4) unique_ptr 在转让所有权时, 原指针不再指向有效数据, 不能使用=进行赋值。

若原对象是一个临时右值(如函数返回时会创建临时对象, 此时会立马剥夺临时对象的所有权并且立马销毁临时对象), 允许进行赋值, 如果将存在一段时间, 则不能(以防止指针悬空)。

(5) std::move() 将一个 unique_ptr 转换后返回, 此时便可赋给另一个。用法 move(源指针)。

(6) 模板 auto_ptr 使用 new 和 delete; unique_ptr 有使用 new[] delete[] 的版本, 示例:

```
unique_ptr<double []> pda(new double(5));
```

SECTION 3 STANDARD TEMPLATE LIBRARY(STL, 标准模板库)

• 基础知识

1. STL 包括了一组表示 **容器**(类似数组)、**迭代器**(能够用来遍历容器的对象, 与能够遍历数组的指针相似, 是一个广义指针)、**函数对象**(类似于函数的对象)和**算法**(完成特定任务的过程)的模板。

2. **分配器**: 各种 STL 容器模板都接受一个可选的模板参数, 该参数指定了使用哪一个分配器对象来管理内存。(使用的是 new 和 delete)

3. 所有的 STL 容器提供的基本方法:

.size() 容器中的元素个数

.swap() 交换两个容器的内容

.begin() 返回一个指向容器第一个元素的迭代器

.end() 返回一个表示超过容器尾的指针

4. 每一个容器都定义了一个迭代器, 为一个名为 iterator 的 typedef, 作用域为类。声明一个迭代器方法:

类名::iterator 迭代器名;

其可以: 解除引用 operator*()、自增 operator++()

E.g. `vector<double>::iterator`

5. **超过结尾**(`past_the_end`): 指向容器最后一个元素后面的元素的迭代器

6. 某些 STL 容器特有的方法:

- (1) `push_back`(追加的元素) 将元素添加到末尾, 并增加长度
- (2) `erase`(开始迭代器, 结束迭代器) 删除制定区间的元素, 接受两个迭代器(包含 `operator+` () 方法), 包括开始不包括结束。
- (3) `insert`(新元素的插入位置迭代器, 被插入区间开始迭代器, 被插入区间结束迭代器); 用于插入元素, 接受 3 个迭代器, 包括开始不包括结束。

e.g. `new_v.insert(new_v.begin(), old_v.begin(), old_v.end());`

将矢量 `old_v` 的内容插入到矢量 `new_v` 的第一个元素前面

• **算法** (头文件 `algorithm`)

`for_each`(开始迭代器, 结束迭代器, 要应用的函数指针); 前两个是一个容器的区间(不包括结束), 最后是函数指针, 会被应用于区间的各个元素, 被指向的函数不能修改元素值。

`random_shuffle`(开始迭代器, 结束迭代器); 随机排列区间的元素, 要求容器随机访问。

`sort`(开始迭代器, 结束迭代器)

(开始迭代器, 结束迭代器, 自定义排序函数)

只接受区间时, 使用 `<` 运算符, 对区间升序排序。(要有对应的 `operator<()` 函数)

接受区间和函数指针时, 函数返回值为 `false` 表示排序不正确。

• **迭代器**:

1. C++ 将 `operator++()` 作为 **前缀版本**, 将 `operator++(int)` 作为 **后缀版本**。其中的参数根本不会使用, 因此无需指定名称

2. 迭代器都能进行解除引用和比较

3. 迭代器的类型:

(1) **输入迭代器**: 来自容器的信息被称为 **输入**, 可被程序用来 **读取容器中的信息**, 但不一定让程序 **修改**, 该种算法 **不修改值**。(只读不写)

基于输入迭代器的任何算法都是 **单通行**的, 可以 **递增**, 不可倒退。

(2) **输出迭代器**: 将信息从程序传输给容器的迭代器。 **只能解除引用修改容器值**, 不能读取。(只写不读) 输出迭代器是单通行的。

(3) **正向迭代器**: 与输入迭代器和输出迭代器相似, 只用 `++` 运算符遍历容器。每次沿容器向前移动一个元素。总是按照相同的顺序遍历一系列的值。仍可以对以前的值解除引用, 能够读写数据, 也可只读数据。

(4) **双向迭代器**: 包含了正向迭代器的特性, 支持 `--` 运算符。

(5) **随机访问迭代器**: 能够直接跳转到任意元素(随机访问), 包含了双向迭代器的特性, 支持随机访问。

随机访问迭代器操作: `a`、`b` 为迭代器值, `n` 为整数, `r` 为随机访问迭代器的变量或引用。这些表达式只有在容器区间内(包括超尾)才合法。其操作如下:

`a+n` 或 `n+a`: 指向 `a` 所指向的元素后的第 `n` 个元素;

`a-n`: 指向 `a` 所指向的元素前的第 `n` 个元素;

`r+=n`、`r-=n`: 复合运算符;

`a[n]` 同 `*(a+n)`; `b-a` 减法; `a<b`, `a>b`, `a<=b`, `a>=b` 用来比较地址。

(6) 迭代器的性能

Table 16.4 Iterator Capabilities

Iterator Capability	Input	Output	Forward	Bidirectional	Random Access
Fixed and repeatable order	No	No	Yes	Yes	Yes
<code>++i</code> <code>i++</code>	Yes	Yes	Yes	Yes	Yes
<code>--i</code> <code>i--</code>	No	No	No	Yes	Yes
<code>i[n]</code>	No	No	No	No	Yes
<code>i + n</code>	No	No	No	No	Yes
<code>i - n</code>	No	No	No	No	Yes
<code>i += n</code>	No	No	No	No	Yes
<code>i -= n</code>	No	No	No	No	Yes

4. 概念、改进与模型:

概念是一系列的要求;改进是概念上的继承(因为其无法使用 C++ 的类继承来描述,如将指针用作迭代器);模型是概念的具体实现。

以下算法使用头文件 `algorithm`

(1) STL `std::sort()` 函数:接受指向容器(数组)的第一个元素的迭代器(指针)和指向超尾的迭代器(指针),用于排序。

用法: `sort(首迭代器, 超尾迭代器);`

(2) STL `std::copy()` 函数:从一容器复制元素到另一迭代器,前两个迭代器指出要复制的元素范围(输入迭代器),最后一个迭代器(输出迭代器)指出要将第一个元素复制到的位置(目的地)。注意:目标容器要足够大。复制不同于插入,其会覆盖原有的数据。

用法 `copy(开始, 结束, 目的地);`

头文件: `iterator`

(3) 将信息复制到输入输出和其他迭代器

适配器:用于将其他接口转为 STL 接口

①表示输出流的迭代器: `ostream_iterator` 模板,其是一个适配器,可将其他的接口转换为 STL 接口用法:

`ostream_iterator<被发送给输出流的数据类型, 输出流所使用的字符类型>`

迭代器名(输出流名称, 每个数据的分隔符(字符串));

如: `ostream_iterator<int,char> cout_iter(cout, " ");`

使用 `*cout_iterator++ = 15` 即可复制,等价于 `cout<<15<<" "`;

可复制到输出流内使用数据,可以创建无名迭代器。

如: `copy(dict.begin(),dict.end(),ostream_iterator<int,char>(cout, ""));`

`//ostream_iterator<int,char>(cout, ""))` 也可以换为一个具体的名称。

② `istream_iterator` 使 `istream` 输入可用作迭代器接口,拥有两个模板参数,声明方法:

`istream_iterator<读取的数据类型, 输入流使用的字符类型>迭代器名(输入流名)`,使用 `cin` 代表 `cin` 管理输入流,省略构造函数表示输入失败。可将其运用到 `copy()` 算法中,直到 EOF、不匹配或其他问题为止。样例:

`copy(istream_iterator<int,char>(cin),istream_iterator<int,char>(),dict.begin());`

③ `reverse_iterator` (反向迭代器):对反向迭代器递增操作将导致其被递减。

`vector` 类有 `rbegin()` 和 `rend()` 的成员函数, `rbegin()` 返回指向超尾元素的反向迭代器 (`vector<T>::reverse_iterator`), `rend()` 返回第一个元素的反向迭代器。

若要反着打印容器内容,可以进行如下操作:

`copy(dice.rbegin(),dice.rend(),ostream_iterator<int,char>(cout, ""));`

反向指针通过先递减再解除引用解决问题。(即 `*rp` 将在 `*rp` 的当前值之前对迭代器解除引用。)

④插入迭代器:添加新的元素,不会覆盖已有的数据,并会自动分配内存来容纳新的内容

`back_insert_iterator` 将容器插入到容器尾部

`front_insert_iterator`:插入到容器的前端

`insert_iterator`:将元素插入到其构造函数指定的位置前面

可以直接使用 `copy()` 函数。

限制如下: `back_insert_iterator` 需要允许在尾部快速插入的容器(插入操作的时间复杂度为常数,如 `vector`), `front_insert_iterator` 需要允许在起始位置做时间固定插入的容器类型(如 `queue`), `insert_iterator` 没有限制。

这些迭代器将容器类型作为模板参数。`back_insert_iterator` 的构造函数假设传递给它的类型有 `push_back()` 方法。声明 `front_insert_iterator` 大同小异。`insert_iterator` 需要指出位置。他们的声明格式为:

`back_insert_iterator <容器类型> 迭代器名(容器名);`

`front_insert_iterator` <容器类型> 迭代器名(容器名);

`insert_iterator`<容器类型> 迭代器名(容器名,插入位置);

• **容器**: 早期的 11 个容器类型: `deque`、`list`、`queue`、`stack`、`vector`、`map`、`multimap`、`set`、`multiset` 和 `bitset`。C++11 新增容器: `forward_list`、`unordered_map`、`unordered_multimap`、`unordered_set` 和 `unordered_multiset`。

1. 容器的基本特征:

X 表示容器; T 表示存储在容器中的对象类型; a、b 表示类型为 X 的值; r 表示类型为 X& 的值; u 表示类型为 X 的标识符。

表达式	返回类型	说明	(时间)复杂度
<code>X::iterator</code>	指向 T 的迭代器	正向迭代器	编译时间
<code>X::value_type</code>	T	T 的类型	编译时间
<code>X u;</code>		创建容器 u(空)	固定 O(1)
<code>X ();</code>		创建匿名空容器	固定 O(1)
<code>X u(a);</code> <code>X u=a;</code>		调用复制构造函数后 <code>u==a</code> .	线性 O(n)
<code>r=a</code>	X&	赋值运算符	线性 O(n)
<code>a.begin()</code>	迭代器	指向第一个元素的迭代器	固定 O(1)
<code>(&a)->~X()</code>	void	析构函数	线性 O(n)
<code>a.end()</code>	迭代器	返回指向超尾元素的迭代器	固定 O(1)
<code>a.size()</code>	无符号整型	返回元素个数	固定 O(1)
<code>a.swap(b)</code>	void	交换 a、b 的内容	固定 O(1)
<code>a==b</code> <code>a!=b</code> 与 <code>a==b</code> 相反	可转为 bool	如果 a、b 长度相等, 相应的元素都相等, 返回真。	线性 O(n)

复杂度描绘了执行操作所需要的时间。从快到慢依次为: 编译时间、固定时间、线性时间。

2. C++11 新增的容器要求: rv 表示类型为 X 的非常量右值

<code>X u(rv);</code> <code>X u=rv;</code>		调用移动构造函数	线性
<code>a=rv;</code>	X&	调用移动赋值运算符	线性
<code>a.cbegin()</code>	<code>const_iterator</code>	返回指向第一个元素的 const 迭代器	固定
<code>a.cend()</code>	<code>const_iterator</code>	返回指向超尾元素的 const 迭代器	固定

移动操作可能修改源对象, 还可能转让所有权, 而不做任何复制。

3. 序列容器: (7 种 STL 容器: `deque`、`forward_list`、`queue`、`priority_queue`、`stack` 和 `vector`)

(1) `queue` 队列能够在队尾添加元素, 在队首删除元素; `deque` 表示双端队列允许在两端添加或删除元素 (array 也被归为序列容器)。

(2) 特征: 按线性顺序排列, 即存在第一个元素, 最后一个元素, 除了该两个元素处其他的元素前后都分别有一个元素。

(3) 序列的要求: t 表示类型为 T 的值, n 表示整数, p、q、i、j 表示迭代器 (T 可做存储在容器中值的类型)

①基本操作[注: 区间 [x,y) 的范围包括 x(起始) 不包括 y(结束)]

表达式	返回类型	说明
<code>X a(n,t);</code> <code>X(n,t);</code>		声明一个由 n 个 t 值组成的序列, 名称为 a 或匿名。 格式: X(需要的数量, 需要的值)
<code>X a(i,j);</code> <code>X (i,j);</code>		声明一个名为 a(或匿名)的容器并初始化为区间 [i,j) 的内容。 格式: X(开始, 结束/[区间])
<code>a.insert(p,t);</code>	迭代器	将 t 插入到 p 的前面

		格式: insert(位置, 待插入值)
a.insert(p,n,t);	void	将 n 个 t 插入到 p 的前面 格式: insert(位置, 需要的数量, 待插入值)
a.insert(p,i,j);	void	将区间[i,j)中的元素插到 p 前 格式: insert(要插入的位置, [区间]/开始位置, 结束位置)
a.erase(p);	迭代器	删除 p 所指向的位置 格式: erase(删除位置)
a.erase(p,q);	迭代器	删除区间[p,q)中的元素 格式: erase([待删除的区间]/开始位置, 结束位置)
a.clean()	void	清空容器

②可选要求: 在允许的情况下, 时间复杂度为固定时间。

表达式	返回类型	含义	容器
a.front	T&	*a.begin()	vector,list,deque
a.back()	T&	*--a.end()	vector,list,deque
a.push_front(t)	void	a.insert(a.begin(),t)	list,deque
a.push_back(t)	void	a.insert(a.end(),t)	vector,list,deque
a.pop_front(t)	void	a.erase(a.begin())	list,deque
a.pop_back(t)	void	a.erase(--a.end())	vector,list,deque
a[n] a.at(n)	T&	*(a.begin()+n)	vector,deque

注意: at() 方法会检查边界, 若越界则引发 out_of_range 异常。

(4) 序列容器的详解

• 矢量类 std::vector

模板类 vector 即 std::vector 位于 vector 头文件中。

可以动态修改长度、可以随机访问元素、在尾部添加删除元素的时间固定、在头部中间删除元素复杂度为线性时间、它是可反转容器, 拥有 rbegin() 和 rend()、使用 reverse_iterator 可以反向遍历容器。

声明格式: vector<元素类型> 对象名(元素个数);

其类似于数组, 可用 operator[]() 访问元素。(提供随机访问功能)

• deque 双向队列 [Double-ended Queue]

模板位于头文件 deque

类似于 vector 容器, 支持随机访问, 从 deque 对象的开始位置插入和删除元素的时间是固定的。

• list 双向链表

模板位于 list 头文件中。

除了第一个和最后一个元素之外, 每个元素都与前后的元素相链接, 可以双向遍历。list 链表中任一位置进行插入和删除的时间是固定的。list 可以反转, 但不支持数组表示法和随机访问。从容器插入或删除元素后, 链表迭代器指向元素不变。

list 成员函数 [Alloc 模板参数有默认值]

函数	说明
void merge(list<T,Alloc>&x) merge(要合并的链表)	将链表 x 与调用链表合并。两个链表必须已近排序。合并后的经过排序的链表保存在调用链表中, x 为空。时间复杂度为线性时间
void remove(const T& val) remove(待删除实例列表)	从链表中删除 val 的所有实例, 复杂度为线性时间。
void sort()	使用<排序, n 个元素时间复杂度为 $n \log n$

void splice(iterator pos , list<T,Alloc> X) splice(插入位置, 待插入链表)	将链表 X 的内容插入到 pos 的前面, X 将清空。复杂度为固定时间。
void unique()	将连续相同的元素压缩为单个元素, 复杂度为线性时间。

insert() 和 splice() 的区别:

insert() 将原始区间的副本插入到目标地址; splice() 将原始区间移到目标地址, 在其执行后迭代器仍然有效。

非成员函数 sort() 需要随机访问迭代器。

• [C++11] forward_list 单链表

在单链表中, 每个节点只链接到下一个节点, 并没有链接到前一个节点。则它只要正向迭代器, 它是不可反转容器, 比 list 更简单。

• queue 队列 [适配器类]

queue 模板位于头文集 queue, 不允许随机访问队列元素, 不允许遍历队列, 可以将元素添加到队尾, 从队首删除元素, 查看首尾的值, 检查元素数目和测试是否为空。

queue 的操作	
方法	说明
bool empty() const	队列为空返回 true 否则为 false
size_type size() const	返回元素数目
T& front()	指向队首元素的引用
T& back()	指向队尾元素的引用
void push(const T&x)	在队尾插入 x
void pop()	删除队首元素

• priority_queue 优先队列 模板:

位于头文集 queue 中, 支持操作与 queue 相同。主要区别在于其最大的元素被移到队首, 内部默认底层类为 vector, 可以修改确定元素的比较方法, 构造函数如下:

```
priority_queue<int> pq1; // 默认
```

```
priority_queue<int> pq2(greater<int>); // 可选
```

• stack 栈:

位于 stack 头文件中, 其特点是先入后出。

不允许随机访问遍历; 允许压栈出栈, 查看栈顶值, 检查元素数目和栈是否为空。

stack 操作	
方法	说明
bool empty() const	栈为空返回 true 否则为 false
size_type size() const	返回元素数目
T& top()	返回指向栈顶的引用
void push(const T&x)	在栈顶压入 x
void pop()	删除栈顶元素(出栈)

• array [C++11]

它是非 STL 容器, 其长度固定, 拥有成员 operator[] 和 at(), 可将 STL 算法用于它。

4. 关联容器: 对容器概念上的另一改进。

关联容器将 **值** 与 **键** 关联起来, 并使用键来查找值。

对于容器 X, X::value_type 指出了存储在容器中的值的类型, X::key_type 指出了键的类型。提供了对元素的快速访问。允许插入新元素, 但不能指定插入的位置。使用树实现, 根节点链接 1 到 2 个节点, 每个节点都如此, 从而形成分支结构。STL 提供 4 个关联容器: set、multiset(位于 set 头文件)、map、multimap(位于 map 头文件); set 的值与键类型相同, 键是唯一的, set 的值

就是键。multiset 与 set 相似，只是可能有多个值相同。

在 map 中，值与键的类型不同，键是唯一的，每个键对应一个值，multimap 类似，只是一个键可以对应多个值。

(1) set 集合

①关联集合，可反转，可排序，且键是唯一的，不可以存储多个相同的值；

②声明：`set<键值类型> 名称;`

第二个模板参数是可选的，可以用来指示对键进行排序的比较函数或对象。默认用 `less<键值类型>`。

③通用函数[要先排序]

`set_union()` 求并集(合并相同的值)，前两个参数指定第一个集合的区间，接下来两个指定第二个集合的区间，第五个指定了插入的地方，即输出迭代器。

用法：`set_union(第一个集合的开始, 第一个集合的结束/[第一个集合的区间], 第二个集合的开始, 第二个集合的结束/[第二个集合的区间], 输出的位置);`

`set_intersection()` 求交集、`set_difference()` 求两集合之差的用法与上面相同

④set 的方法

`lower_bound(键值)` 将键作为参数并返回一个迭代器，指向集合中第一个不小于键参数的成员

`upper_bound(键值)` 将键作为参数并返回一个迭代器，指向集合中第一个大于键参数的成员

(2) multimap

①创建：`multimap<键类型, 数据类型> 名称;`

第三个模板参数可选，用于指出对键进行排序的比较函数或对象，默认为 `less<>`；为了将信息结合在一起，实际的值与键将进行结合，`pair<class T, class U>` 模板将这两种值存储到一个对象当中。如果 `keytype` 是键类型，`datatype` 是数据类型，则值类型为 `pair<const keytype, datatype>` (注意 multimap 直接存值，即 pair)

数据项是按键排序的。

②使用：模板 pair 的构造函数：`pair<键类型, 数据类型>(键, 值);`

使用 `insert(pair 值)` 方法插入容器。对于 pair 对象可以用 `first` 和 `second` 来访问键与值。

③获取 multimap 信息

成员函数 `count(键)` 接受键作为参数，返回具有该键的数目；成员函数 `lower_bound()` 和 `upper_bound()` 将键作为参数，原理与 set 的相同；成员函数 `equal_range()` 用键作为参数，返回两个迭代器

他们表示的区间与该键匹配。该方法将两个值封装在一个 pair 对象中，这个 pair 的两个模板参数均为迭代器。

5. **无序关联容器**：对容器概念的另一种改进。底层差别在于其基于数据结构哈希表，提供了效率。共 4 种：`unordered_set`、`unordered_multiset`、`unordered_map`、`unordered_multimap`。

• 函数对象

函数对象，也叫**函数符**。主要以函数方式与 `operator()` 结合使用的任意对象。包括了函数名、函数指针和重载了 `operator()` 的类对象。

1. 概念：

(1) **生成器**：不用参数的函数符

(2) **一元函数**：接受一个参数的函数符

(3) **二元函数**：接受两个参数的函数符

(4) 返回 bool 的一元函数叫**一元谓词**

(5) 返回 bool 的二元函数叫**二元谓词**

提供给 `for_each()` 的函数符应是一元函数；`sort()` 的其中一个版本将二元谓词作为第三个参数；`list` 模板有一个将谓词作为参数的 `remove_if()` 成员，该函数将谓词应用于区间中的每个元素，如果谓词为 true，则删除元素。即 `remove_if(表示删除条件的谓词);`

E. `g.scores.remove_if(too_big);` // `too_big` 是谓词

2. 预定义的函数符:

函数 `std::transform()` 两个版本:

(1) 接受 4 个参数。前两个参数指定容器区间的迭代器, 第三个指定结果复制到哪。最后一个参数是一元函数符, 应用于区间每个元素, 生成结果中的新元素。

即: **`transform(开始应用位置, 结束位置/[应用区间], 结果保存位置, 要应用的函数符);`**

(2) 使用一个二元函数符, 并将该函数应用于两个区间中的元素。它用第三个参数表示第二个区间的开始位置(没有结束位置, 因为第一个区间表明了长度)

即: `transform(开始应用位置, 结束位置/[应用区间], 第二个区间开始位置, 结果保存位置, 要应用的函数符);`

E.g. `m8, gr8` 是 `vector<double>` 对象, `mean(double, double)` 返回两数平均值, 则 `transform(gr8.begin(), gr8.end(), m8.begin(), ostream_iterator<double, char>(cout, " "), mean);`

输出 `m8` 与 `gr8` 的每个元素的平均值。

(3) 运算符与对应函数符(头文件 `functional`): 定义了多个模板类函数对象, 即: `函数符<类型>()`; 如: `plus<int>()`。

对于内置的运算符和重载的运算符均有函数符:

+	plus	-	minus	*	multiplies	/	divides	%	modulus
-	negate								
==	equal_to	!=	not_equal_to	>	greater				
<	less	>=	greater_equal	<=	less_equal				
&&	logical_and		logical_or	!	logical_not				

3. 自适应函数符与函数适配器

自适应函数符: 携带了标识参数类型与返回类型的 `typedef` 成员。这些成员的名称: `result_type`、`first_argument_type`、`second_argument_type`。

如: `plus<int>::result_type` 是 `int` 的 `typedef`。

意义: 函数适配器对象可以使用函数对象, 并认为存在这些 `typedef` 成员。

STL 使用 `binder1st` 和 `binder2nd` 类自动完成将自适应二元函数转换为一元函数(函数适配器)

(1) `binder1st` 若有一个自适应二元函数对象, 则可以创建一个 `binder1st` 对象, 该对象与一个将被用做该函数的第一个参数的值相关联。即: `binder1st(自适应二元函数, 关联的值)` 函数符名; 如: `binder1st(f2, val)` `f1`; 则 `f1(x)` 等价于 `f2(val, x)`

`bind1st` 函数可以简化过程, 其返回一个函数符。用法:

`bind1st(要转换的函数符, 要绑定的第一个参数)`。

(2) `binder2nd` 类和 `bind2nd`: 与 `binder1st` 和 `bind1st` 类似, 但是关联第二个参数。

[算法:]

7. 非成员函数算法[STL 函数可用于常规数组]概论:

可以用 `==` 比较不同容器, 重载的运算符使用迭代器来比较内容。

算法分为 4 组: 非修改式序列操作、修改式序列操作、排序和相关操作(头文件 `algorithm`)、通用数字运算(头文件 `numeric`)

8. 算法:

(1) 就地算法: 结果在原容器中;

(2) 复制算法: 将结果拷贝到另一位置;

(3) 有些算法有就地与复制两个版本。复制版本以 `_copy`, 将额外接受一个输出迭代器参数, 作为结果存放的位置。

(4) `replace()` 函数原型:

`template<class ForwardIterator, class T>`


```
void replace(ForwardIterator first, ForwardIterator last,
const &T old_vaule, const &T new_vaule);
```

所有的 old_vaule 将被替换为 new_vaule。

即: replace(开始位置, 结束位置/[应用区间], 待替换的旧值, 替换后的新值);

```
template<class InputIterator, class OutputIterator, class T>
```

```
OutputIterator replace(InputIterator first,
```

```
InputIterator last, OutputIterator result,
```

```
const &T old_vaule, const &T new_vaule);
```

复制版本在第三个参数指定了一个名为 result 的新位置, 将结果复制过去。对于复制算法, 返回一个迭代器, 指向复制的值的超尾。

即: 输出迭代器 replace(开始位置, 结束位置/[应用区间], 输出的位置, 待替换的旧值, 替换后的新值);

以 if 结尾的版本将函数应用于容器元素的结果来执行操作, 如果将函数应用于旧值中, 返回值为 true 则 replace_if() 把旧值替换为新值。原型如下

```
template<class ForwardIterator, class Predicate, class T>
```

```
void replace(ForwardIterator first, ForwardIterator last,
```

```
Predicate pred, const &T new_vaule);
```

即: replace(开始位置, 结束位置/[应用区间], 判断是否替换的谓词, 替换后的新值);

(5) STL 和 string 类: string 类包括了 begin()、end()、rbegin()、rend() 等成员, 可使用 STL 接口。next_permutation() 算法将区间内容转为下一种排列方式。对于字符串, 排列按照字母递增的顺序进行。如果成功, 返回 true; 如果该区间已经处于最后的序列中, 该算法返回 false。要得到所有的排列, 应先排序。

用法: next_permutation(开始排列位置, 结束位置/[排列的区间]);

(6) 如果 la 和 lb 均为 list<int> 对象, 则以下调用等价:

```
la.remove(4); remove(lb.begin(), lb.end(), 4);
```

list 链表使用 remove() 来删除某一实例, STL remove() 函数接受区间并删除实例, 但其不能调整容器长度。它将没有删除的元素放在链表开头, 并返回新的超尾值。

(7) 使用 STL: count() 函数, 将一个区间和一个值作为参数, 并返回这个值在出现的次数。即 count([区间], 值);

map 类可以用数组表示法将键作为索引访问存储值。

[其他库:]

22. vector、valarray、array

valarray 不是 STL 的一部分, array 提供了 STL 的方法

valarray 重载了所有的算术运算符, 如+表示每个元素相加, *表示每个元素相乘, 其他的大体相似。

valarray 重载了许多数学函数, 其接受一个 valarray 对象并返回结果。如 valarray<double>(6)={1,1,4,5,1,4}; log(va3);

也可以使用 apply() 方法, 传入函数进行运算, 其不修改调用对象, 返回一个包含结果的新对象。

valarray 有 resize() 方法, 不能自动调整大小。

C++11 提供了接受 valarray 对象作为参数的 begin() 和 end(), 满足 STL 的区间要求。

如果 numbers 是 valarray<double> 对象, 则下面语句中的 vbool[i] 将被设为 numbers[i]>9 的 bool 值。

```
valarray<bool> vbool=numbers>9;
```

下标指定 slice 类: 用作下标索引, 被初始化为 3 个整数值:

起始索引, 索引数, 步长(元素的距离)。如 slice(1,4,3) 指索引 1, 4, 7, 10。

头文件: initializer_list

9. initializer_list 类[C++11]

(1) 如果类包含接受 initializer_list 作为参数的构造函数，则 {} 表示法会调用它。

(2) 所有 initializer_list 元素的类型必须相同。

(3) 使用：该模板包含 begin()、end()、size()。

声明：initializer_list<类型> 名称={列表};

可以按值或引用传递。函数参数可以是 initializer_list 字面量，也可以是其变量。其迭代器类型为 const，不能修改值，但可以互相赋值。

SECTION 4 标准输入输出、文件 I/O

<暂缺>

第六部分 各个标准新特色探究

(一)C++11 特性

<暂缺>

(二)C++14/17 特性

<暂缺>

第七部分 拓展

(一)彩色控制台

控制命令

我们常用的 printf 函数输出来的颜色是终端的配色。如果想要输出不同的颜色进行区分，就需要用到 printf 的控制命令：\033[m。控制命令以\033[开头，以 m 结尾，而中间则是属性码，属性代码之间使用;分隔，如\033[1;34;42m。而属性代码的含义见下面的表格。

printf 属性代码

这里列举了三类属性代码，当然这只是一部分。

通用格式控制		前景色		背景色	
属性代码	功能	属性代码	颜色	属性代码	颜色
0	重置所有属性	30	黑色	40	黑色
1	高亮/加粗	31	红色	41	红色
2	暗淡	32	绿色	42	绿色
4	下划线	33	黄色	43	黄色
5	闪烁	34	蓝色	44	蓝色
7	反转	35	品红	45	品红
8	隐藏	36	青色	46	青色


示例：注意：若打印了乱码，请引入 stdlib.h, 执行 system(“”);

```
#include<stdio.h>
#include<stdlib.h>
void printf_red(const char *s){printf("\033[0m\033[1;31m%s\033[0m", s);}
void printf_green(const char *s)
{printf("\033[0m\033[1;32m%s\033[0m", s);}
void printf_yellow(const char *s){printf("\033[0m\033[1;33m%s\033[0m", s);}
void printf_blue(const char *s){printf("\033[0m\033[1;34m%s\033[0m", s);}
void printf_pink(const char *s){printf("\033[0m\033[1;35m%s\033[0m", s);}
void printf_cyan(const char *s){printf("\033[0m\033[1;36m%s\033[0m", s);}
int main()
{
    system("");
```

```
printf_red("Hello World.\n");  
printf_green("Hello World.\n");  
printf_yellow("Hello World.\n");  
printf_blue("Hello World.\n");  
printf_pink("Hello World.\n");  
printf_cyan("Hello World.\n");  
return 0;  
}
```

上面的代码中，每个函数都对应输出一种颜色的字符串，这里只用了高亮加前景色，没有设置背景色。我们看到printf中的字符串开头和结尾均是\033[0m，这个代码的作用就是重置所有设置过的属性，在开头添加是为了防止其他的设置对自身有影响，而在结尾添加则是为了防止自身对其他地方的设置有影响。然后中间的代码如\033[1;31m%s则是设置为高亮节加红色。

结果如右



(二) C++函数使用的栈机制

<暂缺>

第八部分 算法与数据结构部分