C/C++ 语言参考

目录

1. C/C++ 语言参考

基本 C/C++

- 预处理命令(见 [标题编号.])
- 操作符优先级(见 [标题编号.])
- 转义字符(见 [标题编号.])
- ASCII 码表(见 [标题编号.])
- 基本数据类型(见 [标题编号.])
- 关键字(见 [标题编号.])

标准 C 库:

  - Standard C I/O(见 [标题编号.])
  - Standard C String & Character(见 [标题编号.])
  - Standard C Math(见 [标题编号.])
  - Standard C Time & Date(见 [标题编号.])
  - Standard C Memory(见 [标题编号.])
  - Other standard C functions(见 [标题编号.])

C++

- C++ I/O(见 [标题编号.])
- C++ Strings(见 [标题编号.])

C++ 标准模板库(见 [标题编号.])

  - C++ Bitsets(见 [标题编号.])
  - C++ Double-Ended Queues(见 [标题编号.])
  - C++ Lists(见 [标题编号.])
  - C++ Maps(见 [标题编号.])
  - C++ Multimaps(见 [标题编号.])
  - C++ Multisets(见 [标题编号.])
  - C++ Priority Queues(见 [标题编号.])
  - C++ Queues(见 [标题编号.])
  - C++ Sets(见 [标题编号.])
  - C++ Stacks(见 [标题编号.])
  - C++ Vectors(见 [标题编号.])
  - Iterators(见 [标题编号.])

1.1 预处理命令

#, ##(见 [标题编号.])	manupilate 字符串
#define(见 [标题编号.])	定义变量
#error(见 [标题编号.])	显示一个错误消息
#if, #ifdef, #ifndef, #else, #elif, #endif(见 [标题编号.])	条件操作符
#include(见 [标题编号.])	插入其它文件的内容
#line(见 [标题编号.])	设置行和文件信息
#pragma(见 [标题编号.])	执行特殊命令
#undef(见 [标题编号.])	取消定义变量
预定义变量(见 [标题编号.])	其它变量

1.1.1 预处理命令

#,

和 ## 操作符是和#define 宏使用的. 使用# 使在#后的首个参数返回为一个带引号的字符串. 例如, 命令

```
#define to_string( s ) # s
```

将会使编译器把以下命令

```
cout << to_string( Hello World! ) << endl;
```

理解为

```
cout << "Hello World!" << endl;
```

使用##连结##前后的内容. 例如, 命令

```
#define concatenate( x, y ) x ## y
...
int xy = 10;
...
```

将会使编译器把

```
cout << concatenate( x, y ) << endl;
```

解释为

```
cout << xy << endl;
```

理所当然,将会在标准输出处显示'10'.

#define

语法:

```
#define macro-name replacement-string
```

#define 命令用于把指定的字符串替换文件中的宏名称 . 也就是说, #define 使编译器把文件中每一个 *macro-name* 替换为 *replacement-string*. 替换的字符串结束于行末. 这里是一个经典的#define 应用 (至少是在 C 中):

```
#define TRUE 1
#define FALSE 0
...
int done = 0;
while( done != TRUE ) {
   ...
}
```

#define 命令的另外一个功能就是替换参数,使它 假冒创建函数一样使用. 如下的代码:

```
#define absolute_value( x ) ( ((x) < 0) ? -(x) : (x) )
...
int x = -1;
while( absolute_value( x ) ) {
   ...
}
```

当使用复杂的宏时,最好使用额外的圆括号. 注意在以上的例子中, 变量"x"总是出现在它自己的括号中. 这样, 它就可以在和0比较,或变成负值(乘以-1)前计算值. 同样的, 整个宏也被括号围绕, 以防止和其它代码混淆. 如果你不注意的话,你可能会被编译器曲解你的代码.

#error

语法:

```
#error message
```

#error 命令可以简单的使编译器在发生错误时停止. 当遇到一个#error 时,编译器会自动输出行号而无论 *message* 的内容. 本命令大多是用于调试.

#if, #ifdef, #ifndef, #else, #elif, #endif

这些命令让编译器进行简单的逻辑控制. 当一个文件被编译时, 你可以使用这些命令使某些行保留或者是去处.

```
#if expression
```

如果表达式(expression)的值是"真"(true),那么紧随该命令的代码将会被编译.

```
#ifdef macro
```

如果"macro"已经在一个#define 声明中定义了, 那么紧随该命令的代码将会被编译.

```
#ifndef macro
```

如果"macro"未在一个#define 声明中定义, 那么紧随命令的代码将会被编译.

一些小边注: 命令#elif 是"elseif"的一种缩写,并且他可以想你所意愿的一样工作. 你也可以在一个#if 后插入一个"defined"或者"!defined"以获得更多的功能.

这里是一部分例子:

```
#ifdef DEBUG
  cout << "This is the test version, i="<< i << endl;
#else
  cout << "This is the production version!" << endl;
#endif
```

你应该注意到第二个例子比在你的代码中插入多个"cout"进行调试的方法更简单.

#include

语法:

```
#include <filename>
#include "filename"
```

本命令包含一个文件并在当前位置插入. 两种语法的主要不同之处是在于,如果 *filename* 括在尖括号中,那么编译器不知道如何搜索它. 如果它括在引号中, 那么编译器可以简单的搜索到文件. 两种搜索的方式是由编译器决定的,一般尖括号意味着在标准库目录中搜索, 引号就表示在当前目录中搜索. The spiffy new 整洁的新 C++ #include 目录不需要直接映射到 filenames, 至少对于标准库是这样. 这就是你有时能够成功编译以下命令的原因

```
#include <iostream>
```

#line

语法:

```
#line line_number "filename"
```

#line 命令是用于更改__LINE__ 和 __FILE__变量的值. 文件名是可选的. __LINE__ 和 __FILE__ 变量描述被读取的当前文件和行. 命令

```
#line 10 "main.cpp"
```

更改行号为 10,当前文件改为"main.cpp".

#pragma

#pragma 命令可以让编程者让编译器执行某些事. 因为#pragma 命令的执行很特殊, 不同的编译器使用有所不同. 一个选项可以跟踪程序的执行.

#undef

#undef 命令取消一个先前已定义的宏变量, 譬如一个用#define 定义的变量.

预定义变量

语法:

```
__LINE__
__FILE__
__DATE__
__TIME__
_cplusplus
__STDC__
```

下列参数在不同的编译器可能会有所不同, 但是一般是可用的:

- __LINE__ 和 __FILE__ 变量表示正在处理的当前行和当前文件.
- __DATE__ 变量表示当前日期,格式为 *month/day/year*(月/日/年).
- __TIME__ 变量描述当前的时间,格式为 *hour:minute:second*(时:分:秒).
- _cplusplus 变量只在编译一个 C++程序时定义.
- __STDC__ 变量在编译一个 C 程序时定义,编译 C++时也有可能定义.

1.2 操作符优先级

优先级	操作符
1	() [] -> . ::
	! ~ ++ --
2	- (unary) * (dereference)
	& (address of) sizeof
3	->* .*

4	* (multiply)	/ %
5	+	-
6	<<	>>
7	<	<= > >=
8	=	!=
9	& (bitwise AND)	
10	^	
11	|	
12	&&	
13	||	
14	? :	
15	=	+= -= etc.
16	,	

1.3 转义字符

以下是使用转义字符的代码示例:

```
printf( "This\nis\na\ntest\n\nShe said, \"How are you?\"\n" );
```

输出:

```
This
is
a
test

She said, "How are you?"
```

1.4 ASCII 码表

Ê®½øÖÆ	°Ë½øÖÆ	Ê®Áù½øÖÆ	×Ö·û	ÃèÊö
0	0	00	NUL	
1	1	01	SOH	start of header
2	2	02	STX	start of text
3	3	03	ETX	end of text
4	4	04	EOT	end of transmission

5	5	05	ENQ	enquiry
6	6	06	ACK	acknowledge
7	7	07	BEL	bell
8	10	08	BS	backspace
9	11	09	HT	horizontal tab
10	12	0A	LF	line feed
11	13	0B	VT	vertical tab
12	14	0C	FF	form feed
13	15	0D	CR	carriage return
14	16	0E	SO	shift out
15	17	0F	SI	shift in
16	20	10	DLE	data link escape
17	21	11	DC1	no assignment, but usually XON
18	22	12	DC2	
19	23	13	DC3	no assignment, but usually XOFF
20	24	14	DC4	
21	25	15	NAK	negative acknowledge
22	26	16	SYN	synchronous idle
23	27	17	ETB	end of transmission block
24	30	18	CAN	cancel
25	31	19	EM	end of medium
26	32	1A	SUB	substitute
27	33	1B	ESC	escape
28	34	1C	FS	file seperator
29	35	1D	GS	group seperator
30	36	1E	RS	record seperator
31	37	1F	US	unit seperator
32	40	20	SPC	space
33	41	21	!	
34	42	22	"	
35	43	23	#	
36	44	24	$	
37	45	25	%	
38	46	26	&	
39	47	27	'	
40	50	28	(	
41	51	29	)	

42	52	2A	*
43	53	2B	+
44	54	2C	,
45	55	2D	-
46	56	2E	.
47	57	2F	/
48	60	30	0
49	61	31	1
50	62	32	2
51	63	33	3
52	64	34	4
53	65	35	5
54	66	36	6
55	67	37	7
56	70	38	8
57	71	39	9
58	72	3A	:
59	73	3B	;
60	74	3C	<
61	75	3D	=
62	76	3E	>
63	77	3F	?
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N

79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	^
95	137	5F	_
96	140	60	`
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s

116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~
127	177	7F	DEL delete

1.5 基本数据类型

类型	描述
void	空类型
int	整型
float	浮点类型
double	双精度浮点类型
char	字符类型

C++ 定义了另外两个基本数据类型：**bool** 和 **wchar_t**.

类型	描述
bool	布尔类型, 值为 true 或 false
wchar_t	宽字符类型

类型修饰符

一些基本数据类型能够被 **signed**, **unsigned**, **short**, 和 **long** 修饰. 当类型修饰符单独使用的时候, 默认的类型是 **int**. 下表是所有可能出现的数据类型：

bool
char
unsigned char
signed char
int

unsigned int
signed int
short int
unsigned short int
signed short int
long int
signed long int
unsigned long int
float
double
long double
wchar_t

类型大小和表示范围

基本数据类型的大小以及能够表示的数据范围是与编译器和硬件平台有关的. "cfloat" (或者 "float.h") 头文件往往定义了基本数据类型能够表示的数据的最大值和最小值. 你也可以使用 sizeof(见 [标题编号.]) 来获得类型的大小(字节数). 然而, 很多平台使用了一些数据类型的标准大小, 如. **int** 和 **float** 通常占用 32 位, **char** 占用 8 位, **double** 通常占用 64 位. **bools** 通常以 8 位 来实现.

1.6 关键字

asm(见 [标题编号.])	插入一个汇编指令.
auto(见 [标题编号.])	声明一个本地变量.
bool(见 [标题编号.])	声明一个布尔型变量.
break(见 [标题编号.])	结束一个循环.
case(见 [标题编号.])	一个 switch(见 [标题编号.])语句的一部分.
catch(见 [标题编号.])	处理 thrown(见 [标题编号.]) 产生的异常.
char(见 [标题编号.])	声明一个字符型变量.
class(见 [标题编号.])	声明一个类.

const(见 [标题编号.])	声明一个常量.
const_cast(见 [标题编号.])	从一个 const 变量中抛出.
continue(见 [标题编号.])	在循环中间断循环.
default(见 [标题编号.])	是一个 case(见 [标题编号.])语句中的缺省项.
delete(见 [标题编号.])	释放内存.
do(见 [标题编号.])	构造循环体.
double(见 [标题编号.])	声明一个双精度浮点变量.
dynamic_cast(见 [标题编号.])	动态投射.
else(见 [标题编号.])	是一个 if(见 [标题编号.])语句中的预备条件.
enum(见 [标题编号.])	创建列举类型.
explicit(见 [标题编号.])	仅用在构造器的正确匹配.
extern(见 [标题编号.])	告诉编译器在别的地方变量已经被定义过了.
false(见 [标题编号.])	属于布尔值.
float(见 [标题编号.])	声明一个浮点型变量.
for(见 [标题编号.])	构造循环.
friend(见 [标题编号.])	允许非函数成员使用私有数据.
goto(见 [标题编号.])	跳转到程序的其它地方.
if(见 [标题编号.])	从一次判断的结果处执行代码.
inline(见 [标题编号.])	定义一个函数为内联.
int(见 [标题编号.])	声明一个整型变量.

long(见 [标题编号.])	声明一个长整型变量.
mutable(见 [标题编号.])	忽略 const 变量.
namespace(见 [标题编号.])	用一个定义的范围划分命名空间.
new(见 [标题编号.])	允许动态存储一个新变量.
operator(见 [标题编号.])	创建重载函数.
private(见 [标题编号.])	在一个类中声明私有成员.
protected(见 [标题编号.])	在一个类中声明被保护成员.
public(见 [标题编号.])	在一个类中声明公共成员.
register(见 [标题编号.])	定义一个寄存器变量.
reinterpret_cast(见 [标题编号.])	改变一个变量的类型.
return(见 [标题编号.])	从一个函数中返回.
short(见 [标题编号.])	声明一个短整型变量.
signed(见 [标题编号.])	修改变量类型声明.
sizeof(见 [标题编号.])	返回一个变量或类型的长度.
static(见 [标题编号.])	给一个变量创建永久的存储空间.
static_cast(见 [标题编号.])	执行一个非多态性 cast.
struct(见 [标题编号.])	创建一个新结构体.
switch(见 [标题编号.])	让一个变量在不同的判断下执行不同的代码.
template(见 [标题编号.])	创建一个给特殊函数.
this(见 [标题编号.])	指向当前对象.

throw(见 [标题编号.])	抛出一个异常.
true(见 [标题编号.])	布尔类型的一个值.
try(见 [标题编号.])	执行一个被 throw(见 [标题编号.]) 抛出的异常.
typedef(见 [标题编号.])	从现有的类型中创建一个新类型.
typeid(见 [标题编号.])	描述一个对象.
typename(见 [标题编号.])	声明一个类或未定义的类型.
union(见 [标题编号.])	一个结构体在当前位置分配给多个变量相同的内存.
unsigned(见 [标题编号.])	声明一个无符号整型变量.
using(见 [标题编号.])	用来输入一个 namespace(见 [标题编号.]).
virtual(见 [标题编号.])	创建一个不被已构成类有限考虑的函数.
void(见 [标题编号.])	声明函数或数据是无关联数据类型.
volatile(见 [标题编号.])	警告编译器有关的变量可能被出乎意料的修改.
wchar_t(见 [标题编号.])	声明一个带有宽度的字符型变量.
while(见 [标题编号.])	用来构成循环.

1.6.1 关键字

C/C++ 关键字

asm

语法:

```
asm( "instruction" );
```

asm 允许你在你的代码中直接插入汇编语言指令， 各种不同的编译器为这一个指令允许不一致形式, 比如：

```
asm {
  instruction-sequence
}
```

or

```
asm( instruction );
```

---

auto

关键字 auto 是用来声明完全可选择的局部变量的

---

bool

关键字 bool 是用来声明布尔逻辑变量的；也就是说,变量要么是真，要么是假。举个例子：

```
bool done = false;
while( !done ) {
...
}
```

你也可以查看 data types(见 [标题编号.]) 这一页.

---

break

关键字 break 是用来跳出一个 do, for, or while 的循环. 它也可以结束一个 switch 语句的子句,让程序忽略下面的 case 代码.举个例子:

```
while( x < 100 ) {
```

```
    if( x < 0 )
      break;
    cout << x << endl;
    x++;
  }
```

break 语句只能跳出本层循环,假如你要跳出一个三重嵌套的循环,你就要使用包含其它的逻辑或者用一个 goto 语句跳出这个嵌套循环.

---

case

在 switch 里面用来检测匹配 .

相关主题:
default, switch

---

catch

catch 语句通常通过 throw 语句捕获一个异常.

相关主题:
throw, try

---

char

关键字 char 用来声明布尔型变量. 你也可以查看 data types(见 [标题编号.]) 这一页...

---

class

语法:

```
class class-name : inheritance-list {

  private-members-list;

  protected:
    protected-members-list;
```

```
  public:
    public-members-list;

} object-list;
```

关键字 class 允许你创建新的数据类型. *class-name* 就是你要创建的类的名字, 并且 *inheritance-list* 是一个对你创建的新类可供选择的定义体的表单. 类的默认为私有类型成员, 除非这个表单标注在公有或保护类型之下. *object-list* 是一个或一组声明对象. 举个例子:

```
class Date {
  int Day;
  int Month;
  int Year;
public:
  void display();
};
```

相关主题:

struct, **union**

---

const

关键字 const 用来告诉编译器一个一旦被初始化过的变量就不能再修改.

相关主题:

const_cast

---

const_cast

语法:

```
const_cast<type> (object);
```

关键字 const 用于移除"const-ness"的数据, 目标数据类型必须和原类型相同, 目标数据没有被 const 定义过除外.

相关主题:

dynamic_cast, **reinterpret_cast**, **static_cast**

---

continue

continue 语句用来结束这次循环在一个循环语句中,例如,下面这段代码会显示所有除了 10 之外 0-20 的所有数字:

```
for( int i = 0; i < 21; i++ ) {
  if( i == 10 ) {
    continue;
  }
  cout << i << " ";
}
```

相关主题:

break, do, for, while

---

default

switch 语句中的缺省条件.

相关主题:

case, switch

---

delete

语法:

```
delete p;
delete[] pArray;
```

delete 操作用来释放 p 指向的内存.这个指针先前应该被 new 调用过.上面第二种形式用于删除一个数组.

相关主题:

new

---

do

语法:

```
do {
  statement-list;
} while( condition );
```

do 构建一个循环语句表,直到条件为假.注意循环中的语句至少被执行一次,因为判断条件在循环的最后.

相关主题:
for, while

---

double

关键字 double 用来声明浮点型变量的精度. 你也可以查看 data types(见 [标题编号.]) 这一页.

---

dynamic_cast

语法:

```
dynamic_cast<type> (object);
```

关键字 dynamic_cast 强制将一个类型转化为另外一种类型，并且在执行运行时时检查它保证它的合法性。如果你想在两个互相矛盾的类型之间转化时，cast 的返回值将为 NULL.

相关主题:
const_cast, reinterpret_cast, static_cast

---

else

关键字 else 用在 if 语句中的二中选一.

---

enum

语法:

```
enum name {name-list} var-list;
```

关键字 enum 用来创建一个包含多个名称元素的名称表. *var-list* 是可选的. 例如:

```
enum color {red, orange, yellow, green, blue, indigo, violet};
```

```
color c1 = indigo;
if( c1 == indigo ) {
  cout << "c1 is indigo" << endl;
}
```

---

explicit

当构造函数被指定为explicit的时候,将不会自动把构造函数作为转换构造函数,这仅仅用在当一个初始化语句参数与这个构造函数的形参匹配的情况.

---

extern

关键字extern用来告知编译器变量在当前范围之外声明过了.被extern语句描述过的变量将分派不到任何空间,因为他们在别的地方被定义过了.

Extern语句频繁的用于在多个文件之间的跨范围数据传递.

---

false

"false"是布尔型的值.

相关主题:
bool, **true**

---

float

关键字float用来声明浮点型变量. 你也可以查看 data types(见 [标题编号.]) 这一页.

---

for

语法:

```
for( initialization; test-condition; increment ) {
  statement-list;
}
```

for 构造一个由 4 部分组成的循环:

1. 初始化, 可以由 0 个或更多的由逗号分开的初始化语句构成;
2. 判断条件,如果满足该语句循环继续执行;
3. 增量, 可以由 0 个或更多的由逗号分开的增量语句构成;
4. 语句体,由 0 个或更多的语句构成,当循环条件成立时他们被执行.

例如:

```
for( int i = 0; i < 10; i++ ) {
  cout << "i is " << i << endl;
}

int j, k;
for( j = 0, k = 10;
     j < k;
     j++, k-- ) {
  cout << "j is " << j << " and k is " << k << endl;
}

for( ; ; ) {
  // loop forever!
}
```

相关主题:

do, while

---

friend

关键字 friend 允许类或函数访问一个类中的私有数据.

---

goto

语法:

```
goto labelA;

...

labelA:
```

goto 语句可以完成从当前位置到指定标志位的跳转. 使用 goto 语句要考虑有害性, 所以它不经常使用. 例如, goto 可以用来跳出多重嵌套 for 循环, 它比额外的逻辑性跳出更具有时效性.

相关主题:
break

---

if

语法:

```
if( conditionA ) {
  statement-listA;
}

else if( conditionB ) {
  statement-listB;
}

...

else {
  statement-listN;
}
```

if 构造一个允许不同的代码在不同的条件下执行的分支机制. *conditions* 是判断语句, *statement-list* . 假如条件为假, else 语句块将被执行, 所有的 else 是可选的.

相关主题:
else, **for**, **while**

---

inline

语法:

```
inline int functionA( int i ) {
  ...
}
```

inline 这个关键字请求编译器扩张一个给定的函数。它向这个函数发出一条插入代码的 call。函数里面有静态变量，嵌套的，switches，或者是递归的时候不给

予内联。当一个函数声明包含在一个类声明里面时，编译器会尝试的自动把函数内联。
关键字 inline 请求编译器给一个函数扩展空间,它向这个函数发出一条插入代码的 call.函数里面有 static 数据, 循环, switches, 或者是递归的时候不给予内联.当一个函数声明包含在一个类声明里面时,编译器会尝试的自动把函数内联.

---

int

关键字 int 用来声明整型变量. 你也可以查看 data types(见 [标题编号.]) 这一页.

---

long

关键字 keyword 用来修正数据类型,它用来声明长整型变量. 查看 data types(见 [标题编号.]) 这一页.

相关主题:
short

---

mutable

关键字 mutable 忽略所有 const 语句.一个属于 const 对象的 mutable 成员可以被修改.

---

namespace

语法:

```
namespace name {
  declaration-list;
}
```

关键字 namespace 允许你创建一个新的空间.名字由你选择,忽略创建没有命名的名字空间.一旦你创建了一个名字空间,你必须明确地说明它或者用关键字 using.例如:

```
namespace CartoonNameSpace {
  int HomersAge;
  void incrementHomersAge() {
    HomersAge++;
  }
}

int main() {
  ...
  CartoonNameSpace::HomersAge = 39;
  CartoonNameSpace::incrementHomersAge();
  cout << CartoonNameSpace::HomersAge << endl;
  ...
}
```

---

new

语法:

```
pointer = new type;
pointer = new type( initializer );
pointer = new type[size];
```

new 可以给数据类型分配一个新结点并返回一个指向新分配内存区的首地址. 也可以对它进行初始化.中括号中的 *size* 可以分配尺寸大小.

相关主题:

delete

---

operator

语法:

```
return-type class-name::operator#(parameter-list) {
  ...
}
return-type operator#(parameter-list) {
  ...
}
```

关键字 operator 用于重载函数.在上面语法中用特殊符(#)描述特征的操作将被重载.假如在一个类中,类名应当被指定.对于一元的操作, *parameter-list* 应当

为空, 对于二元的操作,在 operator 右边的 *parameter-list* 应当包含操作数(在 operand 左边的被当作 this 通过).

对于不属于重载函数的operator成员,在左边的操作数被作为第一个参数,在右边的操作数被当作第二个参数被通过.

你不能用**#**, **##**, **.**, **:**, **.***, 或者 **?** 标志重载.

---

private

属于私有类的数据只能被它的内部成员访问,除了 friend 使用.关键字 private 也能用来继承一个私有的基类,所有的公共和保护成员的基类可以变成私有的派生类.

相关主题:
protected, **public**

---

protected

保护数据对于它们自己的类是私有的并且能被派生类继承.关键字 keyword 也能用于指定派生,所有的公共和保护成员的基类可以变成保护的派生类.

相关主题:
private, **public**

---

public

在类中的公共数据可以被任何人访问.关键字public也能用来指定派生,所有的公共和保护成员的基类可以变成保护的派生类.

相关主题:
private, **protected**

---

register

关键字 register 请求编译器优化它定义的变量,并且通常这种优化比人工优化的好.

相关主题:
auto

---

reinterpret_cast

语法:

```
reinterpret_cast<type> (object);
```

reinterpret_cast 操作能把一种数据类型改变成另一种.它应当被用在两种不可调和的指针类型之间.

相关主题:

const_cast, dynamic_cast, static_cast

---

return

语法:

```
return;
return( value );
```

return 语句可以从当前函数跳转到调用该函数的任何地方.返回值是任意的.一个函数可以有不止一个返回语句.

---

short

关键字 short 用来修正数据类型,用来声明短整型变量.查看 data types(见 [标题编号.]) 这一页.

相关主题:

long

---

signed

关键字 signed 用来修正数据类型,用来声明符号字符型型变量. 查看 data types(见 [标题编号.]) 这一页.

相关主题:

unsigned

---

sizeof

sizeof 操作用来用字节计算右边表达式并返回字节数.

---

static

static 数据类型用来给变量创建永久存储空间. 静态变量在函数间调用时保持他们的值不变. 当用在一个类中时, 所有要用到静态变量的时候这个类将把这个变量镜像过去.

---

static_cast

语法:

```
static_cast<type> (object);
```

关键字 static_cast 用来在两个不同类型之间进行强制转换, 并且没有运行时间检查.

相关主题:

const_cast, **dynamic_cast**, **reinterpret_cast**

---

struct

语法:

```
struct struct-name : inheritance-list {

  public-members-list;

  protected:
    protected-members-list;

  private:
    private-members-list;

} object-list;
```

Structs 类似于 classes, struct 中的成员更像是类中的公共成员. 在 C 中, structs 仅能包含数据并不允许有继承表. 例如:

```
struct Date {
  int Day;
  int Month;
  int Year;
};
```

相关主题:

class, union

---

switch

语法:

```
switch( expression ) {
  case A:
    statement list;
    break;
  case B:
    statement list;
    break;
  ...
  case N:
    statement list;
    break;
  default:
    statement list;
    break;
}
```

switch 语句允许你通过一个表达式判断许多数值,它一般用来在多重循环中代替 if()...else if()...else if()... 语句. break 语句必须在每个 case 语句之后,负责循环将执行所有的 case 语句.default case 是可选的.假如所有的 case 都不能匹配的话,他将和 defult case 匹配. 例如:

```
char keystroke = getch();
switch( keystroke ) {
  case 'a':
  case 'b':
  case 'c':
  case 'd':
    KeyABCDPressed();
    break;
  case 'e':
```

```
      KeyEPressed();
      break;
    default:
      UnknownKeyPressed();
      break;
  }
```

相关主题:

break, **case**, **default**, **if**

---

template

语法:

```
template <class data-type> return-type name( parameter-list ) {
  statement-list;
}
```

Templates 能用来创建一个对未知数据类型的操作的函数模板.这个通过用其它数据类型代替一个占位符 *data-type* 来实现. 例如:

```
template<class X> void genericSwap( X &a, X &b ) {
  X tmp;

  tmp = a;
  a = b;
  b = tmp;
}

int main(void) {

  ...

  int num1 = 5;
  int num2 = 21;

  cout << "Before, num1 is " << num1 << " and num2 is " << num2 << endl;
  genericSwap( num1, num2 );
  cout << "After, num1 is " << num1 << " and num2 is " << num2 << endl;

  char c1 = 'a';
  char c2 = 'z';
```

```
    cout << "Before, c1 is " << c1 << " and c2 is " << c2 << endl;
    genericSwap( c1, c2 );
    cout << "After, c1 is " << c1 << " and c2 is " << c2 << endl;

    ...

    return( 0 );
  }
```

---

this

关键字 this 指向当前对象.所有属于一个 class 的函数成员都有一个 this 指向.

相关主题:

class

---

throw

语法:

```
try {
  statement list;
}

catch( typeA arg ) {
  statement list;
}

catch( typeB arg ) {
  statement list;
}

...

catch( typeN arg ) {
  statement list;
}
```

throw 在C++体系下用来处理异常.同 try 和 catch 语句一起使用, C++ 处理异常的系统给程序一个比较可行的机制用于错误校正.当你通常在用 try 去执行一

段有潜在错误的代码时.在代码的某一处,一个 **throw** 语句会被执行, 这将会从 try 的这一块跳转到 catch 的那一块中去. 例如:

```
try {
  cout << "Before throwing exception" << endl;
  throw 42;
  cout << "Shouldn't ever see this" << endl;
}

catch( int error ) {
  cout << "Error: caught exception " << error << endl;
}
```

相关主题:
catch, **try**

---

true

"true"是布尔型的值.

相关主题:
bool, **false**

---

try

try 语句试图去执行由异常产生的代码. 查看 throw 语句获得更多细节.

相关主题:
catch, **throw**

---

typedef

语法:

```
typedef existing-type new-type;
```

关键字 typedef 允许你从一个现有的类型中创建一个新类型.

---

typeid

语法:

```
typeid( object );
```

typeid 操作返回给一个 **type_info** 定义过的对象的那个对象的类型.

---

typename

关键字 typename 能用来在中 template 描述一个未定义类型或者代替关键字 class.

---

union

语法:

```
union union-name {
  public-members-list;
  private:
    private-members-list;
} object-list;
```

Unions 类似于 classes,除了所有的成员分享同一内存外它的缺省值更像公共类型. 例如:

```
union Data {
  int i;
  char c;
};
```

相关主题:

class, **struct**

---

unsigned

关键字 keyword 用来修正数据类型,它用来声明无符整型变量. 查看 data types(见 [标题编号.]) 这一页.

相关主题:

signed

---

using

关键字 keyword 用来在当前范围输入一个 namespace.

相关主题:

namespace

---

virtual

语法:

```
virtual return-type name( parameter-list );
virtual return-type name( parameter-list ) = 0;
```

关键字 virtual 能用来创建虚函数,它通常不被派生类有限考虑.但是假如函数被作为一个纯的虚函数 (被=0 表示)时, 这种情况它一定被派生类有限考虑.

---

volatile

关键字 volatile 在描述变量时使用,阻止编译器优化那些以 valatile 修饰的变量,volatile 被用在一些变量能被意外方式改变的地方,例如:抛出中断,这些变量若无 volatile 可能会和编译器执行的优化 相冲突.

---

void

关键字 keyword 用来表示一个函数不返回任何值,或者普通变量能指向任何类型的数据. Void 也能用来声明一个空参数表. 你也可以查看 data types(见 [标题编号.]) 这一页.

---

wchar_t

关键字 wchar_t 用来声明字符变量的宽度. 你也可以查看 data types(见 [标题编号.]) 这一页.

while

语法:

```
while( condition ) {
  statement-list;
}
```

关键字 while 用于一个只要条件未真就执行 *statement-list* 的循环体. 注意假如起始条件为 false, *statement-list* 将不被执行. (你可以用一个 do 循环来保证 statement-list 至少被执行一次.) 例如:

```
bool done = false;
while( !done ) {

  ProcessData();

  if( StopLooping() ) {
    done = true;
  }

}
```

相关主题:

do, for

cppreference.com(见 [标题编号.]) -> C/C++ 关键字(见 [标题编号.]) -> 细节

C/C++ 关键字

asm

语法:

```
asm( "instruction" );
```

asm 允许你在你的代码中直接插入汇编语言指令， 各种不同的编译器为这一个指令允许不一致形式, 比如:

```
asm {
```

```
    instruction-sequence
  }
```

or

```
  asm( instruction );
```

---

auto

关键字 auto 是用来声明完全可选择的局部变量的

---

bool

关键字 bool 是用来声明布尔逻辑变量的；也就是说,变量要么是真，要么是假。举个例子：

```
  bool done = false;
  while( !done ) {
  ...
  }
```

你也可以查看 data types(见 [标题编号.]) 这一页.

---

break

关键字 break 是用来跳出一个 do, for, or while 的循环. 它也可以结束一个 switch 语句的子句,让程序忽略下面的 case 代码.举个例子:

```
  while( x < 100 ) {
    if( x < 0 )
      break;
    cout << x << endl;
    x++;
  }
```

break 语句只能跳出本层循环,假如你要跳出一个三重嵌套的循环,你就要使用包含其它的逻辑或者用一个 goto 语句跳出这个嵌套循环.

---

case

在 switch 里面用来检测匹配 .

相关主题:
default, switch

---

catch

catch 语句通常通过 throw 语句捕获一个异常.

相关主题:
throw, try

---

char

关键字 char 用来声明布尔型变量. 你也可以查看 data types(见 [标题编号.]) 这一页...

---

class

语法:

```
class class-name : inheritance-list {

  private-members-list;

  protected:
    protected-members-list;

  public:
    public-members-list;

} object-list;
```

关键字 class 允许你创建新的数据类型. *class-name* 就是你要创建的类的名字,并且 *inheritance-list* 是一个对你创建的新类可供选择的定义体的表单.类的默认为私有类型成员,除非这个表单标注在公有或保护类型之下. *object-list* 是一个或一组声明对象.举个例子:

```
class Date {
  int Day;
  int Month;
  int Year;
public:
  void display();
};
```

相关主题:

struct, **union**

---

const

关键字 const 用来告诉编译器一个一旦被初始化过的变量就不能再修改.

相关主题:

const_cast

---

const_cast

语法:

```
const_cast<type> (object);
```

关键字 const 用于移除"const-ness"的数据,目标数据类型必须和原类型相同,目标数据没有被 const 定义过除外.

相关主题:

dynamic_cast, **reinterpret_cast**, **static_cast**

---

continue

continue 语句用来结束这次循环在一个循环语句中,例如,下面这段代码会显示所有除了 10 之外 0-20 的所有数字:

```
for( int i = 0; i < 21; i++ ) {
```

```
    if( i == 10 ) {
      continue;
    }
    cout << i << " ";
  }
```

相关主题:

break, do, for, while

---

default

switch 语句中的缺省条件.

相关主题:

case, switch

---

delete

语法:

```
delete p;
delete[] pArray;
```

delete 操作用来释放 p 指向的内存. 这个指针先前应该被 new 调用过. 上面第二种形式用于删除一个数组.

相关主题:

new

---

do

语法:

```
do {
  statement-list;
} while( condition );
```

do 构建一个循环语句表, 直到条件为假. 注意循环中的语句至少被执行一次, 因为判断条件在循环的最后.

相关主题:

for, while

---

double

关键字 double 用来声明浮点型变量的精度. 你也可以查看 data types(见 [标题编号.]) 这一页.

---

dynamic_cast

语法:

```
dynamic_cast<type> (object);
```

关键字 dynamic_cast 强制将一个类型转化为另外一种类型，并且在执行运行时时检查它保证它的合法性。如果你想在两个互相矛盾的类型之间转化时，cast 的返回值将为 NULL.

相关主题:

const_cast, reinterpret_cast, static_cast

---

else

关键字 else 用在 if 语句中的二中选一.

---

enum

语法:

```
enum name {name-list} var-list;
```

关键字 enum 用来创建一个包含多个名称元素的名称表. *var-list* 是可选的. 例如:

```
enum color {red, orange, yellow, green, blue, indigo, violet};

color c1 = indigo;
if( c1 == indigo ) {
  cout << "c1 is indigo" << endl;
}
```

---

explicit

当构造函数被指定为 explicit 的时候,将不会自动把构造函数作为转换构造函数,这仅仅用在当一个初始化语句参数与这个构造函数的形参匹配的情况.

---

extern

关键字 extern 用来告知编译器变量在当前范围之外声明过了. 被 extern 语句描述过的变量将分派不到任何空间,因为他们在别的地方被定义过了.

Extern 语句频繁的用于在多个文件之间的跨范围数据传递.

---

false

"false"是布尔型的值.

相关主题:
bool, true

---

float

关键字 float 用来声明浮点型变量. 你也可以查看 data types(见 [标题编号.]) 这一页.

---

for

语法:

```
for( initialization; test-condition; increment ) {
  statement-list;
}
```

for 构造一个由 4 部分组成的循环:

1. 初始化, 可以由 0 个或更多的由逗号分开的初始化语句构成;
2. 判断条件,如果满足该语句循环继续执行;
3. 增量, 可以由 0 个或更多的由逗号分开的增量语句构成;

4. 语句体,由 0 个或更多的语句构成,当循环条件成立时他们被执行.

例如:

```
for( int i = 0; i < 10; i++ ) {
  cout << "i is " << i << endl;
}

int j, k;
for( j = 0, k = 10;
     j < k;
     j++, k-- ) {
  cout << "j is " << j << " and k is " << k << endl;
}

for( ; ; ) {
  // loop forever!
}
```

相关主题:

do, **while**

---

friend

关键字 friend 允许类或函数访问一个类中的私有数据.

---

goto

语法:

```
goto labelA;

...

labelA:
```

goto 语句可以完成从当前位置到指定标志位的跳转. 使用 goto 语句要考虑有害性, 所以它不经常使用. 例如, goto 可以用来跳出多重嵌套 for 循环, 它比额外的逻辑性跳出更具有时效性.

相关主题:

break

---

if

语法:

```
if( conditionA ) {
  statement-listA;
}

else if( conditionB ) {
  statement-listB;
}

...

else {
  statement-listN;
}
```

if 构造一个允许不同的代码在不同的条件下执行的分支机制. *conditions* 是判断语句, *statement-list* . 假如条件为假, else 语句块将被执行, 所有的 else 是可选的.

相关主题:

else, for, while

---

inline

语法:

```
inline int functionA( int i ) {
  ...
}
```

inline 这个关键字请求编译器扩张一个给定的函数。它向这个函数发出一条插入代码的 call。函数里面有静态变量，嵌套的，switches，或者是递归的时候不给予内联。当一个函数声明包含在一个类声明里面时，编译器会尝试的自动把函数内联。

关键字 inline 请求编译器给一个函数扩展空间, 它向这个函数发出一条插入代码的 call. 函数里面有 static 数据, 循环, switches, 或者是递归的时候不给予内联. 当一个函数声明包含在一个类声明里面时, 编译器会尝试的自动把函数内联.

---

int

关键字 int 用来声明整型变量. 你也可以查看 data types(见 [标题编号.]) 这一页.

---

long

关键字 keyword 用来修正数据类型,它用来声明长整型变量. 查看 data types(见 [标题编号.]) 这一页.

相关主题:

short

---

mutable

关键字 mutable 忽略所有 const 语句.一个属于 const 对象的 mutable 成员可以被修改.

---

namespace

语法:

```
namespace name {
  declaration-list;
}
```

关键字 namespace 允许你创建一个新的空间.名字由你选择,忽略创建没有命名的名字空间.一旦你创建了一个名字空间,你必须明确地说明它或者用关键字 using.例如:

```
namespace CartoonNameSpace {
  int HomersAge;
  void incrementHomersAge() {
    HomersAge++;
  }
}

int main() {
```

```
    ...
    CartoonNameSpace::HomersAge = 39;
    CartoonNameSpace::incrementHomersAge();
    cout << CartoonNameSpace::HomersAge << endl;
    ...
}
```

---

new

语法:

```
pointer = new type;
pointer = new type( initializer );
pointer = new type[size];
```

new 可以给数据类型分配一个新结点并返回一个指向新分配内存区的首地址. 也可以对它进行初始化.中括号中的 *size* 可以分配尺寸大小.

相关主题:

delete

---

operator

语法:

```
return-type class-name::operator#(parameter-list) {
  ...
}
return-type operator#(parameter-list) {
  ...
}
```

关键字 operator 用于重载函数.在上面语法中用特殊符(#)描述特征的操作将被重载.假如在一个类中,类名应当被指定.对于一元的操作, *parameter-list* 应当为空, 对于二元的操作,在 operator 右边的 *parameter-list* 应当包含操作数(在 operand 左边的被当作 this 通过).

对于不属于重载函数的operator成员,在左边的操作数被作为第一个参数,在右边的操作数被当作第二个参数被通过.

你不能用**#**, **##**, **.**, **:**, **.***, 或者 **?** 标志重载.

---

private

属于私有类的数据只能被它的内部成员访问,除了 friend 使用.关键字 private 也能用来继承一个私有的基类,所有的公共和保护成员的基类可以变成私有的派生类.

相关主题:

protected, **public**

---

protected

保护数据对于它们自己的类是私有的并且能被派生类继承.关键字 keyword 也能用于指定派生,所有的公共和保护成员的基类可以变成保护的派生类.

相关主题:

private, **public**

---

public

在类中的公共数据可以被任何人访问.关键字public也能用来指定派生,所有的公共和保护成员的基类可以变成保护的派生类.

相关主题:

private, **protected**

---

register

关键字 register 请求编译器优化它定义的变量,并且通常这种优化比人工优化的好.

相关主题:

auto

---

reinterpret_cast

语法:

```
reinterpret_cast<type> (object);
```

reinterpret_cast 操作能把一种数据类型改变成另一种.它应当被用在两种不可调和的指针类型之间.

相关主题:

const_cast, **dynamic_cast**, **static_cast**

---

return

语法:

```
return;
return( value );
```

return 语句可以从当前函数跳转到调用该函数的任何地方.返回值是任意的.一个函数可以有不止一个返回语句.

---

short

关键字 short 用来修正数据类型,用来声明短整型变量.查看 data types(见 [标题编号.]) 这一页.

相关主题:

long

---

signed

关键字 signed 用来修正数据类型,用来声明符号字符型型变量. 查看 data types(见 [标题编号.]) 这一页.

相关主题:

unsigned

---

sizeof

sizeof 操作用来用字节计算右边表达式并返回字节数.

---

static

static 数据类型用来给变量创建永久存储空间.静态变量在函数间调用时保持他们的值不变.当用在一个类中时,所有要用到静态变量的时候这个类将把这个变量镜像过去.

---

static_cast

语法:

```
static_cast<type> (object);
```

关键字 static_cast 用来在两个不同类型之间进行强制转换,并且没有运行时间检查.

相关主题:

const_cast, **dynamic_cast**, **reinterpret_cast**

---

struct

语法:

```
struct struct-name : inheritance-list {

  public-members-list;

  protected:
    protected-members-list;

  private:
    private-members-list;

} object-list;
```

Structs 类似于 classes, struct 中的成员更像是类中的公共成员.在 C 中,structs 仅能包含数据并不允许有继承表. 例如:

```
struct Date {
  int Day;
  int Month;
  int Year;
};
```

相关主题:

class, union

---

switch

语法:

```
switch( expression ) {
  case A:
    statement list;
    break;
  case B:
    statement list;
    break;
  ...
  case N:
    statement list;
    break;
  default:
    statement list;
    break;
}
```

switch 语句允许你通过一个表达式判断许多数值,它一般用来在多重循环中代替 if()...else if()...else if()... 语句. break 语句必须在每个 case 语句之后,负责循环将执行所有的 case 语句.default case 是可选的.假如所有的 case 都不能匹配的话,他将和 defult case 匹配. 例如:

```
char keystroke = getch();
switch( keystroke ) {
  case 'a':
  case 'b':
  case 'c':
  case 'd':
    KeyABCDPressed();
    break;
  case 'e':
    KeyEPressed();
    break;
  default:
    UnknownKeyPressed();
    break;
```

```
}
```

相关主题:

break, case, default, if

---

template

语法:

```
template <class data-type> return-type name( parameter-list ) {
  statement-list;
}
```

Templates 能用来创建一个对未知数据类型的操作的函数模板.这个通过用其它数据类型代替一个占位符 *data-type* 来实现. 例如:

```
template<class X> void genericSwap( X &a, X &b ) {
  X tmp;

  tmp = a;
  a = b;
  b = tmp;
}

int main(void) {

  ...

  int num1 = 5;
  int num2 = 21;

  cout << "Before, num1 is " << num1 << " and num2 is " << num2 << endl;
  genericSwap( num1, num2 );
  cout << "After, num1 is " << num1 << " and num2 is " << num2 << endl;

  char c1 = 'a';
  char c2 = 'z';

  cout << "Before, c1 is " << c1 << " and c2 is " << c2 << endl;
  genericSwap( c1, c2 );
  cout << "After, c1 is " << c1 << " and c2 is " << c2 << endl;

  ...
```

```
  return( 0 );
}
```

this

关键字 this 指向当前对象.所有属于一个 class 的函数成员都有一个 this 指向.

相关主题:

class

throw

语法:

```
try {
  statement list;
}

catch( typeA arg ) {
  statement list;
}

catch( typeB arg ) {
  statement list;
}

...

catch( typeN arg ) {
  statement list;
}
```

throw 在 C++体系下用来处理异常.同 try 和 catch 语句一起使用, C++ 处理异常的系统给程序一个比较可行的机制用于错误校正.当你通常在用 try 去执行一段有潜在错误的代码时.在代码的某一处,一个 **throw** 语句会被执行, 这将会从 try 的这一块跳转到 catch 的那一块中去. 例如:

```
try {
  cout << "Before throwing exception" << endl;
```

```
    throw 42;
    cout << "Shouldn't ever see this" << endl;
  }

  catch( int error ) {
    cout << "Error: caught exception " << error << endl;
  }
```

相关主题:
catch, **try**

---

true

"true"是布尔型的值.

相关主题:
bool, **false**

---

try

try 语句试图去执行由异常产生的代码. 查看 throw 语句获得更多细节.

相关主题:
catch, **throw**

---

typedef

语法:

```
typedef existing-type new-type;
```

关键字 typedef 允许你从一个现有的类型中创建一个新类型.

---

typeid

语法:

```
typeid( object );
```

typeid 操作返回给一个 **type_info** 定义过的对象的那个对象的类型.

typename

关键字 typename 能用来在中 template 描述一个未定义类型或者代替关键字 class.

union

语法:

```
union union-name {

  public-members-list;

  private:
    private-members-list;

} object-list;
```

Unions 类似于 classes,除了所有的成员分享同一内存外它的缺省值更像公共类型. 例如:

```
union Data {
  int i;
  char c;
};
```

相关主题:

class, struct

unsigned

关键字 keyword 用来修正数据类型,它用来声明无符整型变量. 查看 data types(见 [标题编号.]) 这一页.

相关主题:

signed

using

关键字 keyword 用来在当前范围输入一个 namespace.

相关主题:

namespace

---

virtual

语法:

```
virtual return-type name( parameter-list );
virtual return-type name( parameter-list ) = 0;
```

关键字 virtual 能用来创建虚函数,它通常不被派生类有限考虑.但是假如函数被作为一个纯的虚函数 (被=0 表示)时, 这种情况它一定被派生类有限考虑.

---

volatile

关键字 volatile 在描述变量时使用,阻止编译器优化那些以 valatile 修饰的变量,volatile 被用在一些变量能被意外方式改变的地方,例如:抛出中断,这些变量若无 volatile 可能会和编译器执行的优化 相冲突.

---

void

关键字 keyword 用来表示一个函数不返回任何值,或者普通变量能指向任何类型的数据. Void 也能用来声明一个空参数表. 你也可以查看 data types(见 [标题编号.]) 这一页.

---

wchar_t

关键字 wchar_t 用来声明字符变量的宽度. 你也可以查看 data types(见 [标题编号.]) 这一页.

---

while

语法:

```
while( condition ) {
  statement-list;
}
```

关键字 while 用于一个只要条件未真就执行 *statement-list* 的循环体. 注意假如起始条件为 false, *statement-list* 将不被执行. (你可以用一个 do 循环来保证 statement-list 至少被执行一次.) 例如:

```
bool done = false;
while( !done ) {

  ProcessData();

  if( StopLooping() ) {
    done = true;
  }

}
```

相关主题:

do, for

1.7 Standard C I/O

clearerr()(见 [标题编号.])	清除错误
fclose()(见 [标题编号.])	关闭一个文件
feof()(见 [标题编号.])	如果到达文件尾(end-of-file)返回"True"(真)
ferror()(见 [标题编号.])	检查一个文件错误
fflush()(见 [标题编号.])	书写输出缓存的内容
fgetc()(见 [标题编号.])	从流获取一个字符
fgetpos()(见 [标题编号.])	获取文件位置指针
fgets()(见 [标题编号.])	从一个流获取一串字符
fopen()(见 [标题编号.])	打开一个文件
fprintf()(见 [标题编号.])	打印格式化的输出到一个文件
fputc()(见 [标题编号.])	写一个字符到一个文件
fputs()(见 [标题编号.])	写一个字符串到一个文件
fread()(见 [标题编号.])	从一个文件读取
freopen()(见 [标题编号.])	用一个不同的名称打开一个存在的流
fscanf()(见 [标题编号.])	从一个文件读取一个格式化的输入

fseek()(见 [标题编号.])	在文件中移动到一个指定的位置
fsetpos()(见 [标题编号.])	在一个文件中移动到一个指定的位置
ftell()(见 [标题编号.])	返回当前文件的位置指针
fwrite()(见 [标题编号.])	写入一个文件
getc()(见 [标题编号.])	从一个文件读取一个字符
getchar()(见 [标题编号.])	从 **STDIN(标准输入)**读取一个字符
gets()(见 [标题编号.])	从 **STDIN(标准输入)**读取一个字符串
perror()(见 [标题编号.])	显示当前错误的一个字符串版本到 **STDERR**(标准错误输出)
printf()(见 [标题编号.])	写格式化的输出到 **STDOUT**(标准输出)
putc()(见 [标题编号.])	写一个字符到一个流
putchar()(见 [标题编号.])	写一个字符到 **STDOUT**(标准输出)
puts()(见 [标题编号.])	写一个字符串到 **STDOUT**(标准输出)
remove()(见 [标题编号.])	清除一个文件
rename()(见 [标题编号.])	重命名一个文件
rewind()(见 [标题编号.])	移动文件位置指针到一个文件的开始处
scanf()(见 [标题编号.])	从 **STDIN**(标准输入)读取格式化输入
setbuf()(见 [标题编号.])	设置一个指定流的缓冲区
setvbuf()(见 [标题编号.])	设置一个指定流的缓冲区和大小
sprintf()(见 [标题编号.])	写格式化的输出到缓冲区
sscanf()(见 [标题编号.])	从一个缓冲区读取格式化的输入
tmpfile()(见 [标题编号.])	返回一个到一个临时文件的指针
tmpnam()(见 [标题编号.])	返回一个独特的文件名
ungetc()(见 [标题编号.])	把一个字符放回一个流
vprintf, vfprintf, vsprintf(见 [标题编号.])	写用参数列表格式化输出

1.7.1 Standard C I/O

---

clearerr

语法:

```
#include <stdio.h>
void clearerr( FILE *stream );
```

clearerr 函数重置错误标记和给出的流的 EOF 指针. 当发生错误时,你可以使用 perror()判断实际上发生了何种错误.

相关主题:

feof(), ferror(), 和 perror().

---

fclose

语法:

```
#include <stdio.h>
int fclose( FILE *stream );
```

函数 fclose()关闭给出的文件流, 释放已关联到流的所有缓冲区. fclose()执行成功时返回 0,否则返回 EOF.

相关主题:

fopen(), freopen(), 和 fflush().

---

feof

语法:

```
#include <stdio.h>
int feof( FILE *stream );
```

函数 feof()在到达给出的文件流的文件尾时返回一个非零值.

相关主题:

clearerr(), ferror(), perror(), putc()和 getc().

---

ferror

语法:

```
#include <stdio.h>
int ferror( FILE *stream );
```

ferror()函数检查 *stream*(流)中的错误, 如果没发生错误返回 0,否则返回非零. 如果发生错误, 使用 perror()检测发生什么错误.

相关主题:

clearerr(), feof(), perror(),

---

fflush

语法:

```
#include <stdio.h>
int fflush( FILE *stream );
```

如果给出的文件流是一个输出流,那么 fflush()把输出到缓冲区的内容写入文件. 如果给出的文件流是输入类型的,那么 fflush()会清除输入缓冲区. fflush()在调试时很实用,特别是对于在程序中输出到屏幕前发生错误片段时. 直接调用 fflush(STDOUT)输出可以保证你的调试输出可以在正确的时间输出.

```
printf( "Before first call\n" );
fflush( STDOUT );
shady_function();
printf( "Before second call\n" );
fflush( STDOUT );
dangerous_dereference();
```

相关主题:

fclose(), fopen(), fread(), fwrite(), getc(), 和 putc().

---

fgetc

语法:

```
#include <stdio.h>
int fgetc( FILE *stream );
```

fgetc()函数返回来自 *stream*(流)中的下一个字符,如果到达文件尾或者发生错误时返回 EOF.

相关主题:

fputc(), getc(), putc(), 和 fopen().

---

fgetpos

语法:

```
#include <stdio.h>
int fgetpos( FILE *stream, fpos_t *position );
```

fgetpos()函数保存给出的文件流(stream)的位置指针到给出的位置变量(position)中. position 变量是 fpos_t 类型的(它在 stdio.h 中定义)并且是可以控制在 FILE 中每个可能的位置对象. fgetpos()执行成功时返回 0,失败时返回一个非零值.

相关主题:

fsetpos(), fseek()和 ftell().

---

fgets

语法:

```
#include <stdio.h>
char *fgets( char *str, int num, FILE *stream );
```

函数 fgets()从给出的文件流中读取[*num - 1*]个字符并且把它们转储到 *str*(字符串)中. fgets()在到达行末时停止,在这种情况下,*str*(字符串)将会被一个新行符结束. 如果 fgets()达到[*num - 1*]个字符或者遇到 EOF, *str*(字符串)将会以 null 结束.fgets()成功时返回 *str*(字符串),失败时返回 NULL.

---

fopen

语法:

```
#include <stdio.h>
FILE *fopen( const char *fname, const char *mode );
```

fopen()函数打开由 *fname*(文件名)指定的文件, 并返回一个关联该文件的流.如果发生错误, fopen()返回 NULL. *mode*(方式)是用于决定文件的用途(例如 用于输入,输出,等等)

Mode(方式)	**意义**
"r"	打开一个用于读取的文本文件
"w"	创建一个用于写入的文本文件
"a"	附加到一个文本文件
"rb"	打开一个用于读取的二进制文件

"wb"	创建一个用于写入的二进制文件
"ab"	附加到一个二进制文件
"r+"	打开一个用于读/写的文本文件
"w+"	创建一个用于读/写的文本文件
"a+"	打开一个用于读/写的文本文件
"rb+"	打开一个用于读/写的二进制文件
"wb+"	创建一个用于读/写的二进制文件
"ab+"	打开一个用于读/写的二进制文件

示例:

```
char ch;
FILE *input = fopen( "stuff", "r" );
ch = getc( input );
```

---

fprintf

语法:

```
#include <stdio.h>
int fprintf( FILE *stream, const char *format, ... );
```

fprintf()函数根据指定的 *format*(格式)(格式)发送信息(参数)到由 *stream*(流)指定的文件. fprintf()只能和 printf()一样工作. fprintf()的返回值是输出的字符数,发生错误时返回一个负值.

示例:

```
char name[20] = "Mary";
FILE *out;
out = fopen( "output.txt", "w" );
if( out != NULL )
  fprintf( out, "Hello %s\n", name );
```

相关主题:

printf() 和 fscanf().

---

fputc

语法:

```
#include <stdio.h>
int fputc( int ch, FILE *stream );
```

函数 fputc()把给出的字符 *ch* 写到给出的输出流. 返回值是字符, 发生错误时返回值是 EOF.

相关主题:

fgetc(), fopen(), fprintf(), fread(), 和 fwrite().

---

fputs

语法:

```
#include <stdio.h>
int fputs( const char *str, FILE *stream );
```

fputs()函数把 *str*(字符串)指向的字符写到给出的输出流. 成功时返回非负值, 失败时返回 EOF.

相关主题:

fgets(), gets(), puts(), fprintf(), 和 fscanf().

---

fread

语法:

```
#include <stdio.h>
int fread( void *buffer, size_t size, size_t num, FILE *stream );
```

函数 fread()读取[*num*]个对象(每个对象大小为 *size*(大小)指定的字节数),并把它们替换到由 *buffer*(缓冲区)指定的数组. 数据来自给出的输入流. 函数的返回值是读取的内容数量...

使用 feof()或 ferror()判断到底发生哪个错误.

相关主题:

fwrite(), fopen(),fscanf(), fgetc()和 getc().

---

freopen

语法:

```
#include <stdio.h>
FILE *freopen( const char *fname, const char *mode, FILE *stream );
```

freopen()函数常用于再分配一个以存在的流给一个不同的文件和方式(mode).在调用本函数后,给出的文件流将会用 *mode*(方式)指定的访问模式引用 *fname*(文件名). freopen()的返回值是新的文件流,发生错误时返回 NULL.

相关主题:

fopen() 和 fclose().

---

fscanf

语法:

```
#include <stdio.h>
int fscanf( FILE *stream, const char *format, ... );
```

函数 fscanf()以 scanf()的执行方式从给出的文件流中读取数据. fscanf()的返回值是事实上已赋值的变量的数,如果未进行任何分配时返回 EOF.

相关主题:

scanf() 和 fprintf().

---

fseek

语法:

```
#include <stdio.h>
int fseek( FILE *stream, long offset, int origin );
```

函数 fseek()为给出的流设置位置数据. *origin* 的值应该是下列值其中之一(在 stdio.h 中定义):

名称	说明
SEEK_SET	从文件的开始处开始搜索
SEEK_CUR	从当前位置开始搜索
SEEK_END	从文件的结束处开始搜索

fseek()成功时返回 0,失败时返回非零. 你可以使用 fseek()移动超过一个文件,但是不能在开始处之前. 使用 fseek()清除关联到流的 EOF 标记.

相关主题:

ftell(), rewind(), fopen(), fgetpos()和 fsetpos().

---

fsetpos

语法:

```
#include <stdio.h>
int fsetpos( FILE *stream, const fpos_t *position );
```

fsetpos()函数把给出的流的位置指针移到由 *position* 对象指定的位置. fpos_t 是在 stdio.h 中定义的. fsetpos()执行成功返回 0,失败时返回非零.

相关主题:

fgetpos(), fseek()和 ftell().

---

ftell

语法:

```
#include <stdio.h>
long ftell( FILE *stream );
```

ftell()函数返回 *stream*(流)当前的文件位置,如果发生错误返回-1.

相关主题:

fseek()和 fgetpos().

---

fwrite

语法:

```
#include <stdio.h>
int fwrite( const void *buffer, size_t size, size_t count, FILE
*stream );
```

fwrite()函数从数组 *buffer*(缓冲区)中, 写 *count* 个大小为 *size*(大小)的对象到 *stream*(流)指定的流. 返回值是已写的对象的数量.

相关主题:

fread(), fscanf(), getc()和 fgetc().

---

getc

语法:

```
#include <stdio.h>
int getc( FILE *stream );
```

getc()函数从 *stream*(流)获取并返回下一个字符,如果到达文件尾返回 EOF.
getc()和 fgetc()是一样的. 例如:

```
char ch;
FILE *input = fopen( "stuff", "r" );

ch = getc( input );
while( ch != EOF ) {
  printf( "%c", ch );
  ch = getc( input );
}
```

相关主题:

fputc(), fgetc(), putc()和 fopen().

---

getchar

语法:

```
#include <stdio.h>
int getchar( void );
```

getchar()函数从 STDIN(标准输入)获取并返回下一个字符,如果到达文件尾返回 EOF.

相关主题:

fputc(), fgetc(), putc()和 fopen().

---

gets

语法:

```
#include <stdio.h>
char *gets( char *str );
```

gets()函数从 STDIN(标准输入)读取字符并把它们加载到 *str*(字符串)里,直到遇到新行(\n)或到达 EOF. 新行字符翻译为一个 null 中断符. gets()的返回值是读入的字符串,如果错误返回 NULL.

相关主题:

fputs(), fgetc(),fgets()和 puts().

---

perror

语法:

```
#include <stdio.h>
void perror( const char *str );
```

perror()函数打印 *str*(字符串)和一个相应的执行定义的错误消息到全局变量 **errno** 中.

---

printf

语法:

```
#include <stdio.h>
int printf( const char *format, ... );
```

printf()函数根据 *format*(格式)给出的格式打印输出到STDOUT(标准输出)和其它参数中.

字符串 *format*(格式)由两类项目组成 - 显示到屏幕上的字符和定义 printf()显示的其它参数. 基本上, 你可以指定一个包含文本在内的 *format*(格式)字符串,也可以是映射到 printf()其它参数的"特殊"字符. 例如本代码

```
char name[20] = "Bob";
int age = 21;
printf( "Hello %s, you are %d years old\n", name, age );
```

显示下列输出:

```
Hello Bob, you are 21 years old
```

%s 表示, ″在这里插入首个参数,一个字符串.″ %d 表示第二个参数(一个整数)应该放置在那里. 不同的″%-*codes*″表示不同的变量类型, 也可以限制变量的长度.

Code 格式	
%c	字符
%d	带符号整数
%i	带符号整数
%e	科学计数法, 使用小写"e"
%E	科学计数法, 使用大写"E"
%f	浮点数
%g	使用%e 或%f 中较短的一个
%G	使用%E 或%f 中较短的一个
%o	八进制
%s	一串字符
%u	无符号整数
%x	无符号十六进制数, 用小写字母
%X	无符号十六进制数, 用大写字母
%p	一个指针
%n	参数应该是一个指向一个整数的指针 指向的是字符数放置的位置
%%	一个'%'符号

一个位于一个%和格式化命令间的整数担当着一个最小字段宽度说明符,并且加上足够多的空格或 0 使输出足够长. 如果你想填充 0,在最小字段宽度说明符前放置 0. 你可以使用一个精度修饰符,它可以根据使用的格式代码而有不同的含义.

- 用%e, %E 和 %f,精度修饰符让你指定想要的小数位数. 例如,
-
- %12.6f

  将会至少显示 12 位数字,并带有 6 位小数的浮点数.

- 用%g 和 %G, 精度修饰符决定显示的有效数的位数最大值.
- 用%s,精度修饰符简单的表示一个**最大**的最大长度, 以补充句点前的最小字段长度.
  所有的 printf()的输出都是右对齐的,除非你在%符号后放置了负号. 例如,
-
- %-12.4f

将会显示 12 位字符,4 位小数位的浮点数并且左对齐. 你可以修改带字母 l 和 h%d, %i, %o, %u 和 %x 等类型说明符指定长型和短型数据类型 (例如 %hd 表示一个短整数). %e, %f 和 %g 类型说明符,可以在它们前面放置 l 指出跟随的是一个 double. %g, %f 和 %e 类型说明符可以置于字符'#'前保证出现小数点, 即使没有小数位. 带%x 类型说明符的'#'字符的使用, 表示显示十六进制数时应该带'0x'前缀. 带%o 类型说明符的'#'字符的使用, 表示显示八进制数时应该带一个'0'前缀.

你可以在输出字符串中包含 连续的 Escape 序列(见 [标题编号.]).

printf()的返回值是打印的字符数,如果发生错误则返回一个负值.

相关主题:

scanf()和 fprintf().

---

putc

语法:

```
#include <stdio.h>
int putc( int ch, FILE *stream );
```

putc()函数把字符 *ch* 写到 *stream*(流)中. 返回值是写入的字符, 发生错误时返回 EOF. 例如:

```
char ch;
FILE *input;
input = fopen( "temp.cpp", "r" );
ch = getc( input );
while( ch != EOF ) {
  printf( "%c", ch );
  ch = getc( input );
}
```

显示"temp.cpp"的内容到屏幕.

相关主题:

fgetc(), fputc(), getchar()和 putchar().

---

putchar

语法:

```
#include <stdio.h>
int putchar( int ch );
```

putchar()函数把 *ch* 写到 **STDOUT(标准输出)**. 代码

```
putchar( ch );
```

和

```
putc( ch, STDOUT );
```

一样.

putchar()的返回值是被写的字符, 发生错误时返回 EOF.

相关主题:
putc()

---

puts

语法:

```
#include <stdio.h>
int puts( char *str );
```

函数 puts()把 *str*(字符串)写到 **STDOUT(标准输出)**上. puts() 成功时返回非负值, 失败时返回 EOF.

相关主题:
putc(), gets()和 printf().

---

remove

语法:

```
#include <stdio.h>
int remove( const char *fname );
```

remove()函数删除由 *fname*(文件名)指定的文件. remove()成功时返回 0, 如果发生错误返回非零.

相关主题:
rename()

---

rename

语法:

```
#include <stdio.h>
int rename( const char *oldfname, const char *newfname );
```

函数 rename()更改文件 *oldfname* 的名称为 *newfname*. rename()成功时返回 0, 错误时返回非零.

相关主题:
remove()

---

rewind

语法:

```
#include <stdio.h>
void rewind( FILE *stream );
```

函数rewind()把文件指针移到由*stream*(流)指定的开始处，同时清除和流相关的错误和EOF标记.

相关主题:

fseek()

---

scanf

语法:

```
#include <stdio.h>
int scanf( const char *format, ... );
```

scanf()函数根据由 *format*(格式)指定的格式从 **stdin**(标准输入)读取,并保存数据到其它参数. 它和 printf()有点类似. *format*(格式)字符串由控制字符,空白字符和非空白字符组成. 控制字符以一个%符号开始,如下:

控制字符	说明
%c	一个单一的字符
%d	一个十进制整数
%i	一个整数
%e, %f, %g	一个浮点数
%o	一个八进制数
%s	一个字符串
%x	一个十六进制数
%p	一个指针
%n	一个等于读取字符数量的整数
%u	一个无符号整数
%[]	一个字符集
%%	一个精度符号

scanf()读取匹配 *format*(格式)字符串的输入. 当读取到一个控制字符,它把值放置到下一个变量. 空白(tabs, 空格等等)会跳过. 非空白字符和输入匹配, 然后丢弃. 如果是一个在%符号和控制符间的数量, 那么只有指定数量的字符转换到变量中. 如果 scanf()遇到一个字符集(用%[]控制字符表示), 那么在括号中的任意字符都会读取到变量中. scanf()的返回值是成功赋值的变量数量, 发生错误时返回EOF.

相关主题:
printf()和 fscanf().

---

setbuf

语法:

```
#include <stdio.h>
void setbuf( FILE *stream, char *buffer );
```

setbuf()函数设置 *stream*(流)使用 *buffer*(缓冲区),如果 *buffer*(缓冲区)是 null,关闭缓冲. 如果使用非标准缓冲尺寸, 它应该由 **BUFSIZ** 字符决定长度.

相关主题:
fopen(), fclose(), setvbuf(),

---

setvbuf

语法:

```
#include <stdio.h>
int setvbuf( FILE *stream, char *buffer, int mode, size_t size );
```

函数 setvbuf()设置用于 *stream*(流)的缓冲区到 *buffer*(缓冲区),其大小为 *size*(大小). *mode*(方式)可以是:

- _IOFBF, 表示完全缓冲
- _IOLBF, 表示线缓冲
- _IONBF, 表示无缓存

相关主题:
setbuf(),

---

sprintf

语法:

```
#include <stdio.h>
int sprintf( char *buffer, const char *format, ... );
```

sprintf()函数和printf()类似, 只是把输出发送到*buffer*(缓冲区)中.返回值是写入的字符数量. 例如:

```
char string[50];
int file_number = 0;

sprintf( string, "file.%d", file_number );
file_number++;
output_file = fopen( string, "w" );
```

相关主题:
printf(), fsprintf(),

---

sscanf

语法:

```
#include <stdio.h>
int sscanf( const char *buffer, const char *format, ... );
```

函数 sscanf()和 scanf()类似, 只是输入从 *buffer*(缓冲区)中读取.

相关主题:
scanf(), fscanf(),

---

tmpfile

语法:

```
#include <stdio.h>
FILE *tmpfile( void );
```

函数 tempfile()用一个独特的文件名打开一个临时文件,并返回一个到该文件的指针.如果发生错误则返回 null.

相关主题:
tmpnam(),

---

tmpnam

语法:

```
#include <stdio.h>
char *tmpnam( char *name );
```

tmpnam()函数创建一个独特的文件名并保存在 *name* 中. tmpnam()最多可以调用 **TMP_MAX** 指定的次数.

相关主题:
tmpfile(),

---

ungetc

语法:

```
#include <stdio.h>
int ungetc( int ch, FILE *stream );
```

函数 ungetc()把字符 *ch* 放回到 *stream*(流)中.

相关主题:

getc(),

---

vprintf, vfprintf 和 vsprintf

语法:

```
#include <stdarg.h>
#include <stdio.h>
int vprintf( char *format, va_list arg_ptr );
int vfprintf( FILE *stream, const char *format, va_list arg_ptr );
int vsprintf( char *buffer, char *format, va_list arg_ptr );
```

这些函数和 printf() 非常相似, fprintf() 和 sprintf() 的不同在于参数列表是一个指向一系列参数的指针. **va_list** 在 STDARG.H 中定义, 并且也可以被 va_arg() (见 [标题编号.]) 使用. 例如:

```
void error( char *fmt, ... ) {
  va_list args;

  va_start( args, fmt );
  fprintf( stderr, "Error: " );
  vfprintf( stderr, fmt, args );
  fprintf( stderr, "\n" );
  va_end( args );
  exit( 1 );
}
```

1.8 Standard C String & Character

atof() (见 [标题编号.])

将字符串转换成浮点数

atoi() (见 [标题编号.])

将字符串转换成整数

atol()(见 [标题编号.])

将字符串转换成长整型数

isalnum()(见 [标题编号.])

当字母或数字字符时，返回真值

isalpha()(见 [标题编号.])

当字母字符时，返回真值

iscntrl()(见 [标题编号.])

当控制字符时，返回真值

isdigit()(见 [标题编号.])

当数字字符时，返回真值

isgraph()(见 [标题编号.])

当非空格可打印字符时，返回真值

islower()(见 [标题编号.])

当小写字母字符时，返回真值

isprint()(见 [标题编号.])

当可打印字符时，返回真值

ispunct()(见 [标题编号.])

当标点字符时，返回真值

isspace()(见 [标题编号.])

当空格字符时，返回真值

isupper()(见 [标题编号.])

当大写字母字符时，返回真值

isxdigit()(见 [标题编号.])

当十六进制字符时，返回真值

memchr()(见 [标题编号.])

在某一内存范围中查找一特定字符

memcmp()(见 [标题编号.])

比较内存内容

memcpy()(见 [标题编号.])

拷贝内存内容

memmove()(见 [标题编号.])

拷贝内存内容

memset()(见 [标题编号.])

将一段内存空间填入某值

strcat()(见 [标题编号.])

连接两个字符串

strchr()(见 [标题编号.])

查找某字符在字符串中首次出现的位置

strcmp()(见 [标题编号.])

比较两个字符串

strcoll()(见 [标题编号.])

采用目前区域的字符排列次序来比较字符串

strcpy()(见 [标题编号.])

拷贝字符串

strcspn()(见 [标题编号.])

在某字符串中匹配指定字符串

strerror()(见 [标题编号.])

返回错误码对应的文本信息

strlen()(见 [标题编号.])

返回指定字符串的长度

strncat()(见 [标题编号.])

连接某一长度的两个字符串

strncmp()(见 [标题编号.])

比较某一长度的两个字符串

strncpy()(见 [标题编号.])

复制某一长度的一个字符串到另一字符串中

strpbrk()(见 [标题编号.])

查找某字符串在另一字符串中首次出现的位置

strrchr()(见 [标题编号.])

查找某字符在字符串中末次出现的位置

strspn()(见 [标题编号.])

返回子串的长度，子串的字符都出现包含于另一字符串中

strstr()(见 [标题编号.])

在一字符串中查找指定的子串首次出现的位置

strtod()(见 [标题编号.])

将字符串转换成浮点数

strtok()(见 [标题编号.])

查找指定字符之前的子串

strtol()(见 [标题编号.])

将字符串转换成长整型数

strtoul()(见 [标题编号.])

将字符串转换成无符号长整型数

strxfrm()(见 [标题编号.])

转换子串, 可以用于字符串比较

tolower()(见 [标题编号.])

将字符转换成小写字符

toupper()(见 [标题编号.])

将字符转换成大写字符

1.8.1 Standard C String & Character

标准 c 字符和字符串

atof

语法:

```
#include <stdlib.h>
double atof( const char *str );
```

功能:将字符串 *str* 转换成一个双精度数值并返回结果。 参数 *str* 必须以有效数字开头,但是允许以“E”或“e”除外的任意非数字字符结尾。例如:

```
x = atof( "42.0is_the_answer" );
```

x 的值为 42.0.

相关主题:
atoi() and atol().

atoi

语法:

```
#include <stdlib.h>
```

```
int atoi( const char *str );
```

功能：将字符串 *str* 转换成一个整数并返回结果。参数 *str* 以数字开头，当函数从 *str* 中读到非数字字符则结束转换并将结果返回。例如，

```
i = atoi( "512.035" );
```

i 的值为 512.

相关主题:
atof() and atol().

---

atol

语法:

```
#include <stdlib.h>
long atol( const char *str );
```

功能：将字符串转换成长整型数并返回结果。函数会扫描参数 *str* 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时才结束转换，并将结果返回。例如，

```
x = atol( "1024.0001" );
```

x 的值为 1024L.

相关主题:
atof() and atoi().

---

isalnum

语法:

```
#include <ctype.h>
int isalnum( int ch );
```

功能：如果参数是数字或字母字符，函数返回非零值，否则返回零值。

```
char c;
scanf( "%c", &c );
if( isalnum(c) )
  printf( "You entered the alphanumeric character %c\n", c );
```

相关主题:
isalpha(), iscntrl(), isdigit(), isgraph(), isprint(), ispunct(), and isspace().

---

isalpha

语法:

```
#include <ctype.h>
int isalpha( int ch );
```

功能：如果参数是字母字符，函数返回非零值，否则返回零值。

```
char c;
scanf( "%c", &c );
if( isalpha(c) )
  printf( "You entered a letter of the alphabet\n" );
```

相关主题:
isalnum(), iscntrl(), isdigit(), isgraph(), isprint(), ispunct(), and isspace().

---

iscntrl

语法:

```
#include <ctype.h>
int iscntrl( int ch );
```

功能：如果参数是控制字符（0 和 0x1F 之间的字符，或者等于 0x7F）函数返回非零值，否则返回零值。

相关主题:
isalnum(), isalpha(), isdigit(), isgraph(), isprint(), ispunct(), and isspace().

---

isdigit

语法:

```
#include <ctype.h>
int isdigit( int ch );
```

功能：如果参数是 0 到 9 之间的数字字符，函数返回非零值，否则返回零值.

```
char c;
scanf( "%c", &c );
if( isdigit(c) )
  printf( "You entered the digit %c\n", c );
```

相关主题:
isalnum(), isalpha(), iscntrl(), isgraph(), isprint(), ispunct(), and isspace().

---

isgraph

语法:

```
#include <ctype.h>
int isgraph( int ch );
```

功能：如果参数是除空格外的可打印字符（可见的字符），函数返回非零值，否则返回零值。

相关主题:
isalnum(), isalpha(), iscntrl(), isdigit(), isprint(), ispunct(), and isspace().

---

islower

语法:

```
#include <ctype.h>
```

```
int islower( int ch );
```

功能：如果参数是小写字母字符，函数返回非零值，否则返回零值。

相关主题:
isupper()

---

isprint

语法:

```
#include <ctype.h>
int isprint( int ch );
```

功能：如果参数是可打印字符（包括空格），函数返回非零值，否则返回零值。

相关主题:
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), ispunct(), and isspace().

---

ispunct

语法:

```
#include <ctype.h>
int ispunct( int ch );
```

功能：如果参数是除字母，数字和空格外可打印字符，函数返回非零值，否则返回零值。

相关主题:
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), isprint(), and isspace().

---

isspace

语法:

```
```

```
#include <ctype.h>
int isspace( int ch );
```

功能：如果参数是空格类字符（即：单空格，制表符，垂直制表符，满页符，回车符，新行符），函数返回非零值，否则返回零值。

相关主题:
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), and ispunct().

---

isupper

语法:

```
#include <ctype.h>
int isupper( int ch );
```

功能：如果参数是大写字母字符，函数返回非零值，否则返回零值。

相关主题:
tolower()

---

isxdigit

语法:

```
#include <ctype.h>
int isxdigit( int ch );
```

功能：如果参数是十六进制数字字符（即：A-F, a-f, 0-9），函数返回非零值，否则返回零值。

相关主题:
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), ispunct(), and isspace().

---

memchr

语法:

```
#include <string.h>
void *memchr( const void *buffer, int ch, size_t count );
```

功能：函数在 *buffer* 指向的数组的 *count* 个字符的字符串里查找 *ch* 首次出现的位置。返回一个指针，指向 *ch* 在字符串中首次出现的位置，如果 *ch* 没有在字符串中找到，返回 NULL。例如:

```
char names[] = "Alan Bob Chris X Dave";
if( memchr(names,'X',strlen(names)) == NULL )
  printf( "Didn't find an X\n" );
else
  printf( "Found an X\n" );
```

相关主题:
memcpy() and strstr().

---

memcmp

语法:

```
#include <string.h>
int memcmp( const void *buffer1, const void *buffer2, size_t count );
```

功能：函数比较 *buffer1* 和 *buffer2* 的前 *count* 个字符。返回值如下:

Value	解释
less than 0	buffer1 is less than buffer2
equal to 0	buffer1 is equal to buffer2
greater than 0	buffer1 is greater than buffer2

相关主题:
memchr(), memcpy(), and strcmp().

---

memcpy

语法:

```
#include <string.h>
void *memcpy( void *to, const void *from, size_t count );
```

功能:函数从*from*中复制*count* 个字符到*to*中,并返回*to*指针。如果*to* 和 *from* 重叠，则函数行为不确定。

相关主题:
memmove().

---

memmove

语法:

```
#include <string.h>
void *memmove( void *to, const void *from, size_t count );
```

功能: 与 mencpy 相同，不同的是当 *to* 和 *from* 重叠，函数正常仍能工作。

相关主题:
memcpy().

---

memset

语法:

```
#include <string.h>
void *memset( void *buffer, int ch, size_t count );
```

功能: 函数拷贝*ch* 到*buffer* 从头开始的*count* 个字符里，并返回*buffer*指针。memset() 可以应用在将一段内存初始化为某个值。例如:

```
memset( the_array, '\0', sizeof(the_array) );
```

这是将一个数组的所以分量设置成零的很便捷的方法。

相关主题:
memcmp(), memcpy(), and memmove().

---

strcat

语法:

```
#include <string.h>
char *strcat( char *str1, const char *str2 );
```

功能：函数将字符串 *str2* 连接到 *str1* 的末端，并返回指针 *str1*. 例如：

```
printf( "Enter your name: " );
scanf( "%s", name );
title = strcat( name, " the Great" );
printf( "Hello, %s\n", title );
```

相关主题:
strchr(), strcmp(), and strcpy().

---

strchr

语法:

```
#include <string.h>
char *strchr( const char *str, int ch );
```

功能：函数返回一个指向 *str* 中 *ch* 首次出现的位置，当没有在 *str* 中找 *ch* 到返回 NULL。

相关主题:
strpbrk(), strspn(), strstr(), and strtok().

---

strcmp

语法:

```
#include <string.h>
int strcmp( const char *str1, const char *str2 );
```

功能：比较字符串 *str1* and *str2*, 返回值如下:

返回值	解释
less than 0	str1 is less than str2
equal to 0	str1 is equal to str2
greater than 0	str1 is greater than str2

例如:

```
printf( "Enter your name: " );
scanf( "%s", name );
if( strcmp( name, "Mary" ) == 0 )
  printf( "Hello, Dr. Mary!\n" );
```

相关主题:
memcmp(), strchr(), strcpy(), and strncmp().

---

strcoll

语法:

```
#include <string.h>
int strcoll( const char *str1, const char *str2 );
```

功能：比较字符串 *str1* 和 *str2*, 很象 strcmp. 但是, strcoll() 使用在目前环境中由 setlocale() 设定的次序进行比较。

---

strcpy

语法:

```
#include <string.h>
```

```
char *strcpy( char *to, const char *from );
```

功能：复制字符串 *from* 中的字符到字符串 *to*，包括空值结束符。返回值为指针 *to*。

相关主题:
memcpy(), strchr(), strcmp(), strncmp(), and strncpy().

---

strcspn

语法:

```
#include <string.h>
size_t strcspn( const char *str1, const char *str2 );
```

功能：函数返回 *str1* 开头连续 n 个字符都不含字符串 *str2* 内字符的字符数。

相关主题:
strrchr(), strpbrk(), strstr(), and strtok().

---

strerror

语法:

```
#include <string.h>
char *strerror( int num );
```

功能：函数返回一个被定义的与某错误代码相关的错误信息。

---

strlen

语法:

```
#include <string.h>
```

```
size_t strlen( char *str );
```

功能：函数返回字符串 *str* 的长度(即空值结束符之前字符数目)。

相关主题:
memcpy(), strchr(), strcmp(), and strncmp().

---

strncat

语法:

```
#include <string.h>
char *strncat( char *str1, const char *str2, size_t count );
```

功能：将字符串 *from* 中至多 *count* 个字符连接到字符串 *to* 中，追加空值结束符。返回处理完成的字符串。

相关主题:
strcat(), strnchr(), strncmp(), and strncpy().

---

strncmp

语法:

```
#include <string.h>
int strncmp( const char *str1, const char *str2, size_t count );
```

功能：比较字符串 *str1* 和 *str2* 中至多 *count* 个字符。返回值如下：

返回值	解释
less than 0	str1 is less than str2
equal to 0	str1 is equal to str2
greater than 0	str1 is greater than str2

如果参数中任一字符串长度小于 *count*, 那么当比较到第一个空值结束符时，就结束处理。

相关主题:

strcmp(), **strnchr()**, and **strncpy()**.

---

strncpy

语法:

```
#include <string.h>
char *strncpy( char *to, const char *from, size_t count );
```

功能：将字符串 *from* 中至多 *count* 个字符复制到字符串 *to* 中。如果字符串 *from* 的长度小于 *count*，其余部分用'\0'填补。返回处理完成的字符串。

相关主题:
memcpy(), **strchr()**, **strncat()**, and **strncmp()**.

---

strpbrk

语法:

```
#include <string.h>
char *strpbrk( const char *str1, const char *str2 );
```

功能：函数返回一个指针，它指向字符串 *str2* 中任意字符在字符串 *str1* 首次出现的位置，如果不存在返回 NULL。

相关主题:
strspn(), **strrchr()**, **strstr()**, and **strtok()**.

---

strrchr

语法:

```
#include <string.h>
char *strrchr( const char *str, int ch );
```

功能：函数返回一个指针，它指向字符 *ch* 在字符串 *str* 末次出现的位置，如果匹配失败，返回 NULL。

相关主题:
strpbrk(), strspn(), strstr(), strtok(),

---

strspn

语法:

```
#include <string.h>
size_t strspn( const char *str1, const char *str2 );
```

功能：函数返回字符串 *str1* 中第一个不包含于字符串 *str2* 的字符的索引。

相关主题:
strpbrk(), strrchr(), strstr(), strtok(),

---

strstr

语法:

```
#include <string.h>
char *strstr( const char *str1, const char *str2 );
```

功能：函数返回一个指针，它指向字符串 *str2* 首次出现于字符串 *str1* 中的位置，如果没有找到，返回 **NULL。**

相关主题:
strchr(), strcspn(), strpbrk(), strspn(), strtok(), strrchr(),

---

strtod

语法:

```

```

```
#include <stdlib.h>
double strtod( const char *start, char **end );
```

功能：函数返回带符号的字符串 *start* 所表示的浮点型数。字符串 *end* 指向所表示的浮点型数之后的部分。如果溢出发生，返回 **HUGE_VAL** 或 **-HUGE_VAL**。

相关主题:
atof()

---

strtok

语法:

```
#include <string.h>
char *strtok( char *str1, const char *str2 );
```

功能：函数返回字符串 *str1* 中紧接“标记”的部分的指针，字符串 *str2* 是作为标记的分隔符。如果分隔标记没有找到，函数返回 NULL。为了将字符串转换成标记,第一次调用 *str1* 指向作为标记的分隔符。之后所以的调用 *str1* 都应为 NULL。

例如:

```
    char str[] = "now # is the time for all # good men to come to the #
aid of their country";
    char delims[] = "#";
    char *result = NULL;


    result = strtok( str, delims );


    while( result != NULL ) {
        printf( "result is \"%s\"\n", result );
        result = strtok( NULL, delims );
    }
```

以上代码的运行结果是:

```
    result is "now "
    result is " is the time for all "
    result is " good men to come to the "
    result is " aid of their country"
```

相关主题:
strchr(), strcspn(), strpbrk(), strrchr(), and strspn().

---

strtol

语法:

```
#include <stdlib.h>
long strtol( const char *start, char **end, int base );
```

功能：函数返回带符号的字符串 *start* 所表示的长整型数。参数 base 代表采用的进制方式。指针 *end* 指向 *start* 所表示的整型数之后的部分。如果返回值无法用长整型表示，函数则返回 **LONG_MAX** 或 **LONG_MIN**. 错误发生时，返回零。

相关主题:
atol().

---

strtoul

语法:

```
#include <stdlib.h>
unsigned long strtoul( const char *start, char **end, int base );
```

功能：函数基本等同 strtol(), 不同的是，它不仅可以返回长整型数，而且可以返回无符号的长整型数。

相关主题:
strtol()

---

strxfrm

语法:

```
#include <string.h>
```

```
size_t strxfrm( char *str1, const char *str2, size_t num );
```

功能:函数将字符串 *str2* 的前 *num* 个字符存储到字符串 *str1* 中。如果 strcoll() 处理字符串 *str1* 和旧的字符串 *str2*, 返回值和 strcmp()的处理结果一样。

相关主题:
strcmp(), strcoll(),

---

tolower

语法:

```
#include <ctype.h>
int tolower( int ch );
```

功能：函数字符 *ch* 的小写形式。

相关主题:
toupper(),

---

toupper

语法:

```
#include <ctype.h>
int toupper( int ch );
```

功能：函数字符 *ch* 的大写形式。

相关主题:
tolower(),

1.9 Standard C Math

abs()(见 [标题编号.])	求绝对值
acos()(见 [标题编号.])	求反余弦
asin()(见 [标题编号.])	求反正弦

atan()(见 [标题编号.])	求反正切
atan2()(见 [标题编号.])	求反正切，按符号判定象限
ceil()(见 [标题编号.])	求不小于某值的最小整数 （求上界）
cos()(见 [标题编号.])	求余弦
cosh()(见 [标题编号.])	求双曲余弦
div()(见 [标题编号.])	求商和余数
exp()(见 [标题编号.])	求 e 的幂
fabs()(见 [标题编号.])	求浮点数的绝对值
floor()(见 [标题编号.])	求不大于某值的最大整数 （求下界）
fmod()(见 [标题编号.])	求模数
frexp()(见 [标题编号.])	求数的科学表示法形式
labs()(见 [标题编号.])	求长整型数的绝对值
ldexp()(见 [标题编号.])	以科学计数法计算
ldiv()(见 [标题编号.])	以长整型返回商和余数
log()(见 [标题编号.])	自然对数
log10()(见 [标题编号.])	以 10 为底的自然对数
modf()(见 [标题编号.])	将一个数分解成整数和小数部分
pow()(见 [标题编号.])	求幂
sin()(见 [标题编号.])	求正弦
sinh()(见 [标题编号.])	求双曲正弦
sqrt()(见 [标题编号.])	求平方根
tan()(见 [标题编号.])	求正切
tanh()(见 [标题编号.])	求双曲正切

1.9.1 Standard C Math

标准 c 数学函数

abs

语法:

```
#include <stdlib.h>
int abs( int num );
```

功能： 函数返回参数 *num.* 的绝对值。例如:

```
    int magic_number = 10;
    cout << "Enter a guess: ";
    cin >> x;
    cout << "Your guess was " << abs( magic_number - x ) << " away from
the magic number." << endl;
```

相关主题:

labs().

---

acos

语法:

```
#include <math.h>
double acos( double arg );
```

功能： 函数返回参数 *arg* 的反余弦值。参数 *arg* 应当在-1 和 1 之间。

相关主题:

asin(), atan(), atan2(), sin(), cos(), tan(), sinh(), cosh(), and tanh().

---

asin

语法:

```
#include <math.h>
double asin( double arg );
```

功能： 函数返回参数 *arg* 的反正弦值。参数 *arg* 应当在-1 和 1 之间。

相关主题:

acos(), atan(), atan2(), sin(), cos(), tan(), sinh(), cosh(), and tanh().

---

atan

语法:

```
#include <math.h>
double atan( double arg );
```

功能： 函数返回参数 *arg* 的反正切值。

相关主题:

asin(), acos(), atan2(), sin(), cos(), tan(), sinh(), cosh(), and tanh().

---

atan2

语法:

```
#include <math.h>
double atan2( double y, double x );
```

功能： 函数计算 y/x 的反正切值，按照参数的符号计算所在的象限。

相关主题:

asin(), acos(), atan(), sin(), cos(), tan(), sinh(), cosh(), and tanh().

---

ceil

语法:

```
#include <math.h>
double ceil( double num );
```

功能： 函数返回参数不小于 *num* 的最小整数。例如,

```
y = 6.04;
x = ceil( y );
```

x 为 7.0.

相关主题:

floor() and fmod().

---

cos

语法:

```
#include <math.h>
double cos( double arg );
```

功能： 函数返回参数 *arg* 的余弦值， *arg* 以弧度表示给出。

相关主题:

asin(), acos(), atan(), sin(), atan2(), tan(), sinh(), cosh(), and tanh().

---

cosh

语法:

```
#include <math.h>
double cosh( double arg );
```

功能： 函数返回参数 *arg* 的双曲余弦值。

相关主题:
asin(), acos(), atan(), sin(), atan2(), tan(), sinh(), cos(), and tanh().

---

div

语法:

```
#include <stdlib.h>
div_t div( int numerator, int denominator );
```

功能： 函数返回参数 *numerator* / *denominator* 的商和余数。结构类型 **div_t** 定义在 stdlib.h 中:

```
    int quot;  // 商数
    int rem;   // 余数
```

例, 以下代码显示 x/y 的商和余数:

```
    div_t temp;
    temp = div( x, y );
    printf( "%d divided by %d yields %d with a remainder of %d\n", x, y,
temp.quot, temp.rem );
```

相关主题:
ldiv().

---

exp

语法:

```
#include <math.h>
double exp( double arg );
```

功能： 函数返回参数 returns e (2.7182818) 的 *arg* 次幂。

相关主题:
log().

---

fabs

语法:

```
#include <math.h>
double fabs( double arg );
```

功能： 函数返回参数 *arg* 的绝对值。

相关主题:

abs().

---

floor

语法:

```
#include <math.h>
double floor( double arg );
```

功能： 函数返回参数不大于 *arg* 的最大整数。例如,

```
y = 6.04;
x = floor( y );
```

x 的值为 6.0.

相关主题:

ceil().

---

fmod

语法:

```
#include <math.h>
double fmod( double x, double y );
```

功能： 函数返回参数 *x/y* 的余数。

相关主题:

ceil(), floor(), and fabs().

---

frexp

语法:

```
#include <math.h>
```

```
double frexp( double num, int *exp );
```

功能： 函数将参数 *num* 分成两部分： 0.5 和 1 之间的尾数（由函数返回）并返回指数 *exp*。转换成如下的科学计数法形式：

```
num = mantissa * (2 ^ exp)
```

相关主题:

ldexp().

---

labs

语法:

```
#include <stdlib.h>
long labs( long num );
```

功能： 函数返回参数 *num* 的绝对值。

相关主题:

abs().

---

ldexp

语法:

```
#include <math.h>
double ldexp( double num, int exp );
```

功能： 函数返回参数 *num* * (2 ^ *exp*)。如果发生溢出返回 **HUGE_VAL**。

相关主题:

frexp() and modf().

---

ldiv

语法:

```
#include <stdlib.h>
ldiv_t ldiv( long numerator, long denominator );
```

功能： 函数返回参数 *numerator* / *denominator* 的商和余数。结构类型 **ldiv_t** 定义在 stdlib.h 中：

```
long quot; // 商数
```

```
    long rem;   // 余数
```

相关主题:

div().

---

log

语法:

```
#include <math.h>
double log( double num );
```

功能： 函数返回参数 *num* 的自然对数。如果 *num* 为负,产生域错误；如果 *num* 为零，产生范围错误。

相关主题:

log10().

---

log10

语法:

```
#include <math.h>
double log10( double num );
```

功能： 函数返回参数 *num* 以 10 为底的对数。如果 *num* 为负,产生域错误；如果 *num* 为零，产生范围错误。

相关主题:

log().

---

modf

语法:

```
#include <math.h>
double modf( double num, double *i );
```

功能： 函数将参数 *num* 分割为整数和小数，返回小数部分并将整数部分赋给 *i*。

相关主题:

frexp() and ldexp().

---

pow

语法:

```
#include <math.h>
double pow( double base, double exp );
```

功能： 函数返回以参数 *base* 为底的 *exp* 次幂。如果 *base* 为零或负和 *exp* 小于等于零或非整数时,产生域错误。如果溢出，产生范围错误。

相关主题:

exp(), log(), and sqrt().

---

sin

语法:

```
#include <math.h>
double sin( double arg );
```

功能： 函数返回参数 *arg* 的正弦值，*arg* 以弧度表示给出。

相关主题:

asin(), acos(), atan(), cosh(), atan2(), tan(), sinh(), cos(), and tanh().

---

sinh

语法:

```
#include <math.h>
double sinh( double arg );
```

功能： 函数返回参数 *arg* 的双曲正弦值。

相关主题:

asin(), acos(), atan(), cosh(), atan2(), tan(), sin(), cos(), and tanh().

---

sqrt

语法:

```
#include <math.h>
double sqrt( double num );
```

功能： 函数返回参数 *num* 的平方根。如果 *num* 为负,产生域错误。

相关主题:

exp(), log(), and pow().

---

tan

语法:

```
#include <math.h>
double tan( double arg );
```

功能： 函数返回参数 *arg* 的正切值，*arg* 以弧度表示给出。

相关主题:

asin(), acos(), atan(), cosh(), atan2(), sinh(), sin(), cos(), and tanh().

---

tanh

语法:

```
#include <math.h>
double tanh( double arg );
```

功能： 函数返回参数 *arg* 的双曲正切值。

相关主题:

asin(), acos(), atan(), cosh(), atan2(), tan(), sin(), cos(), and sinh().

1.10 Standard C Time & Date

asctime()(见 [标题编号.])	时间文本格式
clock()(见 [标题编号.])	返回自程序开始运行所经过的时间
ctime()(见 [标题编号.])	返回特定格式时间
difftime()(见 [标题编号.])	两时刻的间隔
gmtime()(见 [标题编号.])	返回指向当前格林威治时间的指针
localtime()(见 [标题编号.])	返回指向当前时间的指针
mktime()(见 [标题编号.])	返回指定时间的日历格式
strftime()(见 [标题编号.])	返回日期和时间的单个元素
time()(见 [标题编号.])	返回系统的当前日历时间

1.10.1 Standard C Time & Date

标准 c 时间与日期函数

asctime

语法:

```
#include <time.h>
char *asctime( const struct tm *ptr );
```

功能： 函数将 *ptr* 所指向的时间结构转换成下列字符串：

```
day month date hours:minutes:seconds year\n\0
```

例如:

```
Mon Jun 26 12:03:53 2000
```

相关主题:

localtime(), gmtime(), time(), and ctime().

clock

语法:

```
#include <time.h>
clock_t clock( void );
```

功能： 函数返回自程序开始运行的处理器时间，如果无可用信息，返回-1。 转换返回值以秒记， 返回值除以 CLOCKS_PER_SECOND. （注: 如果编译器是 POSIX 兼容的， CLOCKS_PER_SECOND 定义为 1000000.）

相关主题:

time(), asctime(), and ctime().

ctime

语法:

```
#include <time.h>
char *ctime( const time_t *time );
```

功能： 函数转换参数 *time* 为本地时间格式：

```
day month date hours:minutes:seconds year\n\0
```

ctime() 等同

```
asctime( localtime( tp ) );
```

相关主题:

localtime(), gmtime(), time(), and asctime().

---

difftime

语法:

```
#include <time.h>
double difftime( time_t time2, time_t time1 );
```

功能： 函数返回时间参数 *time2* 和 *time1* 之差的秒数表示。

相关主题:

localtime(), gmtime(), time(), and asctime().

---

gmtime

语法:

```
#include <time.h>
struct tm *gmtime( const time_t *time );
```

功能： 函数返回给定的统一世界时间（通常是格林威治时间），如果系统不支持统一世界时间系统返回 NULL。 警告!(见 [标题编号.])

相关主题:

localtime(), time(), and asctime().

---

localtime

语法:

```
#include <time.h>
struct tm *localtime( const time_t *time );
```

功能： 函数返回本地日历时间。警告！(见 [标题编号.])

相关主题:

gmtime(), time(), and asctime().

---

mktime

语法:

```
#include <time.h>
time_t mktime( struct tm *time );
```

功能： 函数转换参数 *time* 类型的本地时间至日历时间，并返回结果。如果发生错误，返回－1。

相关主题:

time(), gmtime(), asctime(), and ctime().

---

strftime

语法:

```
#include <time.h>
size_t strftime( char *str, size_t maxsize, const char *fmt, struct tm
*time );
```

功能： 函数按照参数 *fmt* 所设定格式将 *time* 类型的参数格式化为日期时间信息，然后存储在字符串 *str* 中（至多 *maxsize* 个字符）。用于设定时间不同类型的代码为：

代码	含义
%a	星期的缩略形式
%A	星期的完整形式
%b	月份的缩略形式
%B	月份的完整形式
%c	月份的缩略形式
%d	月中的第几天(1-31)
%H	小时, 24 小时格式 (0-23)
%I	小时, 12 小时格式 (1-12)
%j	年中的第几天(1-366)
%m	月份 (1-12). **Note:** 某些版本的 Microsoft Visual C++ 可能使用取值范围 0-11.
%M	分钟(0-59)
%p	本地时间的上午或下午（AM or PM）

%S 秒钟(0-59)
%U 年中的第几周，星期天是一周的第一天
%w 星期几的数字表示(0-6, 星期天=
%W 一年中的第几周，星期天是一周的第一天
%x 标准日期字符串
%X 标准时间字符串
%y 年(0-99)
%Y 用 CCYY 表示的年（如：2004）
%Z 时区名
%% 百分号

函数 strftime()返回值为处理结果字符串 *str* 中字符的个数，如果发生错误返回零。

相关主题:
time(), localtime(), and gmtime().

---

time

语法:

```
#include <time.h>
time_t time( time_t *time );
```

功能： 函数返回当前时间，如果发生错误返回零。如果给定参数 *time* ，那么当前时间存储到参数 *time* 中。

相关主题:
localtime(), gmtime(), strftime(), ctime(),

1.11 Standard C Memory

标准 c 内存函数

calloc()(见 [标题编号.]) 分配一个二维储存空间
free()(见 [标题编号.]) 释放已分配空间
malloc()(见 [标题编号.]) 分配空间
realloc()(见 [标题编号.]) 改变已分配空间的大小

1.11.1 Standard C Memory

标准 c 内存函数

calloc

语法:

```
#include <stdlib.h>
void *calloc( size_t num, size_t size );
```

功能： 函数返回一个指向 *num* 数组空间，每一数组元素的大小为 *size*。如果错误发生返回 NULL。

相关主题:

free(), malloc(), and realloc().

free

语法:

```
#include <stdlib.h>
void free( void *ptr );
```

功能： 函数释放指针 *ptr* 指向的空间，以供以后使用。指针 *ptr* 必须由先前对 malloc(), calloc(), realloc()的调用返回。例如:

```
typedef struct data_type {
  int age;
  char name[20];
} data;

data *willy;
willy = (data*) malloc( sizeof(willy) );
...
free( willy );
```

相关主题:

calloc(), malloc(), and realloc().

malloc

语法:

```
#include <stdlib.h>
void *malloc( size_t size );
```

功能： 函数指向一个大小为 *size* 的空间，如果错误发生返回 NULL。 存储空间的指针必须为堆，不能是栈。这样以便以后用 free 函数释放空间。例如：

```
typedef struct data_type {
  int age;
  char name[20];
} data;

data *bob;
bob = (data*) malloc( sizeof(data) );
if( bob != NULL ) {
  bob->age = 22;
  strcpy( bob->name, "Robert" );
  printf( "%s is %d years old\n", bob->name, bob->age );
}
free( bob );
```

相关主题:

free(), realloc(), and calloc().

---

realloc

语法:

```
#include <stdlib.h>
void *realloc( void *ptr, size_t size );
```

功能： 函数将 *ptr* 对象的储存空间改变为给定的大小 *size*。 参数 *size* 可以是任意大小，大于或小于原尺寸都可以。 返回值是指向新空间的指针，如果错误发生返回 NULL。

相关主题:

free(), malloc(), and calloc().

1.12 Other standard C functions

abort()(见 [标题编号.])	停止程序执行
assert()(见 [标题编号.])	当表达式非真，停止程序执行
atexit()(见 [标题编号.])	当程序退出执行设定的程序
bsearch()(见 [标题编号.])	执行折半查找

exit()(见 [标题编号.])	停止程序执行
getenv()(见 [标题编号.])	获取指定环境变量的值
longjmp()(见 [标题编号.])	从设定点执行程序
qsort()(见 [标题编号.])	执行快速排序
raise()(见 [标题编号.])	向程序发送信号
rand()(见 [标题编号.])	返回一个随机数
setjmp()(见 [标题编号.])	设置程序执行点
signal()(见 [标题编号.])	将某函数设置成一个信号句柄
srand()(见 [标题编号.])	初始化随机数发生源
system()(见 [标题编号.])	执行系统调用
va_arg()(见 [标题编号.])	使用可变长度参数列表

1.12.1 Other standard C functions

其他标准 c 函数

abort

语法:

```
#include <stdlib.h>  void abort( void );
```

功能： 终止程序的执行。返回值依赖于执行，可以通过返回值显示错误。

相关主题:

exit() and **atexit()**.

---

assert

语法:

```
#include <assert.h>  void assert( int exp );
```

功能： 宏 assert()用于错误检测。如果表达式的结果为零，宏写错误信息到 STDERR 并退出程序执行。如果宏 NDEBUG 已经定义，宏 assert()将被忽略。

相关主题:

abort()

---

atexit

语法:

```
#include <stdlib.h>  int
atexit( void (*func)(void) );
```

功能： 当程序终止执行时，函数调用函数指针 func 所指向的函数。可以执行多重调用(至少 32 个)，这些函数以其注册的倒序执行。执行成功返回零值，失败则返回非零值。

相关主题:

exit() and **abort()**.

---

bsearch

语法:

```
#include <stdlib.h> void *bsearch( const void *key, const void *buf, size_t num, size_t size, int (*compare)(const void *, const void *) );
```

功能： 函数用折半查找法在从数组元素 buf[0]到 buf[num-1] 匹配参数 key。如果函数 compare 的第一个参数小于第二个参数，返回负值；如果等于返回零值；如果大于返回正值。数组 buf 中的元素应以升序排列。函数 bsearch()的返回值是指向匹配项，如果没有发现匹配项，返回 NULL。

相关主题:

qsort().

---

exit

语法:

```
#include <stdlib.h> void exit( int exit_code );
```

功能： 终止程序的执行。参数 exit_code 传递给返回值，通常零值表示正常结束，非零值表示应错误返回。

相关主题:

atexit() and **abort()**.

---

getenv

语法:

```
#include <stdlib.h> char *getenv( const char *name );
```

功能： 函数返回环境变量 *name* 的值，非常依赖执行情况。如果无对应的环境变量 *name* 返回 NULL。

相关主题:

system().

---

longjmp

语法:

```
#include <setjmp.h> void longjmp( jmp_buf envbuf, int status );
```

功能： 函数使程序从前次对 setjmp()的调用处继续执行。参数 *envbuf* 一般通过调用 setjmp() 设定。参数 *status* 为 setjmp()的返回值，用来指示不同地点 longjmp()的执行. *status* 不能设定为零。

相关主题:

setjmp().

---

qsort

语法:

```
#include <stdlib.h> void qsort( void *buf, size_t num, size_t size, int (*compare)(const void *, const void *) );
```

功能： 对 *buf* 指向的数据(包含 *num* 项,每项的大小为 *size*)进行快速排序。如果函数 *compare* 的第一个参数小于第二个参数，返回负值；如果等于返回零值；如果大于返回正值。函数对 *buf* 指向的数据按升序排序。

相关主题:

bsearch().

---

raise

语法:

```
#include <signal.h> int raise( int signal );
```

功能： 函数对程序发送指定的信号 *signal*. 一些信号:

信号	含义
SIGABRT	终止错误
SIGFPE	浮点错误
SIGILL	无效指令
SIGINT	用户输入
SIGSEGV	非法内存存取
SIGTERM	终止程序

返回零值为成功，非零为失败。

相关主题:

signal()

---

rand

语法:

```
#include <stdlib.h> int rand( void );
```

功能： 函数返回一个在零到 RAND_MAX 之间的伪随机整数。例如：

```
srand( time(NULL) );    for( i = 0; i < 10; i++ )      printf( "Random number #%d: %d\n", i, rand() );
```

相关主题:

srand()

---

setjmp

语法:

```
#include <setjmp.h>  int setjmp( jmp_buf envbuf );
```

功能： 函数将系统栈保存于 *envbuf* 中，以供以后调用 longjmp()。当第一次调用 setjmp(),它的返回值为零。之后调用 longjmp(),longjmp()的第二个参数即为 setjmp()的返回值。是否有点疑问？请参阅 longjmp().

相关主题:

longjmp()

---

signal

语法:

```
#include <signal.h>  void ( *signal( int signal, void (* func) (int)) ) (int);
```

功能： 当函数收到参数 *signal* 所表示的信号，参数 *func* 所指向的函数即被调用。*func* 可以被定制为信号句柄或以下的宏(定义在 signal.h 中):

宏	解释
SIG_DFL	默认信号处理
SIG_IGN	忽略信号

signal()返回先前为信号定义的函数地址，当错误发生返回 SIG_ERR。

---

srand

语法:

```
#include <stdlib.h>  void srand( unsigned seed );
```

功能： 设置 rand()随机序列种子。对于给定的种子 *seed*, rand()会反复产生特定的随机序列。

```
srand( time(NULL) );    for( i = 0; i < 10; i++ )      printf( "Random number #%d: %d\n", i, rand() );
```

相关主题:

rand(), **time()**(见 [标题编号.]).

---

system

语法:

```
#include <stdlib.h>  int system( const char *command );
```

功能： 函数返回给定的命令字符串 *command* 进行系统调用。如果命令执行正确通常返回零值。如果 *command* 为 NULL, system()将尝试是否有可用的命令解释器。 如果有返回非零值,否则返回零值。

相关主题:

exit(),

va_arg

语法:

```
#include <stdarg.h> type va_arg( va_list argptr, type ); void va_end( va_list argptr ); void va_start( va_list argptr, last_parm );
```

功能： 宏 va_arg()用于给函数传递可变长度的参数列表。

1. 首先，必须调用 va_start() 传递有效的参数列表 va_list 和函数强制的第一个参数。第一个参数代表将要传递的参数的个数。
2. 其次，调用 va_arg()传递参数列表 va_list 和将被返回的参数的类型。va_arg()的返回值是当前的参数。
3. 再次，对所有的参数重复调用 va_arg()
4. 最后，调用 va_end()传递 va_list 对完成后的清除是必须的。

For example:

```
int sum( int, ... );
int main( void ) {

   int answer = sum( 4, 4, 3, 2, 1 );
   printf( "The answer is %d\n", answer );

   return( 0 );
}

int sum( int num, ... ) {
   int answer = 0;
   va_list argptr;

   va_start( argptr, num );
   for( ; num > 0; num-- )
      answer += va_arg( argptr, int );

   va_end( argptr );
   return( answer );
}
```

这段代码显示 10, 他们是 4+3+2+1。

1.13 C++ I/O

<iostream>库自动定义了一些标准对象:

- **cout**, ostream 类的一个对象，可以将数据显示在标准输出设备上.
- **cerr**, ostream 类的另一个对象，它无缓冲地向标准错误输出设备输出数据.
- **clog**, 类似 **cerr**,但是它使用缓冲输出.
- **cin**, istream 类的一个对象,它用于从标准输入设备读取数据.

<fstream>库允许编程人员利用 **ifstream** 和 **ofstream** 类进行文件输入和输出.

一些 C++ I/O 流(精度,判断等)的行为可以通过操作不同的标志(见 [标题编号.])来修改。

Constructors(见 [标题编号.])	构造器
bad()(见 [标题编号.])	如果出现错误则返回 true
clear()(见 [标题编号.])	清除状态标志
close()(见 [标题编号.])	关闭一个流
eof()(见 [标题编号.])	如果处于文件结尾处则返回 true
fail()(见 [标题编号.])	如果出现错误则返回 true
fill()(见 [标题编号.])	控制默认填充字符
flags()(见 [标题编号.])	操作 flags
flush()(见 [标题编号.])	清空缓冲区
gcount()(见 [标题编号.])	返回读取的最后一次输入的字符数
get()(见 [标题编号.])	读取字符
getline()(见 [标题编号.])	读取一行字符
good()(见 [标题编号.])	如果没有出现过错误则返回 true
ignore()(见 [标题编号.])	读取字符并忽略指定字符
open()(见 [标题编号.])	创建一个输入流
peek()(见 [标题编号.])	检查下一个输入的字符
precision()(见 [标题编号.])	设置精度
put()(见 [标题编号.])	写字符
putback()(见 [标题编号.])	返回字符给一个流
rdstate()(见 [标题编号.])	返回流的状态
read()(见 [标题编号.])	读取字条符
seekg()(见 [标题编号.])	在一个输入流中进行随机访问
seekp()(见 [标题编号.])	在一个输出流中进行随机访问
setf()(见 [标题编号.])	设置格式标志
sync_with_stdio()(见 [标题编号.])	同标准 I/O 同步
tellg()(见 [标题编号.])	使用输入流读取流指针
tellp()(见 [标题编号.])	使用输出流读取流指针
unsetf()(见 [标题编号.])	清除格式标志
width()(见 [标题编号.])	操作域宽度

write()(见 [标题编号.])　　写字符

1.13.1 C++ I/O

构造器

语法:

```
fstream( const char *filename, openmode mode );
ifstream( const char *filename, openmode mode );
ofstream( const char *filename, openmode mode );
```

Tstream, ifstream, 和 ofstream 对象用于文件输入/输出. 可选择*模式*通过使用 ios stream mode flags 定义了一个文件如何打开。*filename* 指定被打开的文件并与流相关联。例如，下面的代码读取输入的数据并追加结果到一个输出文件中.

```
ifstream fin( "/tmp/data.txt" );
ofstream fout( "/tmp/results.txt", ios::app );
while( fin >> temp )
  fout << temp + 2 << endl;
fin.close();
fout.close();
```

输入和输出文件流可以相似的方式被使用在 C++预定义 I/O 流，**cin** 和 **cout**.

相关主题:

close(), open()

---

bad

语法:

```
bool bad();
```

如果当前的流发生致命的错误，bad()函数返回 **true**，否则返回 **false**。

相关主题:

good()

---

clear

语法:

```
void clear( iostate flags = goodbit );
```

函数 clear()清除与当前流相关联的标志(见 [标题编号.])。默认标志是 goodbit 它清除所有标志.否则只有指定的标志被清除。

相关主题:

rdstate()

---

close

语法:

```
void close();
```

Tclose()函数关闭相关的文件流。

相关主题:

open()

---

eof

语法:

```
bool eof();
```

如果到达相关联的输入文件的末尾，eof()函数返回 **true**，否则返回 **false**。例如：

```
char ch;
ifstream fin( "temp.txt" );
while( !fin.eof() ) {
  fin >> ch;
  cout << ch;
}
fin.close();
```

相关主题:

bad(), **fail()**, **good()**, **rdstate()**, **clear()**

---

fail

语法:

```
bool fail();
```

如果当前流发生错误 fail()函数返回 **true** ， 否则返回 **false** 。

相关主题:

good(), eof(), bad(), clear(), rdstate()

---

fill

语法:

```
char fill();
char fill( char ch );
```

函数 fill()可以返回当前填充字符，或者设置当前填充字符为 *ch* 。填充字符被定义为用来填充字符，当一个数字比较指定宽度 T 小时。默认的填充字符是空格。

相关主题:

precision(), width()

---

flags

语法:

```
fmtflags flags();
fmtflags flags( fmtflags f );
```

flags()函数或者返回当前流的格式标志(见 [标题编号.])，或者为当前流设置标志为 *f*。

相关主题:

unsetf(), setf()

---

flush

语法:

```
```

```
ostream &flush();
```

flush()函数可以引起当把前流的缓冲区写出到附属设备中去。这个函数对于打印调试信息很用处，因为当程序有机会把缓冲区内容写出到屏幕之前，程序会被中断。灵活地使用 flush()可以保证你所有的调试状态都实在的打印出来。

相关主题:
put(), write()

---

gcount

语法:

```
streamsize gcount();
```

函数 gcount()被用于输入流，并返回上一次输入操作被读入的字符的数目。

相关主题:
get(), getline(), read()

---

get

语法:

```
int get();
istream &get( char &ch );
istream &get( char *buffer, streamsize num );
istream &get( char *buffer, streamsize num, char delim );
istream &get( streambuf &buffer );
istream &get( streambuf &buffer, char delim );
```

get()函数被用于输入流，和以下这些:

- 读入一个字符并返回它的值，
- 读入一个字符并把它存储在 *ch*，
- 读取字符到 *buffer* 直到 *num* - 1 个字符被读入, 或者碰到 EOF 或换行标志，
- 读取字符到 *buffer* 直到已读入 *num* - 1 个字符，或者碰到 EOF 或 *delim*(*delim* 直到下一次不会被读取)，
- 读取字符到 *buffer* 中，直到碰到换行或 EOF，
- 或是读取字符到 *buffer* 中，直到碰到换行，EOF 或 *delim*。(相反, *delim* 直到下一个 get() 不会被读取).

例如，下面的代码一个字符一个字符的显示文件 temp.txt 中的内容：

```
char ch;
ifstream fin( "temp.txt" );
while( fin.get(ch) )
  cout << ch;
fin.close();
```

相关主题:

put(), read(), getline()

---

getline

语法:

```
istream &getline( char *buffer, streamsize num );
istream &getline( char *buffer, streamsize num, char delim );
```

getline()函数用于输入流，读取字符到 *buffer* 中，直到下列情况发生：

- *num* - 1 个字符已经读入,
- 碰到一个换行标志，
- 碰到一个 EOF，
- 或者，任意地读入，直到读到字符 *delim*。*delim* 字符不会被放入 *buffer* 中。

相关主题:

get(), read()

---

good

语法:

```
bool good();
```

如果当前流没有发生错误，函数 good()返回 **true** ，否则返回 **false**。

相关主题:

bad(), fail(), eof(), clear(), rdstate()

---

ignore

语法:

```
istream &ignore( streamsize num=1, int delim=EOF );
```

ignore()函数用于输入流。它读入字符，直到已经读了 *num* 个字符(默认为 1)或是直到字符 *delim* 被读入(默认为 EOF).

相关主题:
get(), getline()

---

open

语法:

```
void open( const char *filename );
void open( const char *filename, openmode mode );
```

函数 open()用于文件流。它打开 *filename* 并将其与当前的流相关联。可以选择的模式有：

模式	含义
ios::app	添加输出
ios::ate	当已打开时寻找到 EOF
ios::binary	以二进制模式打开文件
ios::in	为读取打开文件
ios::out	为写入打开文件
ios::trunc	覆盖存在的文件

如果 open()失败，当用于一个布尔表达式中时，作为结果的流会给出对错误的评估。例如：

```
ifstream inputStream("file.txt");
if( !inputStream ) {
  cerr << "Error opening input stream" << endl;
  return;
}
```

相关主题:
close(), fstream(), ifstream(), ofstream(),

---

peek

语法:

```
int peek();
```

函数 peek()用于输入流中，并返回在流中的下一个字符或如果是处于被入的文件的结尾处返回 EOF。peek()不会把字符从流中移除。

相关主题:
get(), putback()

---

precision

语法:

```
streamsize precision();
streamsize precision( streamsize p );
```

precision()函数设置或返回当前要被显示的浮点变量的位数。例如，下面的代码：

```
float num = 314.15926535;
cout.precision( 5 );
cout << num;
```

displays

```
314.16
```

相关主题:
width(), fill()

---

put

语法:

```
ostream &put( char ch );
```

函数 put()用于输出流，并把字符 *ch* 写入流中。

相关主题:
write(), get()

---

putback

语法:

```
istream &putback( char ch );
```

putback()函数用于输入流，并且返回以前读的字符 *ch* 到输入流中。

相关主题:

peek()

---

rdstate

语法:

```
iostate rdstate();
```

rdstate()函数返回当前流的状态。**iostate** 对象有下面这些标志：

标志	含义
badbit	发生致命的错误
eofbit	已经发现 EOF
failbit	一个非致命性错误已经发生
goodbit	没有发生错误

相关主题:

eof(), **good()**, **bad()**, **clear()**, **fail()**

---

read

语法:

```
istream &read( char *buffer, streamsize num );
```

函数 read()用于输入流，在将字符放入 *buffer* 之前从流中读取 *num* 个字节。如果碰到 EOF，read()中止，丢弃不论多少个字节已经放入。例如：

```
    struct {
```

```
    int height;
    int width;
  } rectangle;

  input_file.read( (char *)(&rectangle), sizeof(rectangle) );
  if( input_file.bad() ) {
    cerr << "Error reading data" << endl;
    exit( 0 );
  }
```

相关主题:

gcount(), get(), getline(), write()

---

seekg

语法:

```
istream &seekg( off_type offset, ios::seekdir origin );
istream &seekg( pos_type position );
```

函数 seekg()用于输入流,并且它将重新设置"get"指针到当前流的从 *origin* 偏移 *offset* 个字节的位置上，或是置"get"指针在 *position* 位置。

相关主题:

seekp(), tellg(), tellp()

---

seekp

语法:

```
ostream &seekp( off_type offset, ios::seekdir origin );
ostream &seekp( pos_type position );
```

seekp()函数用于输出流，但在其它方面和 seekg()很类似。

相关主题:

seekg(), tellg(), tellp()

---

setf

语法:

```
```

```
fmtflags setf( fmtflags flags );
fmtflags setf( fmtflags flags, fmtflags needed );
```

函数 setf()设置当前流的格式化标志(见 [标题编号.])为 *flags*。可选标志 *needed* 只允许 *flags* 标志和 *needed* 标志都被设置。返回值是前面设置的标志。例如：

```
int number = 0x3FF;
cout.setf( ios::dec );
cout << "Decimal: " << number << endl;
cout.unsetf( ios::dec );
cout.setf( ios::hex );
cout << "Hexadecimal: " << number << endl;
```

提示，上面的代码和下面的代码的功能是一致的:

```
int number = 0x3FF;
cout <<"Decimal: " <<number <<endl <<hex <<"Hexadecimal: " <<number
<< dec << endl;
```

参考 manipulators(见 [标题编号.]).

相关主题:

flags(), unsetf()

---

sync_with_stdio

语法:

```
static bool sync_with_stdio( bool sync=true );
```

sync_with_stdio()函数有打开或关闭使用 C++风格 I/O 系统混合 C 风格的 I/O 系统的功能。

---

tellg

语法:

```
pos_type tellg();
```

tellg()函数用于输入流，并返回流中"get"指针的当前位置。

相关主题:
seekg(), seekp(), tellp()

---

tellp

语法:

```
pos_type tellp();
```

tellp()函数用于输出流中，并返回在流中当前"put"指针的位置。 例如，下面的代码显示了当一个文件指针写入一个流的时候的情形：

```
string s("In Xanadu did Kubla Khan...");

ofstream fout("output.txt");

for( int i=0; i < s.length(); i++ ) {
  cout << "File pointer: " << fout.tellp();
  fout.put( s[i] );
  cout << " " << s[i] << endl;
}

fout.close();
```

相关主题:
seekg(), seekp(), tellg()

---

unsetf

语法:

```
void unsetf( fmtflags flags );
```

函数 unsetf()用于清除与当前流相关的给定的标志 *flags*。什么标志呢?(见 [标题编号.])

相关主题:
setf(), flags()

---

width

语法:

```
int width();
int width( int w );
```

函数 width()返回当前的宽度。可选择参数 *w* 用于设定宽度大小。宽度是指每一次输出中显示的字符的最小数目。例如：

```
cout.width( 5 );
cout << "2";
```

displays

```
    2
```

(在一个'2'的后面紧跟着四个空格)

相关主题:
precision(), fill()

---

write

语法:

```
ostream &write( const char *buffer, streamsize num );
```

write()函数用于输出流，从 *buffer* 中写 *num* 个字节到当前输出流中。

相关主题:
read(), put()

1.14 C++ String

Constructors(见 [标题编号.])	构造函数，用于字符串初始化
Operators(见 [标题编号.])	操作符，用于字符串比较和赋值
append()(见 [标题编号.])	在字符串的末尾添加文本
assign()(见 [标题编号.])	为字符串赋新值
at()(见 [标题编号.])	按给定索引值返回字符
begin()(见 [标题编号.])	返回一个迭代器，指向第一个字符

c_str()(见 [标题编号.])	将字符串以 C 字符数组的形式返回
capacity()(见 [标题编号.])	返回重新分配空间前的字符容量
compare()(见 [标题编号.])	比较两个字符串
copy()(见 [标题编号.])	将内容复制为一个字符数组
data()(见 [标题编号.])	返回内容的字符数组形式
empty()(见 [标题编号.])	如果字符串为空，返回真
end()(见 [标题编号.])	返回一个迭代器，指向字符串的末尾。（最后一个字符的下一个位置）
erase()(见 [标题编号.])	删除字符
find()(见 [标题编号.])	在字符串中查找字符
find_first_of()(见 [标题编号.])	查找第一个与 value 中的某值相等的字符
find_first_not_of()(见 [标题编号.])	查找第一个与 value 中的所有值都不相等的字符
find_last_of()(见 [标题编号.])	查找最后一个与 value 中的某值相等的字符
find_last_not_of()(见 [标题编号.])	查找最后一个与 value 中的所有值都不相等的字符
get_allocator()(见 [标题编号.])	返回配置器
insert()(见 [标题编号.])	插入字符
length()(见 [标题编号.])	返回字符串的长度
max_size()(见 [标题编号.])	返回字符的最大可能个数
rbegin()(见 [标题编号.])	返回一个逆向迭代器，指向最后一个字符
rend()(见 [标题编号.])	返回一个逆向迭代器，指向第一个元素的前一个位置
replace()(见 [标题编号.])	替换字符
reserve()(见 [标题编号.])	保留一定容量以容纳字符串（设置 capacity 值）
resize()(见 [标题编号.])	重新设置字符串的大小
rfind()(见 [标题编号.])	查找最后一个与 value 相等的字符（逆向查找）
size()(见 [标题编号.])	返回字符串中字符的数量
substr()(见 [标题编号.])	返回某个子字符串
swap()(见 [标题编号.])	交换两个字符串的内容

1.14.1 C++ String

构造函数(Constructors)

语法:

```
  string();
  string( size_type length, char ch );
  string( const char *str );
  string( const char *str, size_type length );
  string( string &str, size_type index, size_type length );
  string( input_iterator(见 [标题编号.]) start, input_iterator(见 [标题编号.]) end );
```

字符串的构造函数创建一个新字符串，包括:

- 以 length 为长度的 ch 的拷贝（即 length 个 ch）
- 以 str 为初值 (长度任意),
- 以 index 为索引开始的子串，长度为 length, 或者
- 以从 start 到 end 的元素为初值.

例如,

```
    string str1( 5, 'c' );
    string str2( "Now is the time..." );
    string str3( str2, 11, 4 );
    cout << str1 << endl;
    cout << str2 << endl;
    cout << str3 << endl;
```

显示

```
    ccccc
    Now is the time...
    time
```

操作符(Operators)

语法:

```
  ==
  >
  =
  <=
  !=
  +
```

```
  +=
  []
```

你可以用 ==, >, =, <=, and !=比较字符串. 可以用 + 或者 += 操作符连接两个字符串, 并且可以用[]获取特定的字符.

相关主题:
at(), compare().

添加文本(append)

语法:

```
  basic_string &append( const basic_string &str );
  basic_string &append( const char *str );
  basic_string &append( const basic_string &str, size_type index,
size_type len );
  basic_string &append( const char *str, size_type num );
  basic_string &append( size_type num, char ch );
  basic_string &append( input_iterator(见 [标题编号.]) start,
input_iterator(见 [标题编号.]) end );
```

append() 函数可以完成以下工作:

- 在字符串的末尾添加 str,
- 在字符串的末尾添加 str 的子串,子串以 index 索引开始，长度为 len
- 在字符串的末尾添加 str 中的 num 个字符,
- 在字符串的末尾添加 num 个字符 ch,
- 在字符串的末尾添加以迭代器 start 和 end 表示的字符序列.

例如以下代码:

```
    string str = "Hello World";
    str.append( 10, '!' );
    cout << str << endl;
```

显示

```
    Hello World!!!!!!!!!!
```

相关主题:
+ 操作符

赋值(assign)

语法:

```
  basic_string &assign( const basic_string &str );
  basic_string &assign( const char *str );
  basic_string &assign( const char *str, size_type num );
  basic_string &assign( const basic_string &str, size_type index,
size_type len );
  basic_string &assign( size_type num, char ch );
```

函数以下列方式赋值:

- 用 str 为字符串赋值,
- 用 str 的开始 num 个字符为字符串赋值,
- 用 str 的子串为字符串赋值,子串以 index 索引开始，长度为 len
- 用 num 个字符 ch 为字符串赋值.

例如以下代码:

```
    string str1, str2 = "War and Peace";
    str1.assign( str2, 4, 3 );
    cout << str1 << endl;
```

显示

```
    and
```

---

at

语法:

```
  reference at( size_type index );
```

at()函数返回一个引用，指向在 index 位置的字符. 如果 index 不在字符串范围内, at() 将报告"out of range"错误，并抛出 **out_of_range** 异常。 比如下列代码:

```
    string text = "ABCDEF";
    char ch = text.at( 2 );
```

显示字符 'C'.

相关主题:
[]操作符

---

begin

语法:

```
iterator(见 [标题编号.]) begin();
```

begin()函数返回一个迭代器(见 [标题编号.]),指向字符串的第一个元素.

相关主题:
end()

---

c_str

语法:

```
const char *c_str();
```

c_str()函数返回一个指向正规 C 字符串的指针, 内容与本字符串相同.

相关主题:
[] 操作符

容量(capacity)

语法:

```
size_type capacity();
```

capacity()函数返回在重新申请更多的空间前字符串可以容纳的字符数. 这个数字至少与 size()一样大.

相关主题:
max_size(), reserve(), resize(), size(),

比较(compare)

语法:

```
  int compare( const basic_string &str );
  int compare( const char *str );
  int compare( size_type index, size_type length, const basic_string
&str );
  int compare( size_type index, size_type length, const basic_string &str,
size_type index2,
  size_type length2 );
  int compare( size_type index, size_type length, const char *str,
size_type length2 );
```

compare()函数以多种方式比较本字符串和 str,返回:

返回值	情况
小于零	this < str
零	this == str
大于零	this > str

不同的函数:

- 比较自己和 str,
- 比较自己的子串和 str,子串以 index 索引开始，长度为 length
- 比较自己的子串和 str 的子串，其中 index2 和 length2 引用 str，index 和 length 引用自己
- 比较自己的子串和 str 的子串，其中 str 的子串以索引 0 开始，长度为 length2，自己的子串以 index 开始，长度为 length

相关主题:
操作符

拷贝(copy)

语法:

```
  size_type copy( char *str, size_type num, size_type index );
```

copy()函数拷贝自己的 num 个字符到 str 中（从索引 index 开始）。返回值是拷贝的字符数

---

data

语法:

```
const char *data();
```

data()函数返回指向自己的第一个字符的指针.

相关主题:

c_str()

---

empty

语法:

```
bool empty();
```

如果字符串为空则 empty()返回真(true)，否则返回假(false).

---

end

语法:

```
iterator(见 [标题编号.]) end();
```

end()函数返回一个迭代器(见 [标题编号.])，指向字符串的末尾(最后一个字符的下一个位置).

相关主题:

begin()

删除(erase)

语法:

```
iterator(见 [标题编号.]) erase( iterator(见 [标题编号.]) pos );
iterator(见 [标题编号.]) erase( iterator(见 [标题编号.]) start,
iterator(见 [标题编号.]) end );
basic_string &erase( size_type index = 0, size_type num = npos );
```

erase()函数可以:

- 删除 pos 指向的字符, 返回指向下一个字符的迭代器(见 [标题编号.]),
- 删除从 start 到 end 的所有字符, 返回一个迭代器(见 [标题编号.]),指向被删除的最后一个字符的下一个位置
- 删除从 index 索引开始的 num 个字符, 返回***this**.

参数 *index* 和 *num* 有默认值, 这意味着 erase()可以这样调用:只带有 *index* 以删除 index 后的所有字符，或者不带有任何参数以删除所有字符. 例如:

```
    string s("So, you like donuts, eh? Well, have all the donuts in the
world!");
    cout << "The original string is '" << s << "'" << endl;

    s.erase( 50, 14 );
    cout << "Now the string is '" << s << "'" << endl;

    s.erase( 24 );
    cout << "Now the string is '" << s << "'" << endl;

    s.erase();
    cout << "Now the string is '" << s << "'" << endl;
```

将显示

```
    The original string is 'So, you like donuts, eh? Well, have all the
donuts in the world!'
    Now the string is 'So, you like donuts, eh? Well, have all the donuts'
    Now the string is 'So, you like donuts, eh?'
    Now the string is ''
```

查找(find)

语法:

```
  size_type find( const basic_string &str, size_type index );
  size_type find( const char *str, size_type index );
  size_type find( const char *str, size_type index, size_type length );
  size_type find( char ch, size_type index );
```

find()函数:

- 返回 str 在字符串中第一次出现的位置（从 index 开始查找）。如果没找到则返回 **string::npos**,
- 返回 str 在字符串中第一次出现的位置（从 index 开始查找，长度为 length）。如果没找到就返回 **string::npos**,
- 返回字符 ch 在字符串中第一次出现的位置（从 index 开始查找）。如果没找到就返回 **string::npos**

例如,

```
string str1( "Alpha Beta Gamma Delta" );
unsigned int loc = str1.find( "Omega", 0 );
if( loc != string::npos )
  cout << "Found Omega at " << loc << endl;
else
  cout << "Didn't find Omega" << endl;
```

---

find_first_of

语法:

```
  size_type find_first_of( const basic_string &str, size_type index = 0 );
  size_type find_first_of( const char *str, size_type index = 0 );
  size_type find_first_of( const char *str, size_type index, size_type
num );
  size_type find_first_of( char ch, size_type index = 0 );
```

find_first_of()函数:

- 查找在字符串中第一个与 str 中的某个字符匹配的字符，返回它的位置。搜索从 index 开始，如果没找到就返回 **string::npos**
- 查找在字符串中第一个与 str 中的某个字符匹配的字符，返回它的位置。搜索从 index 开始，最多搜索 num 个字符。如果没找到就返回 **string::npos,**
- 查找在字符串中第一个与 ch 匹配的字符，返回它的位置。搜索从 index 开始。

相关主题:

find()

---

find_first_not_of

语法:

```
  size_type find_first_not_of( const basic_string &str, size_type index
= 0 );
  size_type find_first_not_of( const char *str, size_type index = 0 );
  size_type find_first_not_of( const char *str, size_type index, size_type
num );
  size_type find_first_not_of( char ch, size_type index = 0 );
```

find_first_not_of()函数:

- 在字符串中查找第一个与 str 中的字符都不匹配的字符，返回它的位置。搜索从 index 开始。如果没找到就返回 **string::nops**
- 在字符串中查找第一个与 str 中的字符都不匹配的字符，返回它的位置。搜索从 index 开始，最多查找 num 个字符。如果没找到就返回 **string::nops**
- 在字符串中查找第一个与 ch 不匹配的字符，返回它的位置。搜索从 index 开始。如果没找到就返回 **string::nops**

相关主题:
find()

---

find_last_of

语法:

```
  size_type find_last_of( const basic_string &str, size_type index =
npos );
  size_type find_last_of( const char *str, size_type index = npos );
  size_type find_last_of( const char *str, size_type index, size_type
num );
  size_type find_last_of( char ch, size_type index = npos );
```

find_last_of()函数:

- 在字符串中查找最后一个与 str 中的某个字符匹配的字符,返回它的位置。搜索从 index 开始。如果没找到就返回 **string::nops**
- 在字符串中查找最后一个与 str 中的某个字符匹配的字符,返回它的位置。搜索从 index 开始，最多搜索 num 个字符。如果没找到就返回 **string::nops**
- 在字符串中查找最后一个与 ch 匹配的字符，返回它的位置。搜索从 index 开始。如果没找到就返回 **string::nops**

相关主题:
find()

---

find_last_not_of

语法:

```
  size_type find_last_not_of( const basic_string &str, size_type index =
npos );
  size_type find_last_not_of( const char *str, size_type index = npos);
  size_type find_last_not_of( const char *str, size_type index, size_type
num );
  size_type find_last_not_of( char ch, size_type index = npos );
```

find_last_not_of()函数:

- 在字符串中查找最后一个与 str 中的字符都不匹配的字符,返回它的位置。搜索从 index 开始。如果没找到就返回 **string::nops**
- 在字符串中查找最后一个与 str 中的字符都不匹配的字符,返回它的位置。搜索从 index 开始，最多查找 num 个字符如果没找到就返回 **string::nops**
- 在字符串中查找最后一个与 ch 不匹配的字符， 返回它的位置。搜索从 index 开始。如果没找到就返回 **string::nops**

相关主题:

find()

---

get_allocator

语法:

```
  allocator_type get_allocator();
```

get_allocator()函数返回本字符串的配置器.

插入(insert)

语法:

```
  iterator(见 [标题编号.]) insert( iterator(见 [标题编号.]) i, const char
&ch );
  basic_string &insert( size_type index, const basic_string &str );
  basic_string &insert( size_type index, const char *str );
  basic_string &insert( size_type index1, const basic_string &str,
```

```
size_type index2, size_type num );
  basic_string &insert( size_type index, const char *str, size_type num );
  basic_string &insert( size_type index, size_type num, char ch );
  void insert( iterator(见 [标题编号.]) i, size_type num, const char
&ch );
  void insert( iterator(见 [标题编号.]) i, iterator(见 [标题编号.])
start, iterator(见 [标题编号.]) end );
```

insert()函数的功能非常多:

- 在迭代器 i 表示的位置前面插入一个字符 ch,
- 在字符串的位置 index 插入字符串 str,
- 在字符串的位置 index 插入字符串 str 的子串(从 *index2* 开始，长 num 个字符),
- 在字符串的位置 index 插入字符串 str 的 num 个字符,
- 在字符串的位置 index 插入 num 个字符 ch 的拷贝,
- 在迭代器 i 表示的位置前面插入 num 个字符 ch 的拷贝,
- 在迭代器 i 表示的位置前面插入一段字符，从 start 开始，以 end 结束.

相关主题:

replace()

长度(length)

语法:

```
  size_type length();
```

length()函数返回字符串的长度. 这个数字应该和 size()返回的数字相同.

相关主题:

size()

---

max_size

语法:

```
  size_type max_size();
```

max_size()函数返回字符串能保存的最大字符数。

---

rbegin

语法:

```
const reverse_iterator(见 [标题编号.]) rbegin();
```

rbegin()返回一个逆向迭代器(见 [标题编号.])，指向字符串的最后一个字符。

相关主题:

rend()

---

rend

语法:

```
const reverse_iterator(见 [标题编号.]) rend();
```

rend()函数返回一个逆向迭代器(见 [标题编号.])，指向字符串的开头（第一个字符的前一个位置）。

相关主题:

rbegin()

替换(replace)

语法:

```
basic_string &replace( size_type index, size_type num, const
basic_string &str );
  basic_string &replace( size_type index1, size_type num1, const
basic_string &str, size_type index2,
  size_type num2 );
  basic_string &replace( size_type index, size_type num, const char
*str );
  basic_string &replace( size_type index, size_type num1, const char *str,
size_type num2 );
  basic_string &replace( size_type index, size_type num1, size_type num2,
char ch );
  basic_string &replace( iterator(见 [标题编号.]) start, iterator(见 [标
题编号.]) end, const basic_string &str );
  basic_string &replace( iterator(见 [标题编号.]) start, iterator(见 [标
```

```
题编号.]) end, const char *str );
  basic_string &replace( iterator(见 [标题编号.]) start, iterator(见 [标题编号.]) end, const char *str, size_type num );
  basic_string &replace( iterator(见 [标题编号.]) start, iterator(见 [标题编号.]) end, size_type num, char ch );
```

replace()函数:

- 用 str 中的 num 个字符替换本字符串中的字符,从 index 开始
- 用 str 中的 num2 个字符（从 index2 开始）替换本字符串中的字符，从 index1 开始，最多 num1 个字符
- 用 str 中的 num 个字符（从 index 开始）替换本字符串中的字符
- 用 str 中的 num2 个字符（从 index2 开始）替换本字符串中的字符，从 index1 开始，num1 个字符
- 用 num2 个 ch 字符替换本字符串中的字符，从 index 开始
- 用 str 中的字符替换本字符串中的字符,迭代器 start 和 end 指示范围
- 用 str 中的 num 个字符替换本字符串中的内容,迭代器 start 和 end 指示范围，
- 用 num 个 ch 字符替换本字符串中的内容，迭代器 start 和 end 指示范围.

例如，以下代码显示字符串"They say he carved it himself...find your soul-mate, Homer."

```
string s = "They say he carved it himself...from a BIGGER spoon";
string s2 = "find your soul-mate, Homer.";

s.replace( 32, s2.length(), s2 );

cout << s << endl;
```

相关主题:
insert()

保留空间(reserve)

语法:

```
void reserve( size_type num );
```

reserve()函数设置本字符串的 capacity 以保留 num 个字符空间。

相关主题:
capacity()

---

resize

语法:

```
void resize( size_type num );
void resize( size_type num, char ch );
```

resize()函数改变本字符串的大小到 *num*, 新空间的内容不确定。也可以指定用 ch 填充。

---

rfind

语法:

```
size_type rfind( const basic_string &str, size_type index );
size_type rfind( const char *str, size_type index );
size_type rfind( const char *str, size_type index, size_type num );
size_type rfind( char ch, size_type index );
```

rfind()函数:

- 返回最后一个与 str 中的某个字符匹配的字符，从 index 开始查找。如果没找到就返回 **string::npos**
- 返回最后一个与 str 中的某个字符匹配的字符，从 index 开始查找,最多查找 num 个字符。如果没找到就返回 **string::npos**
- 返回最后一个与 ch 匹配的字符，从 index 开始查找。如果没找到就返回 **string::npos**

例如,在下列代码中第一次调用 rfind()返回 **string::npos**,因为目标词语不在开始的 8 个字符中。然而，第二次调用返回 9，因为目标词语在开始的 20 个字符之中。

```
int loc;
string s = "My cat's breath smells like cat food.";

loc = s.rfind( "breath", 8 );
cout << "The word breath is at index " << loc << endl;

loc = s.rfind( "breath", 20 );
cout << "The word breath is at index " << loc << endl;
```

相关主题:

find()

---

size

语法:

```
size_type size();
```

size()函数返回字符串中现在拥有的字符数。

相关主题:
length(), max_size()

---

substr

语法:

```
basic_string substr( size_type index, size_type num = npos );
```

substr()返回本字符串的一个子串，从 index 开始，长 num 个字符。如果没有指定，将是默认值 **string::npos**。这样，substr()函数将简单的返回从 index 开始的剩余的字符串。

例如:

```
string s("What we have here is a failure to communicate");

string sub = s.substr(21);

cout << "The original string is " << s << endl;
cout << "The substring is " << sub << endl;
```

显示：

```
The original string is What we have here is a failure to communicate
The substring is a failure to communicate
```

交换(swap)

语法:

```
void swap( basic_string &str );
```

swap()函数把 str 和本字符串交换。例如：

```
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

显示：

```
And this is second
This comes first
```

1.15 C++ 标准模板库

C++ STL (Standard Template Library 标准模板库) 是通用类模板和算法的集合，它提供给程序员一些标准的数据结构的实现如 queues(见 [标题编号.])(队列)，lists(见 [标题编号.])(链表)，和 stacks(见 [标题编号.])(栈)等.

C++ STL 提供给程序员以下三类数据结构的实现:

- 顺序结构
  - C++ Vectors(见 [标题编号.])
  - C++ Lists
  - C++ Double-Ended Queues(见 [标题编号.])
- 容器适配器
  - C++ Stacks(见 [标题编号.])
  - C++ Queues(见 [标题编号.])
  - C++ Priority Queues(见 [标题编号.])
- 联合容器
  - C++ Bitsets(见 [标题编号.])
  - C++ Maps(见 [标题编号.])
  - C++ Multimaps(见 [标题编号.])
  - C++ Sets(见 [标题编号.])
  - C++ Multisets(见 [标题编号.])

程序员使用复杂数据结构的最困难的部分已经由 STL 完成. 如果程序员想使用包含 int 数据的 stack, 他只要写出如下的代码:

```
stack<int> myStack;
```

接下来，他只要简单的调用 push()(见 [标题编号.]) 和 pop()(见 [标题编号.]) 函数来操作栈. 借助 C++ 模板的威力，他可以指定任何的数据类型，不仅仅是 int 类型. STL stack 实现了栈的功能，而不管容纳的是什么数据类型.

1.15.1 C++ Bitset

Constructors(见 [标题编号.])	创建新 bitsets
Operators(见 [标题编号.])	比较和赋值 bitsets
any()(见 [标题编号.])	如果有任何一个位被设置就返回 true
count()(见 [标题编号.])	返回被设置的位的个数
flip()(见 [标题编号.])	反转 bits 中的位
none()(见 [标题编号.])	如果没有位被设置则返回 true
reset()(见 [标题编号.])	清空所有位
set()(见 [标题编号.])	设置位
size()(见 [标题编号.])	返回可以容纳的位的个数
test()(见 [标题编号.])	返回指定位的状态
to_string()(见 [标题编号.])	返回 bitset 的字符串表示
to_ulong()(见 [标题编号.])	返回 bitset 的整数表示

1.15.1.1 C++ Bitset

Constructors

语法:

```
bitset();
bitset( unsigned long val );
```

C++ Bitsets 能以无参的形式创建，或者提供一个长无符号整数，它将被转化为二进制，然后插入到 bitset 中。当创建 bitset 时，模板中提供的数字决定 bitset 有多长。

例如，以下代码创建两个 bitsets，然后显示它们：

```
// 创建一个 8 位长的 bitset
bitset bs;
```

```
// 显示这个 bitset
for( int i = (int) bs.size(); i >= 0; i-- ) {
  cout << bs[i] << " ";
}
cout << endl;

// 创建另一个 bitset
bitset bs2( (long) 131 );

// 显示
for( int i = (int) bs2.size(); i >= 0; i-- ) {
  cout << bs2[i] << " ";
}
cout << endl;
```

Operators

语法:

```
!=, ==, &=, ^=, |=, ~, <<=, >>=, []
```

这些操作符都可以和 bitsets 一起工作。它们被这样定义:

- != 返回真如果两个 bitset 不相等。
- == 返回真如果两个 bitset 相等。
- &= 完成两个 bitset 间的与运算。
- ^= 完成两个 bitset 间的异或运算。
- |= 完成两个
- ~ 反置 bitset (和调用 flip()类似)
- <<= 把 bitset 向左移动
- >>= 把 bitset 向右移动
- [x] 返回第 x 个位的引用

例如，以下代码创建一个 bitset，然后向左移动 4 个位:

```
// 创建一个 bitset
bitset bs2( (long) 131 );

cout << "bs2 is " << bs2 << endl;

// 向左移动 4 位
```

```
bs2 <<= 4;

cout << "now bs2 is " << bs2 << endl;
```

当上述代码运行时，显示：

```
bs2 is 10000011
now bs2 is 00110000
```

---

any

语法:

```
bool any();
```

any()函数返回真如果有位被设置为 1，否则返回假。

---

count

语法:

```
size_type count();
```

count()函数 bitset 中被设置成 1 的位的个数。

---

flip

语法:

```
bitset &flip();
bitset &flip( size_t pos );
```

flip()函数反置 bitset 中所有的位，即将 1 设为 0，0 设为 1。如果指定 pos，那么只有 pos 上的位被反置。

相关主题:

~operator

---

none

语法:

```
bool none();
```

none()返回真如果没有位被设为 1，否则返回假。

---

reset

语法:

```
bitset &reset();
bitset &reset( size_t pos );
```

reset()重置 bitset（全部设为 0），如果指定 pos，那么只有 pos 上的位被重置。

---

set

语法:

```
bitset &set();
bitset &set( size_t pos, int val=1 );
```

set()函数设置 bitset 上所有的位，然后返回 bitset。如果指定 pos，那么只有 pos 上的位被设置。

---

size

语法:

```
size_t size();
```

size()返回 bitset 能容纳的位。

test

语法:

```
bool test( size_t pos );
```

test()函数返回在 pos 上的位的值。

---

to_string

语法:

```
string to_string();
```

to_string()函数返回 bitset 的字符串形式。

---

to_ulong

语法:

```
unsigned long to_ulong();
```

to_ulong()返回 bitset 的无符号长整数形式。

1.15.2 C++ Double-Ended Queue

Constructors(见 [标题编号.])	创建一个新双向队列
Operators(见 [标题编号.])	比较和赋值双向队列
assign()(见 [标题编号.])	设置双向队列的值
at()(见 [标题编号.])	返回指定的元素
back()(见 [标题编号.])	返回最后一个元素
begin()(见 [标题编号.])	返回指向第一个元素的迭代器
clear()(见 [标题编号.])	删除所有元素
empty()(见 [标题编号.])	返回真如果双向队列为空
end()(见 [标题编号.])	返回指向尾部的迭代器
erase()(见 [标题编号.])	删除一个元素
front()(见 [标题编号.])	返回第一个元素
get_allocator()(见 [标题编号.])	返回双向队列的配置器

insert()(见 [标题编号.])	插入一个元素到双向队列中
max_size()(见 [标题编号.])	返回双向队列能容纳的最大元素个数
pop_back()(见 [标题编号.])	删除尾部的元素
pop_front()(见 [标题编号.])	删除头部的元素
push_back()(见 [标题编号.])	在尾部加入一个元素
push_front()(见 [标题编号.])	在头部加入一个元素
rbegin()(见 [标题编号.])	返回指向尾部的逆向迭代器
rend()(见 [标题编号.])	返回指向头部的逆向迭代器
resize()(见 [标题编号.])	改变双向队列的大小
size()(见 [标题编号.])	返回双向队列中元素的个数
swap()(见 [标题编号.])	和另一个双向队列交换元素

1.15.2.1 C++ Double-Ended Queue

Constructors

语法:

```
  deque();
  deque( size_type size );
  deque( size_type num, const TYPE(见 [标题编号.]) &val );
  deque( const deque &from );
  deque( input iterator(见 [标题编号.]) start, input iterator(见 [标题编号.]) end );
```

C++ Deques 能用以下方式创建:

- 无参，创建一个空双向队列
- *size* - 创建一个大小为 size 的双向队列
- *num* and *val* - 放置 num 个 val 的拷贝到队列中，
- *from* - 从 from 创建一个内容一样的双向队列
- *start* 和 *end* - 创建一个队列，保存从 start 到 end 的元素。

例如，下列代码创建并显示一个双向队列：

```
// 创建一个双向队列，里面有 10 个 1
deque dq( 10, 1 );
// 创建一个迭代器
```

```
deque::iterator iter;

// 显示这个双向队列
for( iter = dq.begin(); iter != dq.end(); iter++ ){
  cout << *iter << endl;
}
```

Operators

语法:

```
[]
```

你可以使用[]操作符访问双向队列中单个的元素。

---

assign

语法:

```
void assign( input iterator(见 [标题编号.]) start, input iterator(见 [标题编号.]) end);
void assign( Size num, const TYPE(见 [标题编号.]) &val );
```

assign()函数用 start 和 end 指示的范围为双向队列赋值,或者设置成 num 个 val。

---

at

语法:

```
reference at( size_type pos );
```

at()函数返回一个引用，指向双向队列中位置 pos 上的元素。

---

back

语法:

```
```

```
reference back();
```

back()返回一个引用，指向双向队列中最后一个元素。

---

begin

语法:

```
iterator(见 [标题编号.]) begin();
```

begin()函数返回一个迭代器(见 [标题编号.])，指向双向队列的第一个元素。

---

clear

语法:

```
void clear();
```

clear()函数删除双向队列中所有元素。

---

empty

语法:

```
bool empty();
```

empty()返回真如果双向队列为空，否则返回假。

---

end

语法:

```
iterator(见 [标题编号.]) end();
```

end()函数返回一个迭代器(见 [标题编号.])，指向双向队列的尾部。

erase

语法:

```
  iterator(见 [标题编号.]) erase( iterator(见 [标题编号.]) pos );
  iterator(见 [标题编号.]) erase( iterator(见 [标题编号.]) start,
iterator(见 [标题编号.]) end );
```

erase()函数删除 pos 位置上的元素，或者删除 start 和 end 之间的所有元素。返回值是一个 iterator(见 [标题编号.])，指向被删除元素的后一个元素。

front

语法:

```
  reference front();
```

front()函数返回一个引用，指向双向队列的头部。

get_allocator

语法:

```
  allocator_type get_allocator();
```

get_allocator()函数返回双向队列的配置器。

insert

语法:

```
  iterator(见 [标题编号.]) insert( iterator(见 [标题编号.]) pos,
size_type num, const TYPE(见 [标题编号.]) &val );
  void insert( iterator(见 [标题编号.]) pos, input_iterator(见 [标题编
号.]) start, input_iterator(见 [标题编号.]) end );
```

insert()在 pos 前插入 num 个 val 值，或者插入从 start 到 end 范围内的元素到 pos 前面。

---

max_size

语法:

```
size_type max_size();
```

max_size()返回双向队列能容纳的最大元素个数。

---

pop_back

语法:

```
void pop_back();
```

pop_back()删除双向队列尾部的元素。

---

pop_front

语法:

```
void pop_front();
```

pop_front()删除双向队列头部的元素。

---

push_back

语法:

```
void push_back( const TYPE(见 [标题编号.]) &val );
```

push_back()函数在双向队列的尾部加入一个值为 val 的元素。

---

push_front

语法:

```
void push_front( const TYPE(见 [标题编号.]) &val );
```

push_front()函数在双向队列的头部加入一个值为 val 的元素。

---

rbegin

语法:

```
reverse_iterator(见 [标题编号.]) rbegin();
```

rbegin()返回一个指向双向队列尾部的逆向迭代器(见 [标题编号.])。

---

rend

语法:

```
reverse_iterator(见 [标题编号.]) rend();
```

rend()返回一个指向双向队列头部的逆向迭代器(见 [标题编号.])。

---

resize

语法:

```
void resize( size_type num, TYPE(见 [标题编号.]) val );
```

resize()改变双向队列的大小为 num，另加入的元素都被填充为 val。

---

size

语法:

```
```

```
size_type size();
```

size()函数返回双向队列中的元素个数。

---

swap

语法:

```
void swap( deque &target );
```

swap()函数交换 target 和现双向队列中元素。

1.15.3 C++ List

assign()(见 [标题编号.])	给 list 赋值
back()(见 [标题编号.])	返回最后一个元素
begin()(见 [标题编号.])	返回指向第一个元素的迭代器
clear()(见 [标题编号.])	删除所有元素
empty()(见 [标题编号.])	如果 list 是空的则返回 true
end()(见 [标题编号.])	返回末尾的迭代器
erase()(见 [标题编号.])	删除一个元素
front()(见 [标题编号.])	返回第一个元素
get_allocator()(见 [标题编号.])	返回 list 的配置器
insert()(见 [标题编号.])	插入一个元素到 list 中
max_size()(见 [标题编号.])	返回 list 能容纳的最大元素数量
merge()(见 [标题编号.])	合并两个 list
pop_back()(见 [标题编号.])	删除最后一个元素
pop_front()(见 [标题编号.])	删除第一个元素
push_back()(见 [标题编号.])	在 list 的末尾添加一个元素
push_front()(见 [标题编号.])	在 list 的头部添加一个元素
rbegin()(见 [标题编号.])	返回指向第一个元素的逆向迭代器
remove()(见 [标题编号.])	从 list 删除元素
remove_if()(见 [标题编号.])	按指定条件删除元素
rend()(见 [标题编号.])	指向 list 末尾的逆向迭代器
resize()(见 [标题编号.])	改变 list 的大小
reverse()(见 [标题编号.])	把 list 的元素倒转

size()(见 [标题编号.])	返回 list 中的元素个数
sort()(见 [标题编号.])	给 list 排序
splice()(见 [标题编号.])	合并两个 list
swap()(见 [标题编号.])	交换两个 list
unique()(见 [标题编号.])	删除 list 中重复的元素

1.15.3.1 C++ List

赋值(assign)

语法:

```
  void assign( input_iterator(见 [标题编号.]) start, input_iterator(见
[标题编号.]) end );
  void assign( size_type num, const TYPE(见 [标题编号.]) &val );
```

assign()函数以迭代器 start 和 end 指示的范围为 list 赋值或者为 list 赋值 num 个以 val 为值的元素。

相关主题:

insert(),

---

back

语法:

```
  reference back();
```

back()函数返回一个引用，指向 list 的最后一个元素。

相关主题:

front(), pop_back(),

---

begin

语法:

```

```

```
  iterator(见 [标题编号.]) begin();
```

begin()函数返回一个迭代器(见 [标题编号.])，指向 list 的第一个元素。例如，

```
    // 创建一个元素类型是字符的链表
    list<char> charList;
    for( int i=0; i < 10; i++ )
      charList.push_front( i + 65 );

    // 显示这个链表
    list<char>::iterator theIterator;
    for( theIterator = charList.begin(); theIterator != charList.end();
theIterator++ )
      cout << *theIterator;
```

相关主题:

end(),

---

clear

语法:

```
  void clear();
```

clear()函数删除 list 的所有元素。

---

empty

语法:

```
  bool empty();
```

empty()函数返回真(true)如果链表为空，否则返回假。例如：

```
    list<int> the_list;
    for( int i = 0; i < 10; i++ )
      the_list.push_back( i );
    while( !the_list.empty() ) {
      cout << the_list.front() << endl;
      the_list.pop_front();
```

```
}
```

---

end

语法:

```
iterator(见 [标题编号.]) end();
```

end()函数返回一个迭代器(见 [标题编号.])，指向链表的末尾。

相关主题:

begin(),

---

erase

语法:

```
iterator(见 [标题编号.]) erase( iterator(见 [标题编号.]) pos );
iterator(见 [标题编号.]) erase( iterator(见 [标题编号.]) start,
iterator(见 [标题编号.]) end );
```

erase()函数删除以 pos 指示位置的元素，或者删除 *start* 和 *end* 之间的元素。返回值是一个迭代器(见 [标题编号.])，指向最后一个被删除元素的下一个元素。

---

front

语法:

```
reference front();
```

front()函数返回一个引用，指向链表的第一个元素。

```
list<int> the_list;
for( int i = 0; i < 10; i++ )
  the_list.push_back( i );
while( !the_list.empty() ) {
  cout << the_list.front() << endl;
```

```
    the_list.pop_front();
  }
```

相关主题:

back(),

---

get_allocator

语法:

```
allocator_type get_allocator();
```

get_allocator()函数返回链表的配置器。

---

insert

语法:

```
iterator(见 [标题编号.]) insert( iterator(见 [标题编号.]) pos, const TYPE(见 [标题编号.]) &val );
void insert( iterator(见 [标题编号.]) pos, size_type num, const TYPE(见 [标题编号.]) &val );
void insert( iterator(见 [标题编号.]) pos, input_iterator(见 [标题编号.]) start, input_iterator(见 [标题编号.]) end );
```

insert()插入元素 val 到位置 pos，或者插入 num 个元素 val 到 pos 之前，或者插入 start 到 end 之间的元素到 pos 的位置。返回值是一个迭代器(见 [标题编号.])，指向被插入的元素。

---

max_size

语法:

```
size_type max_size();
```

max_size()函数返回链表能够储存的元素数目。

---

merge

语法:

```
void merge( list &lst );
void merge( list &lst, Comp compfunction );
```

merge()函数把自己和 lst 链表连接在一起，产生一个整齐排列的组合链表。如果指定 compfunction，则将指定函数作为比较的依据。

---

pop_back

语法:

```
void pop_back();
```

pop_back()函数删除链表的最后一个元素。

相关主题:

pop_front(),

---

pop_front

语法:

```
void pop_front();
```

pop_front()函数删除链表的第一个元素。

相关主题:

pop_back(),

---

push_back

语法:

```
void push_back( const TYPE(见 [标题编号.]) &val );
```

push_back()将 val 连接到链表的最后。例如：

```
    list<int> the_list;
    for( int i = 0; i < 10; i++ )
      the_list.push_back( i );
```

相关主题:

push_front(),

---

push_front

Syntax:

```
  void push_front( const TYPE(见 [标题编号.]) &val );
```

push_front()函数将 val 连接到链表的头部。

相关主题:

push_back(),

---

rbegin

语法:

```
  reverse_iterator(见 [标题编号.]) rbegin();
```

rbegin()函数返回一个逆向迭代器(见 [标题编号.])，指向链表的末尾。

相关主题:

rend(),

---

remove

语法:

```
  void remove( const TYPE(见 [标题编号.]) &val );
```

remove()函数删除链表中所有值为 val 的元素。例如

```
    // 创建一个链表，元素是字母表的前 10 个元素
    list<char> charList;
    for( int i=0; i < 10; i++ )
      charList.push_front( i + 65 );
```

```
// 删除所有'E'的实例
charList.remove( 'E' );
```

---

remove_if

语法:

```
void remove_if( UnPred pr );
```

remove_if()以一元谓词 pr 为判断元素的依据，遍历整个链表。如果 pr 返回 true 则删除该元素。

---

rend

语法:

```
reverse_iterator(见 [标题编号.]) rend();
```

rend()函数迭代器(见 [标题编号.])指向链表的头部。

---

resize

语法:

```
void resize( size_type num, TYPE(见 [标题编号.]) val );
```

resize()函数把 list 的大小改变到 *num*。被加入的多余的元素都被赋值为 val

---

reverse

语法:

```
void reverse();
```

reverse()函数把 list 所有元素倒转。

size

语法:

```
size_type size();
```

size()函数返回 list 中元素的数量。

排序(sort)

语法:

```
void sort();
void sort( Comp compfunction );
```

sort()函数为链表排序，默认是升序。如果指定 compfunction 的话，就采用指定函数来判定两个元素的大小。

splice

语法:

```
void splice( iterator(见 [标题编号.]) pos, list &lst );
void splice( iterator(见 [标题编号.]) pos, list &lst, iterator(见 [标题编号.]) del );
void splice( iterator(见 [标题编号.]) pos, list &lst, iterator(见 [标题编号.]) start, iterator(见 [标题编号.]) end );
```

splice()函数把 lst 连接到 pos 的位置。如果指定其他参数，则插入 lst 中 del 所指元素到现链表的 pos 上，或者用 start 和 end 指定范围。

swap

语法:

```
void swap( list &lst );
```

swap()函数交换 lst 和现链表中的元素。

---

unique

语法:

```
void unique();
void unique( BinPred pr );
```

unique()函数删除链表中所有重复的元素。如果指定 pr，则使用 pr 来判定是否删除。

1.15.4 C++ Map

begin()(见 [标题编号.])	返回指向 map 头部的迭代器
clear()(见 [标题编号.])	删除所有元素
count()(见 [标题编号.])	返回指定元素出现的次数
empty()(见 [标题编号.])	如果 map 为空则返回 true
end()(见 [标题编号.])	返回指向 map 末尾的迭代器
equal_range()(见 [标题编号.])	返回特殊条目的迭代器对
erase()(见 [标题编号.])	删除一个元素
find()(见 [标题编号.])	查找一个元素
get_allocator()(见 [标题编号.])	返回 map 的配置器
insert()(见 [标题编号.])	插入元素
key_comp()(见 [标题编号.])	返回比较元素 key 的函数
lower_bound()(见 [标题编号.])	返回键值>=
max_size()(见 [标题编号.])	返回可以容纳的最大元素个数
rbegin()(见 [标题编号.])	返回一个指向 map 尾部的逆向迭代器
rend()(见 [标题编号.])	返回一个指向 map 头部的逆向迭代器
size()(见 [标题编号.])	返回 map 中元素的个数
swap()(见 [标题编号.])	交换两个 map
upper_bound()(见 [标题编号.])	返回键值>给定元素的第一个位置
value_comp()(见 [标题编号.])	返回比较元素 value 的函数

1.15.4.1 C++ Map

C++ Maps 被用作储存“关键字/值”对

---

begin

语法:

```
iterator(见 [标题编号.]) begin();
```

begin()函数返回一个迭代器(见 [标题编号.])指向 map 的第一个元素。

---

clear

语法:

```
void clear();
```

clear()函数删除 map 中的所有元素。

---

count

语法:

```
size_type count( const KEY TYPE(见 [标题编号.]) &key );
```

count()函数返回 map 中键值等于 key 的元素的个数。

---

empty

语法:

```
bool empty();
```

empty()函数返回真(true)如果 map 为空，否则返回假(false)。

end

语法:

```
  iterator(见 [标题编号.]) end();
```

end()函数返回一个迭代器(见 [标题编号.])指向 map 的尾部。

equal_range

Syntax:

```
  pair equal_range( const KEY_TYPE(见 [标题编号.]) &key );
```

equal_range()函数返回两个迭代器——一个指向第一个键值为 key 的元素，另一个指向最后一个键值为 key 的元素。

erase

语法:

```
  void erase( iterator(见 [标题编号.]) pos );
  void erase( iterator(见 [标题编号.]) start, iterator(见 [标题编号.])
end );
  size_type erase( const KEY_TYPE(见 [标题编号.]) &key );
```

erase()函数删除在 pos 位置的元素，或者删除在 start 和 end 之间的元素，或者删除那些值为 key 的所有元素。

find

语法:

```
  iterator(见 [标题编号.]) find( const KEY_TYPE(见 [标题编号.]) &key );
```

find()函数返回一个迭代器(见 [标题编号.])指向键值为 key 的元素，如果没找到就返回指向 map 尾部的迭代器(见 [标题编号.])。

---

get_allocator

语法:

```
allocator_type get_allocator();
```

get_allocator()函数返回 map 的配置器。

---

insert

语法:

```
iterator(见 [标题编号.]) insert( iterator(见 [标题编号.]) pos, const pair<KEY_TYPE(见 [标题编号.]),VALUE_TYPE(见 [标题编号.])> &val );
void insert( input iterator(见 [标题编号.]) start, input iterator(见 [标题编号.]) end );
pair<iterator, bool> insert( const pair<KEY_TYPE(见 [标题编号.]),VALUE_TYPE(见 [标题编号.])> &val );
```

insert()函数：

- 插入 val 到 pos 的后面，然后返回一个指向这个元素的迭代器(见 [标题编号.])。
- 插入 start 到 end 的元素到 map 中。
- 只有在 val 不存在时插入 val。返回值是一个指向被插入元素的迭代器(见 [标题编号.])和一个描述是否插入的 bool 值。

---

key_comp

语法:

```
key_compare key_comp();
```

key_comp()函数返回一个比较 key 的函数。

---

lower_bound

语法:

```
iterator(见 [标题编号.]) lower_bound( const KEY_TYPE(见 [标题编号.]) &key );
```

lower_bound()函数返回一个迭代器(见 [标题编号.])，指向 map 中键值>=

---

max_size

语法:

```
size_type max_size();
```

max_size()函数返回 map 能够保存的最大元素个数。

---

rbegin

语法:

```
reverse_iterator(见 [标题编号.]) rbegin();
```

rbegin()函数返回一个指向 map 尾部的逆向迭代器(见 [标题编号.])。

---

rend

语法:

```
reverse_iterator(见 [标题编号.]) rend();
```

rend()函数返回一个指向 map 头部的逆向迭代器(见 [标题编号.])。

---

size

语法:

```
size_type size();
```

size()函数返回 map 中保存的元素个数。

---

swap

语法:

```
void swap( map &obj );
```

swap()交换 obj 和现 map 中的元素。

---

upper_bound

语法:

```
iterator(见 [标题编号.]) upper_bound( const KEY_TYPE(见 [标题编号.]) &key );
```

upper_bound()函数返回一个迭代器(见 [标题编号.]),指向 map 中键值>key 的第一个元素。

---

value_comp

语法:

```
value_compare value_comp();
```

value_comp()函数返回一个比较元素 value 的函数。

1.15.5 C++ Multimap

begin()(见 [标题编号.])	返回指向第一个元素的迭代器
clear()(见 [标题编号.])	删除所有元素
count()(见 [标题编号.])	返回一个元素出现的次数

empty()(见 [标题编号.])	如果 multimap 为空则返回真
end()(见 [标题编号.])	返回一个指向 multimap 末尾的迭代器
equal_range()(见 [标题编号.])	返回指向元素的 key 为指定值的迭代器对
erase()(见 [标题编号.])	删除元素
find()(见 [标题编号.])	查找元素
get_allocator()(见 [标题编号.])	返回 multimap 的配置器
insert()(见 [标题编号.])	插入元素
key_comp()(见 [标题编号.])	返回比较 key 的函数
lower_bound()(见 [标题编号.])	返回键值>=
max_size()(见 [标题编号.])	返回可以容纳的最大元素个数
rbegin()(见 [标题编号.])	返回一个指向 mulitmap 尾部的逆向迭代器
rend()(见 [标题编号.])	返回一个指向 multimap 头部的逆向迭代器
size()(见 [标题编号.])	返回 multimap 中元素的个数
swap()(见 [标题编号.])	交换两个 multimaps
upper_bound()(见 [标题编号.])	返回键值>给定元素的第一个位置
value_comp()(见 [标题编号.])	返回比较元素 value 的函数

1.15.5.1 C++ Multimap

---

begin

语法:

```
iterator(见 [标题编号.]) begin();
```

begin()函数返回一个迭代器(见 [标题编号.]), 指向 multimap 的第一个元素。

---

clear

语法:

```
void clear();
```

clear()函数删除 multimap 中的所有元素。

---

count

语法:

```
size_type count( const key_type &key );
```

count()函数返回 multimap 中键值等于 key 的元素的个数。

---

empty

语法:

```
bool empty();
```

empty()函数返回真(true)如果 multimap 为空，否则返回假(false)。

---

end

语法:

```
iterator(见 [标题编号.]) end();
```

end()函数返回一个迭代器(见 [标题编号.])，指向 multimap 的尾部。

---

equal_range

语法:

```
pair equal_range( const key_type &key );
```

equal_range()函数查找 multimap 中键值等于 key 的所有元素，返回指示范围的两个迭代器。

---

erase

语法:

```
  void erase( iterator(见 [标题编号.]) pos );
  void erase( iterator(见 [标题编号.]) start, iterator(见 [标题编号.])
end );
  size_type erase( const key_type &key );
```

erase()函数删除在 pos 位置的元素，或者删除在 start 和 end 之间的元素，或者删除那些值为 key 的所有元素。

---

find

语法:

```
  iterator(见 [标题编号.]) find( const key_type &key );
```

find()函数返回一个迭代器(见 [标题编号.])指向键值为 key 的元素，如果没找到就返回指向 multimap 尾部的迭代器(见 [标题编号.])。

---

get_allocator

语法:

```
  allocator_type get_allocator();
```

get_allocator()函数返回 multimap 的配置器。

---

insert

语法:

```
  iterator(见 [标题编号.]) insert( iterator(见 [标题编号.]) pos, const
TYPE(见 [标题编号.]) &val );
  void insert( input iterator(见 [标题编号.]) start, input iterator(见
[标题编号.]) end );
  pair insert( const TYPE(见 [标题编号.]) &val );
```

insert()函数：

- 插入 val 到 pos 的后面，然后返回一个指向这个元素的迭代器(见 [标题编号.])。
- 插入 start 到 end 的元素到 multimap 中。
- 只有在 val 不存在时插入 val。返回值是一个指向被插入元素的迭代器(见 [标题编号.])和一个描述是否插入的 bool 值。

---

key_comp

语法:

```
key_compare key_comp();
```

key_comp()函数返回一个比较 key 的函数。

---

lower_bound

语法:

```
iterator(见 [标题编号.]) lower_bound( const key_type &key );
```

lower_bound()函数返回一个迭代器(见 [标题编号.])，指向 multimap 中键值>=

---

max_size

语法:

```
size_type max_size();
```

max_size()函数返回 multimap 能够保存的最大元素个数。

---

rbegin

语法:

```
reverse_iterator(见 [标题编号.]) rbegin();
```

rbegin()函数返回一个指向 multimap 尾部的逆向迭代器(见 [标题编号.])。

rend

语法:

```
reverse_iterator(见 [标题编号.]) rend();
```

rend()函数返回一个指向 multimap 头部的逆向迭代器(见 [标题编号.])。

size

语法:

```
size_type size();
```

size()函数返回 multimap 中保存的元素个数。

swap

语法:

```
void swap( multimap &obj );
```

swap()交换 obj 和现 mulitmap 中的元素。

upper_bound

语法:

```
iterator(见 [标题编号.]) upper_bound( const key_type &key );
```

upper_bound()函数返回一个迭代器(见 [标题编号.]),指向multimap中键值>key的第一个元素。

value_comp

语法:

```
value_compare value_comp();
```

value_comp()函数返回一个比较元素 value 的函数。

1.15.6 C++ Multiset

begin()(见 [标题编号.])	返回指向第一个元素的迭代器
clear()(见 [标题编号.])	清除所有元素
count()(见 [标题编号.])	返回指向某个值元素的个数
empty()(见 [标题编号.])	如果集合为空，返回 true
end()(见 [标题编	返回

号.])	指向最后一个元素的迭代器
equal_range()(见「标题编号.])	返回集合中与给定值相等的上下限的两个迭代器
erase()(见「标题编号.])	删除集合中的元素
find()(见「标题编号.])	返回一个指向被查找到元素的迭代器

get_allocator()(见[标题编号.])	返回多元集合的分配器
insert()(见[标题编号.])	在集合中插入元素
key_comp()(见[标题编号.])	返回一个用于元素间值比较的函数
lower_bound()(见[标题编号.])	返回指向大于（或等于）某值的第一个元素的迭代器

max_size()(见「标题编号.」)	返回集合能容纳的元素的最大限值
rbegin()(见「标题编号.」)	返回指向多元集合中最后一个元素的反向迭代器
rend()(见「标题编号.」)	返回指向多元集合中第一个元素的反向迭代器

size()(见「标题编号.」)	多元集合中元素的数目
swap()(见「标题编号.」)	交换两个多元集合变量
upper_bound()(见「标题编号.」)	返回一个大于某个值元素的迭代器
value_comp()(见「标题编号.」)	返回一个用于比较元素间的值的函数

1.15.6.1 C++ Multiset

```
void
clear();
```

清除当前集合中的所有元素。

语法:

```
bool
empty();
```

如果当前多元集合为空，返回 true；否则返回 false。

语法:

```
pair
equal_range( const
key_type &key );
```

返回集合中与给定值相等的上下限的两个迭代器。

```
allocator_type
get_allocator();
```

返回当前集合的分配器。

返回一个指向大于或者等于 key 值的第一个元素的迭代器。

返回指向当前多元集合中最后一个元素的反向迭代器。

返回当前多元集合中元素的数目。

```
upper_bound( const
key_type &key );
```

在当前多元集合中返回一个指向大于 Key 值的元素的迭代器

empty

语法:

```
bool empty();
```

empty()函数返回真(true)如果优先队列为空，否则返回假(false)。

---

pop

语法:

```
void pop();
```

pop()函数删除优先队列中的第一个元素。

---

push

语法:

```
void push( const TYPE(见 [标题编号.]) &val );
```

push()函数添加一个元素到优先队列中，值为 val。

---

size

语法:

```
size_type size();
```

size()函数返回优先队列中存储的元素个数。

---

top

语法:

```
TYPE(见 [标题编号.]) &top();
```

top()返回一个引用，指向优先队列中有最高优先级的元素。注意只有 pop()函数删除一个元素。

1.15.8 C++ Queue

C++队列是一种容器适配器，它给予程序员一种先进先出(FIFO)的数据结构。

back()(见 [标题编号.])	返回最后一个元素
empty()(见 [标题编号.])	如果队列空则返回真
front()(见 [标题编号.])	返回第一个元素
pop()(见 [标题编号.])	删除第一个元素
push()(见 [标题编号.])	在末尾加入一个元素
size()(见 [标题编号.])	返回队列中元素的个数

1.15.8.1 C++ Queue

---

back

语法:

```
TYPE(见 [标题编号.]) &back();
```

back()返回一个引用，指向队列的最后一个元素。

---

empty

语法:

```
bool empty();
```

empty()函数返回真(true)如果队列为空，否则返回假(false)。

---

front

语法:

```
TYPE(见 [标题编号.]) &front();
```

front()返回队列第一个元素的引用。

---

pop

语法:

```
void pop();
```

pop()函数删除队列的一个元素。

---

push

语法:

```
void push( const TYPE(见 [标题编号.]) &val );
```

push()函数往队列中加入一个元素。

---

size

语法:

```
size_type size();
```

size()返回队列中元素的个数。

1.15.9 C++ Set

```
begin()(见「标题编返回
```

号.])	指向第一个元素的迭代器
clear()(见「标题编号.])	清除所有元素
count()(见「标题编号.])	返回某个值元素的个数
empty()(见「标题编号.])	如果集合为空，返回 true
end()(见「标题编号.])	返回指向最后一个元素的迭代器
equal_range()(见「标题编号.])	返回集合

	中与给定值相等的上下限的两个迭代器
erase()(见「标题编号.」)	删除集合中的元素
find()(见「标题编号.」)	返回一个指向被查找到元素的迭代器
get_allocator()(见「标题编号.」)	返回集合的分配器
insert()(见「标题编号.」)	在集合中插入元素

lower_bound()(见「标题编号.」)	返回指向大于（或等于）某值的第一个元素的迭代器
key_comp()(见「标题编号.」)	返回一个用于元素间值比较的函数
max_size()(见「标题编号.」)	返回集合能容纳的元素的最大限值
rbegin()(见「标题	返回

编号.])	指向集合中最后一个元素的反向迭代器
rend()(见「标题编号.])	返回指向集合中第一个元素的反向迭代器
size()(见「标题编号.])	集合中元素的数目
swap()(见「标题编号.])	交换两个集合变量
upper_bound()(见「标题编号.])	返回大于某个

	值元素的迭代器
value_comp()(见[标题编号.])	返回一个用于比较元素间的值的函数

1.15.9.1 C++ Set

```
&key );
```

返回当前集合中出现的某个值的元素的数目。

`end();`

返回指向当前集合中最后一个元素的迭代器。

```
void erase(
iterator i );
void erase(
iterator start,
iterator end );
size_type
erase( const
key_type&key );
```

说明：

- 删除 i 元素；
- 删除从 start 开始到 end 结束的元素；
- 删除等于 key 值的所有元素（返回被删除的元素的个数）。

```
void insert(
input_iterator start,
input_iterator end );
pair insert( const TYPE
&val );
```

说明：

- 在迭代器 i 前插入 val；
- 将迭代器 start 开始到 end 结束返回内的元素插入到集合中；
- 在当前集合中插入 val 元素，并返回指向该元素的迭代器和一个布尔值来说明 val 是否成功的被插入了。

（应该注意的是在集合(Sets)中不能插入两个相同的元素。）

交换当前集合和 object 集合中的元素。

```
value_comp();
```
返回一个用于比较元素间的值的函数对象。

1.15.10 C++ Stack

C++ Stack（堆栈） 是一个容器类的改编，为程序员提供了堆栈的全部功能，——也就是说实现了一个先进后出（FILO）的数据结构。

操作(见 [标题编号.])	比较和分配堆栈
empty()(见 [标题编号.])	堆栈为空则返回真
pop()(见 [标题编号.])	移除栈顶元素
push()(见 [标题编号.])	在栈顶增加元素
size()(见 [标题编号.])	返回栈中元素数目
top()(见 [标题编号.])	返回栈顶元素

1.15.10.1 C++ Stack

操作

语法:

```
==
<=
>=
<
>
!=
```

所有的这些操作可以被用于堆栈. 相等指堆栈有相同的元素并有着相同的顺序。

---

empty

语法:

```
bool empty();
```

如当前堆栈为空，empty() 函数 返回 **true** 否则返回 **false**.

---

pop

语法:

```
void pop();
```

pop() 函数移除堆栈中最顶层元素。

相关主题:

top(),

---

push

Syntax:

```
void push( const TYPE &val );
```

push() 函数将 *val* 值压栈，使其成为栈顶的第一个元素。如:

```
stack<int> s;
for( int i=0; i < 10; i++ )
  s.push(i);
```

---

size

语法:

```
size_type size();
```

size() 函数返当前堆栈中的元素数目。如:

```
stack<int> s;
for( int i=0; i < 10; i++ )
  s.push(i);
cout << "This stack has a size of " << s.size() << endl;
```

---

top

语法:

```
TYPE &top();
```

top() 函数返回对栈顶元素的引用. 举例,如下代码显现和清空一个堆栈。

```
while( !s.empty() ) {
  cout << s.top() << " ";
  s.pop();
}
```

相关主题:

pop(),

1.15.11 C++ Vector

Vectors 包含着一系列连续存储的元素,其行为和数组类似。访问 Vector 中的任意元素或从末尾添加元素都可以在常量级时间复杂度(见 [标题编号.])内完成，而查找特定值的元素所处的位置或是在 Vector 中插入元素则是线性时间复杂度(见 [标题编号.])。

Constructors(见 [标题编号.])	构造函数
Operators(见 [标题编号.])	对 vector 进行赋值或比较
assign()(见 [标题编号.])	对 Vector 中的元素赋值
at()(见 [标题编号.])	返回指定位置的元素
back()(见 [标题编号.])	返回最末一个元素
begin()(见 [标题编号.])	返回第一个元素的迭代器
capacity()(见 [标题编号.])	返回 vector 所能容纳的元素数量(在不重新分配内存的情况下）
clear()(见 [标题编号.])	清空所有元素
empty()(见 [标题编号.])	判断 Vector 是否为空（返回 true 时为空）
end()(见 [标题编号.])	返回最末元素的迭代器(译注:实指向最末元素的下一个位置)
erase()(见 [标题编号.])	删除指定元素
front()(见 [标题编号.])	返回第一个元素
get_allocator()(见 [标题编号.])	返回 vector 的内存分配器
insert()(见 [标题编号.])	插入元素到 Vector 中
max_size()(见 [标题编号.])	返回 Vector 所能容纳元素的最大数量（上限）
pop_back()(见 [标题编号.])	移除最后一个元素
push_back()(见 [标题编号.])	在 Vector 最后添加一个元素
rbegin()(见 [标题编号.])	返回 Vector 尾部的逆迭代器
rend()(见 [标题编号.])	返回 Vector 起始的逆迭代器
reserve()(见 [标题编号.])	设置 Vector 最小的元素容纳数量

resize()(见 [标题编号.])	改变 Vector 元素数量的大小
size()(见 [标题编号.])	返回 Vector 元素数量的大小
swap()(见 [标题编号.])	交换两个 Vector

1.15.11.1 C++ Vector

构造函数

语法:

```
  vector();
  vector( size_type num, const TYPE(见 [标题编号.]) &val );
  vector( const vector &from );
  vector( input_iterator(见 [标题编号.]) start, input_iterator(见 [标题编号.]) end );
```

C++ Vectors 可以使用以下任意一种参数方式构造:

- 无参数 - 构造一个空的 vector,
- 数量(num)和值(val) - 构造一个初始放入 num 个值为 val 的元素的 Vector
- vector(from) - 构造一个与 vector from 相同的 vector
- 迭代器(start)和迭代器(end) - 构造一个初始值为[start,end)区间元素的 Vector(注:半开区间).

举例, 下面这个实例构造了一个包含 5 个值为 42 的元素的 Vector

```
vector<int> v1( 5, 42 );
```

运算符

语法:

```
  v1 == v2
  v1 != v2
  v1 <= v2
  v1 >= v2
  v1  v2
```

```
v[]
```

C++ Vectors 能够使用标准运算符: ==, !=, <=, >=, . 要访问 vector 中的某特定位置的元素可以使用 [] 操作符.

两个 vectors 被认为是相等的,如果:

1. 它们具有相同的容量
2. 所有相同位置的元素相等.

vectors 之间大小的比较是按照词典规则.
相关内容: **at()**.

---

assign 函数

语法:

```
void assign( input_iterator(见 [标题编号.]) start, input_iterator(见 [标题编号.]) end );
void assign( size_type num, const TYPE(见 [标题编号.]) &val );
```

assign() 函数要么将区间[start, end)的元素赋到当前 vector,或者赋 num 个值为 val 的元素到 vector 中.这个函数将会清除掉为 vector 赋值以前的内容.

---

at 函数

语法:

```
TYPE(见 [标题编号.]) at( size_type loc );
```

at() 函数 返回当前 Vector 指定位置 *loc 的元素的引用*. at() 函数 比 [] 运算符更加安全, 因为它不会让你去访问到Vector内越界的元素. 例如, 考虑下面的代码:

```
vector<int> v( 5, 1 );

for( int i = 0; i < 10; i++ ) {
  cout << "Element " << i << " is " << v[i] << endl;
```

```
}
```

这段代码访问了 vector 末尾以后的元素,这将可能导致很危险的结果.以下的代码将更加安全:

```
vector<int> v( 5, 1 );

for( int i = 0; i < 10; i++ ) {
  cout << "Element " << i << " is " << v.at(i) << endl;
}
```

取代试图访问内存里非法值的作法,at() 函数能够辨别出访问是否越界并在越界的时候抛出一个异常.

相关内容: **[] 操作符**

---

back 函数

语法:

```
TYPE(见 [标题编号.]) back();
```

back() 函数返回当前 vector 最末一个元素的引用.例如:

```
vector<int> v;

for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}

cout << "The first element is " << v.front()
     << " and the last element is " << v.back() << endl;
```

这段代码产生如下结果:

```
The first element is 0 and the last element is 4
```

相关内容: **front().**

---

begin 函数

语法:

```
iterator(见 [标题编号.]) begin();
```

begin()函数返回一个指向当前vector起始元素的迭代器(见 [标题编号.]).例如,下面这段使用了一个迭代器来显示出 vector 中的所有元素:

```
vector<int> v1( 5, 789 );
vector<int>::iterator it;
for( it = v1.begin(); it != v1.end(); it++ )
  cout << *it << endl;
```

相关内容: **end().**

---

capacity 函数

语法:

```
size_type capacity();
```

capacity() 函数 返回当前 vector 在重新进行内存分配以前所能容纳的元素数量.

---

clear 函数

语法:

```
void clear();
```

clear()函数删除当前 vector 中的所有元素.

---

empty 函数

语法:

```
bool empty();
```

如果当前 vector 没有容纳任何元素,则 empty()函数返回 true,否则返回 false.
例如,以下代码清空一个 vector,并按照逆序显示所有的元素:

```
vector<int> v;

for( int i = 0; i < 5; i++ ) {
    v.push_back(i);
  }

while( !v.empty() ) {
    cout << v.back() << endl;
    v.pop_back();
  }
```

相关内容: **size()**

---

end 函数

语法:

```
  iterator(见 [标题编号.]) end();
```

end() 函数返回一个指向当前 vector 末尾元素的**下一位置**的迭代器(见 [标题编号.]).注意,如果你要访问末尾元素,需要先将此迭代器自减 1.

相关内容: **begin()**

---

erase 函数

语法:

```
  iterator(见 [标题编号.]) erase( iterator(见 [标题编号.]) loc );
  iterator(见 [标题编号.]) erase( iterator(见 [标题编号.]) start,
iterator(见 [标题编号.]) end );
```

erase 函数要么删作指定位置 loc 的元素,要么删除区间[start, end)的所有元素.
返回值是指向删除的最后一个元素的下一位置的迭代器.例如:

```
// 创建一个 vector,置入字母表的前十个字符
vector<char> alphaVector;
for( int i=0; i < 10; i++ )
    alphaVector.push_back( i + 65 );
```

```
int size = alphaVector.size();

vector<char>::iterator startIterator;
vector<char>::iterator tempIterator;

for( int i=0; i < size; i++ )
{
    tartIterator = alphaVector.begin();
    alphaVector.erase( startIterator );

    // Display the vector
    for( tempIterator = alphaVector.begin(); tempIterator !=
alphaVector.end(); tempIterator++ )
    cout << *tempIterator;
    cout << endl;
}
```

这段代码将会显示如下输出:

```
BCDEFGHIJ
CDEFGHIJ
DEFGHIJ
EFGHIJ
FGHIJ
GHIJ
HIJ
IJ
J
```

相关内容: pop_back().

---

front 函数

语法:

```
TYPE(见 [标题编号.]) front();
```

front()函数返回当前 vector 起始元素的引用

相关内容: **back()**.

---

get_allocator 函数

语法:

```
allocator_type get_allocator();
```

get_allocator() 函数返回当前 vector 的内存分配器.

---

insert 函数

语法:

```
iterator(见 [标题编号.]) insert( iterator(见 [标题编号.]) loc, const
```

```
TYPE(见 [标题编号.]) &val );
  void insert( iterator(见 [标题编号.]) loc, size_type num, const TYPE(见
[标题编号.]) &val );
  void insert( iterator(见 [标题编号.]) loc, input iterator(见 [标题编
号.]) start, input_iterator(见 [标题编号.]) end );
```

insert() 函数有以下三种用法:

- 在指定位置 loc 前插入值为 val 的元素,返回指向这个元素的迭代器(见 [标题编号.]),
-
- 在指定位置 loc 前插入 num 个值为 val 的元素
-
- 在指定位置 loc 前插入区间[start, end)的所有元素 .
-

举例:

```
//创建一个 vector,置入字母表的前十个字符
vector<char> alphaVector;
for( int i=0; i < 10; i++ )
  alphaVector.push_back( i + 65 );

//插入四个 C 到 vector 中
vector<char>::iterator theIterator = alphaVector.begin();
alphaVector.insert( theIterator, 4, 'C' );

//显示 vector 的内容
for(theIterator = alphaVector.begin(); theIterator != alphaVector.end();
theIterator++ )
  cout << *theIterator;
```

这段代码将显示:

```
CCCCABCDEFGHIJ
```

---

max_size 函数

语法:

```
size_type max_size();
```

max_size() 函数返回当前 vector 所能容纳元素数量的最大值(译注:包括可重新分配内存).

---

pop_back

语法:

```
void pop_back();
```

pop_back()函数删除当前 vector 最末的一个元素,例如:

```
vector<char> alphaVector;
for( int i=0; i < 10; i++ )
    alphaVector.push_back( i + 65 );

int size = alphaVector.size();
vector<char>::iterator theIterator;
for( int i=0; i < size; i++ ) {
  alphaVector.pop_back();
  for( theIterator = alphaVector.begin(); theIterator !=
alphaVector.end(); theIterator++ )
      cout << *theIterator;
  cout << endl;
}
```

这段代码将显示以下输出:

```
ABCDEFGHI
ABCDEFGH
ABCDEFG
ABCDEF
ABCDE
ABCD
ABC
AB
A
```

相关内容: **erase()**.

---

push_back 函数

语法:

```
void push_back( const TYPE(见 [标题编号.]) &val );
```

push_back()添加值为 val 的元素到当前 vector 末尾

---

rbegin 函数

语法:

```
reverse_iterator(见 [标题编号.]) rbegin();
```

rbegin 函数返回指向当前 vector 末尾的逆迭代器(见 [标题编号.]).(译注:实际指向末尾的下一位置,而其内容为末尾元素的值,详见逆代器相关内容)

---

rend 函数

语法:

```
reverse_iterator(见 [标题编号.]) rend();
```

rend()函数返回指向当前 vector 起始位置的逆迭代器(见 [标题编号.]).

reserve 函数

语法:

```
void reserve( size_type size );
```

reserve()函数为当前 vector 预留至少共容纳 size 个元素的空间.(译注:实际空间可能大于 size)

resize 函数

语法:

```
void resize( size_type size, TYPE(见 [标题编号.]) val );
```

resize() 函数改变当前 vector 的大小为 size,且对新创建的元素赋值 val

size 函数

语法:

```
size_type size();
```

size() 函数返回当前 vector 所容纳元素的数目

相关内容: **empty()**

---

swap 函数

语法:

```
void swap( vector &from );
```

swap()函数交换当前 vector 与 vector from 的元素

1.15.12 Iterators

迭代器可被用来访问一个容器类的所包函的全部元素，其行为像一个指针。举一个例子，你可用一个迭代器来实现对 vector 容器中所含元素的遍历。有这么几种迭代器如下：

迭代器	描述
input_iterator	提供读功能的向前移动迭代器，它们可被进行增加(++)，比较与解引用（*）。
output_iterator	提供写功能的向前移动迭代器，它们可被进行增加(++)，比较与解引用（*）。
forward_iterator	可向前移动的，同时具有读写功能的迭代器。同时具有 input 和 output 迭代器的功能，并可对迭代器的值进行储存。
bidirectional_iterator	双向迭代器，同时提供读写功能，同 forward 迭代器，但可用来进行增加(++)或减少(--)操作。
random_iterator	随机迭代器，提供随机读写功能.是功能最强大的迭代器， 具有双向迭代器的全部功能，同时实现指针般的算术与比较运算。
reverse_iterator	如同随机迭代器或双向迭代器，但其移动是反向的 。（ Either a random iterator or a bidirectional iterator that moves in reverse direction.）（我不太理解它的行为）

第种容器类都联系于一种类型的迭代器。第个 STL 算法的实现使用某一类型的迭代器。举个例子，vector 容器类就有一个 **random-access** 随机迭代器，这也意味着其可以使用随机读写的算法。既然随机迭代器具有全部其它迭代器的特性，这也就是说为其它迭代器设计的算法也可被用在 vector 容器上。

如下代码对 vector 容器对象生成和使用了迭代器：

```
vector<int> the_vector;
vector<int>::iterator the_iterator;

for( int i=0; i < 10; i++ )
  the_vector.push_back(i);

int total = 0;
the_iterator = the_vector.begin();
while( the_iterator != the_vector.end() ) {
  total += *the_iterator;
  the_iterator++;
}
cout << "Total=" << total << endl;
```

提示：通过对一个迭代器的解引用操作（*），可以访问到容器所包含的元素。

1.16 所有的 C 函数

- abort(见 [标题编号.])

- ftell(见 [标题编号.])
- fwrite(见 [标题编号.])
- getc(见 [标题编号.])
- getchar(见 [标题编号.])
- getenv(见 [标题编号.])
- gets(见 [标题编号.])
- gmtime(见 [标题编号.])
- isalnum(见 [标题编号.])
- isalpha(见 [标题编号.])
- iscntrl(见 [标题编号.])
- isdigit(见 [标题编号.])
- isgraph(见 [标题编号.])
- islower(见 [标题编号.])
- isprint(见 [标题编号.])
- ispunct(见 [标题编号.])
- isspace(见 [标题编号.])
- isupper(见 [标题编号.])
- isxdigit(见 [标题编号.])
- labs(见 [标题编号.])
- ldexp(见 [标题编号.])
- ldiv(见 [标题编号.])
- localtime(见 [标题编号.])
- log(见 [标题编号.])
- log10(见 [标题编号.])
- longjmp(见 [标题编号.])
- malloc(见 [标题编号.])
- memchr(见 [标题编号.])
- memcmp(见 [标题编号.])
- memcpy(见 [标题编号.])
- memmove(见 [标题编号.])
- memset(见 [标题编号.])
- mktime(见 [标题编号.])
- modf(见 [标题编号.])
- perror(见 [标题编号.])
- pow(见 [标题编号.])
- printf(见 [标题编号.])
- putc(见 [标题编号.])
- putchar(见 [标题编号.])
- puts(见 [标题编号.])
- qsort(见 [标题编号.])
- raise(见 [标题编号.])
- rand(见 [标题编号.])
- realloc(见 [标题编号.])
- remove(见 [标题编号.])

-

1.17 所有的 C++ 函数

- Constructors(¼û [±êÌâ±à°Å.]) (deque)
- Constructors(¼û [±êÌâ±à°Å.]) (bitset)
- Constructors(¼û [±êÌâ±à°Å.]) (string)
- Constructors(¼û [±êÌâ±à°Å.]) (vector)
- Operators(¼û [±êÌâ±à°Å.]) (deque)
- Operators(¼û [±êÌâ±à°Å.]) (stack)
- Operators(¼û [±êÌâ±à°Å.]) (string)
- Operators(¼û [±êÌâ±à°Å.]) (vector)
- any(¼û [±êÌâ±à°Å.]) (bitset)
- append(¼û [±êÌâ±à°Å.]) (string)
- assign(¼û [±êÌâ±à°Å.]) (deque)
- assign(¼û [±êÌâ±à°Å.]) (list)
- assign(¼û [±êÌâ±à°Å.]) (string)
- assign(¼û [±êÌâ±à°Å.]) (vector)
- at(¼û [±êÌâ±à°Å.]) (deque)
- at(¼û [±êÌâ±à°Å.]) (string)
- at(¼û [±êÌâ±à°Å.]) (vector)
- back(¼û [±êÌâ±à°Å.]) (deque)
- back(¼û [±êÌâ±à°Å.]) (list)
- back(¼û [±êÌâ±à°Å.]) (queue)
- back(¼û [±êÌâ±à°Å.]) (vector)
- bad(¼û [±êÌâ±à°Å.]) (io)
- begin(¼û [±êÌâ±à°Å.]) (deque)
- begin(¼û [±êÌâ±à°Å.]) (list)
- begin(¼û [±êÌâ±à°Å.]) (map)
- begin(¼û [±êÌâ±à°Å.]) (multimap)
- begin(¼û [±êÌâ±à°Å.]) (multiset)
- begin(¼û [±êÌâ±à°Å.]) (set)
- begin(¼û [±êÌâ±à°Å.]) (string)
- begin(¼û [±êÌâ±à°Å.]) (vector)
- c_str(¼û [±êÌâ±à°Å.]) (string)
- capacity(¼û [±êÌâ±à°Å.]) (string)
- capacity(¼û [±êÌâ±à°Å.]) (vector)
- clear(¼û [±êÌâ±à°Å.]) (deque)
- clear(¼û [±êÌâ±à°Å.]) (io)
- clear(¼û [±êÌâ±à°Å.]) (list)
- clear(¼û [±êÌâ±à°Å.]) (map)
- clear(¼û [±êÌâ±à°Å.]) (multimap)
- clear(¼û [±êÌâ±à°Å.]) (multiset)
- clear(¼û [±êÌâ±à°Å.]) (set)

- clear(¼û [±êÌâ±àºÅ.]) (vector)
- compare(¼û [±êÌâ±àºÅ.]) (string)
- copy(¼û [±êÌâ±àºÅ.]) (string)
- count(¼û [±êÌâ±àºÅ.]) (bitset)
- count(¼û [±êÌâ±àºÅ.]) (map)
- count(¼û [±êÌâ±àºÅ.]) (multimap)
- count(¼û [±êÌâ±àºÅ.]) (multiset)
- count(¼û [±êÌâ±àºÅ.]) (set)
- data(¼û [±êÌâ±àºÅ.]) (string)
- empty(¼û [±êÌâ±àºÅ.]) (deque)
- empty(¼û [±êÌâ±àºÅ.]) (list)
- empty(¼û [±êÌâ±àºÅ.]) (map)
- empty(¼û [±êÌâ±àºÅ.]) (multimap)
- empty(¼û [±êÌâ±àºÅ.]) (multiset)
- empty(¼û [±êÌâ±àºÅ.]) (priorityqueue)
- empty(¼û [±êÌâ±àºÅ.]) (queue)
- empty(¼û [±êÌâ±àºÅ.]) (set)
- empty(¼û [±êÌâ±àºÅ.]) (stack)
- empty(¼û [±êÌâ±àºÅ.]) (string)
- empty(¼û [±êÌâ±àºÅ.]) (vector)
- end(¼û [±êÌâ±àºÅ.]) (deque)
- end(¼û [±êÌâ±àºÅ.]) (list)
- end(¼û [±êÌâ±àºÅ.]) (map)
- end(¼û [±êÌâ±àºÅ.]) (multimap)
- end(¼û [±êÌâ±àºÅ.]) (multiset)
- end(¼û [±êÌâ±àºÅ.]) (set)
- end(¼û [±êÌâ±àºÅ.]) (string)
- end(¼û [±êÌâ±àºÅ.]) (vector)
- eof(¼û [±êÌâ±àºÅ.]) (io)
- equal_range(¼û [±êÌâ±àºÅ.]) (map)
- equal_range(¼û [±êÌâ±àºÅ.]) (multimap)
- equal_range(¼û [±êÌâ±àºÅ.]) (multiset)
- equal_range(¼û [±êÌâ±àºÅ.]) (set)
- erase(¼û [±êÌâ±àºÅ.]) (deque)
- erase(¼û [±êÌâ±àºÅ.]) (list)
- erase(¼û [±êÌâ±àºÅ.]) (map)
- erase(¼û [±êÌâ±àºÅ.]) (multimap)
- erase(¼û [±êÌâ±àºÅ.]) (multiset)
- erase(¼û [±êÌâ±àºÅ.]) (set)
- erase(¼û [±êÌâ±àºÅ.]) (string)
- erase(¼û [±êÌâ±àºÅ.]) (vector)
- fail(¼û [±êÌâ±àºÅ.]) (io)
- fill(¼û [±êÌâ±àºÅ.]) (io)
- find(¼û [±êÌâ±àºÅ.]) (map)

- find(¼û [±êÌâ±à°Å.]) (multimap)
- find(¼û [±êÌâ±à°Å.]) (multiset)
- find(¼û [±êÌâ±à°Å.]) (set)
- find(¼û [±êÌâ±à°Å.]) (string)
- find_first_not_of(¼û [±êÌâ±à°Å.]) (string)
- find_first_of(¼û [±êÌâ±à°Å.]) (string)
- find_last_not_of(¼û [±êÌâ±à°Å.]) (string)
- find_last_of(¼û [±êÌâ±à°Å.]) (string)
- flags(¼û [±êÌâ±à°Å.]) (io)
- flip(¼û [±êÌâ±à°Å.]) (bitset)
- flush(¼û [±êÌâ±à°Å.]) (io)
- front(¼û [±êÌâ±à°Å.]) (deque)
- front(¼û [±êÌâ±à°Å.]) (list)
- front(¼û [±êÌâ±à°Å.]) (queue)
- front(¼û [±êÌâ±à°Å.]) (vector)
- fstream(¼û [±êÌâ±à°Å.]) (io)
- gcount(¼û [±êÌâ±à°Å.]) (io)
- get(¼û [±êÌâ±à°Å.]) (io)
- get_allocator(¼û [±êÌâ±à°Å.]) (deque)
- get_allocator(¼û [±êÌâ±à°Å.]) (list)
- get_allocator(¼û [±êÌâ±à°Å.]) (map)
- get_allocator(¼û [±êÌâ±à°Å.]) (multimap)
- get_allocator(¼û [±êÌâ±à°Å.]) (multiset)
- get_allocator(¼û [±êÌâ±à°Å.]) (set)
- get_allocator(¼û [±êÌâ±à°Å.]) (string)
- get_allocator(¼û [±êÌâ±à°Å.]) (vector)
- getline(¼û [±êÌâ±à°Å.]) (io)
- good(¼û [±êÌâ±à°Å.]) (io)
- ignore(¼û [±êÌâ±à°Å.]) (io)
- insert(¼û [±êÌâ±à°Å.]) (deque)
- insert(¼û [±êÌâ±à°Å.]) (list)
- insert(¼û [±êÌâ±à°Å.]) (map)
- insert(¼û [±êÌâ±à°Å.]) (multimap)
- insert(¼û [±êÌâ±à°Å.]) (multiset)
- insert(¼û [±êÌâ±à°Å.]) (set)
- insert(¼û [±êÌâ±à°Å.]) (string)
- insert(¼û [±êÌâ±à°Å.]) (vector)
- key_comp(¼û [±êÌâ±à°Å.]) (map)
- key_comp(¼û [±êÌâ±à°Å.]) (multimap)
- key_comp(¼û [±êÌâ±à°Å.]) (multiset)
- key_comp(¼û [±êÌâ±à°Å.]) (set)
- length(¼û [±êÌâ±à°Å.]) (string)
- lower_bound(¼û [±êÌâ±à°Å.]) (map)
- lower_bound(¼û [±êÌâ±à°Å.]) (multimap)

- lower_bound(¼û [±êÌâ±àºÅ.]) (multiset)
- lower_bound(¼û [±êÌâ±àºÅ.]) (set)
- max_size(¼û [±êÌâ±àºÅ.]) (deque)
- max_size(¼û [±êÌâ±àºÅ.]) (list)
- max_size(¼û [±êÌâ±àºÅ.]) (map)
- max_size(¼û [±êÌâ±àºÅ.]) (multimap)
- max_size(¼û [±êÌâ±àºÅ.]) (multiset)
- max_size(¼û [±êÌâ±àºÅ.]) (set)
- max_size(¼û [±êÌâ±àºÅ.]) (string)
- max_size(¼û [±êÌâ±àºÅ.]) (vector)
- merge(¼û [±êÌâ±àºÅ.]) (list)
- none(¼û [±êÌâ±àºÅ.]) (bitset)
- open(¼û [±êÌâ±àºÅ.]) (io)
- peek(¼û [±êÌâ±àºÅ.]) (io)
- pop(¼û [±êÌâ±àºÅ.]) (priorityqueue)
- pop(¼û [±êÌâ±àºÅ.]) (queue)
- pop(¼û [±êÌâ±àºÅ.]) (stack)
- pop_back(¼û [±êÌâ±àºÅ.]) (deque)
- pop_back(¼û [±êÌâ±àºÅ.]) (list)
- pop_back(¼û [±êÌâ±àºÅ.]) (vector)
- pop_front(¼û [±êÌâ±àºÅ.]) (deque)
- pop_front(¼û [±êÌâ±àºÅ.]) (list)
- precision(¼û [±êÌâ±àºÅ.]) (io)
- push(¼û [±êÌâ±àºÅ.]) (priorityqueue)
- push(¼û [±êÌâ±àºÅ.]) (queue)
- push(¼û [±êÌâ±àºÅ.]) (stack)
- push_back(¼û [±êÌâ±àºÅ.]) (deque)
- push_back(¼û [±êÌâ±àºÅ.]) (list)
- push_back(¼û [±êÌâ±àºÅ.]) (vector)
- push_front(¼û [±êÌâ±àºÅ.]) (deque)
- push_front(¼û [±êÌâ±àºÅ.]) (list)
- put(¼û [±êÌâ±àºÅ.]) (io)
- putback(¼û [±êÌâ±àºÅ.]) (io)
- rbegin(¼û [±êÌâ±àºÅ.]) (deque)
- rbegin(¼û [±êÌâ±àºÅ.]) (list)
- rbegin(¼û [±êÌâ±àºÅ.]) (map)
- rbegin(¼û [±êÌâ±àºÅ.]) (multimap)
- rbegin(¼û [±êÌâ±àºÅ.]) (multiset)
- rbegin(¼û [±êÌâ±àºÅ.]) (set)
- rbegin(¼û [±êÌâ±àºÅ.]) (string)
- rbegin(¼û [±êÌâ±àºÅ.]) (vector)
- rdstate(¼û [±êÌâ±àºÅ.]) (io)
- read(¼û [±êÌâ±àºÅ.]) (io)
- remove(¼û [±êÌâ±àºÅ.]) (list)

- remove_if(¼û [±êÌâ±àºÅ.]) (list)
- rend(¼û [±êÌâ±àºÅ.]) (deque)
- rend(¼û [±êÌâ±àºÅ.]) (list)
- rend(¼û [±êÌâ±àºÅ.]) (map)
- rend(¼û [±êÌâ±àºÅ.]) (multimap)
- rend(¼û [±êÌâ±àºÅ.]) (multiset)
- rend(¼û [±êÌâ±àºÅ.]) (set)
- rend(¼û [±êÌâ±àºÅ.]) (string)
- rend(¼û [±êÌâ±àºÅ.]) (vector)
- replace(¼û [±êÌâ±àºÅ.]) (string)
- reserve(¼û [±êÌâ±àºÅ.]) (string)
- reserve(¼û [±êÌâ±àºÅ.]) (vector)
- reset(¼û [±êÌâ±àºÅ.]) (bitset)
- resize(¼û [±êÌâ±àºÅ.]) (deque)
- resize(¼û [±êÌâ±àºÅ.]) (list)
- resize(¼û [±êÌâ±àºÅ.]) (string)
- resize(¼û [±êÌâ±àºÅ.]) (vector)
- reverse(¼û [±êÌâ±àºÅ.]) (list)
- rfind(¼û [±êÌâ±àºÅ.]) (string)
- seekg(¼û [±êÌâ±àºÅ.]) (io)
- seekp(¼û [±êÌâ±àºÅ.]) (io)
- set(¼û [±êÌâ±àºÅ.]) (bitset)
- setf(¼û [±êÌâ±àºÅ.]) (io)
- size(¼û [±êÌâ±àºÅ.]) (bitset)
- size(¼û [±êÌâ±àºÅ.]) (deque)
- size(¼û [±êÌâ±àºÅ.]) (list)
- size(¼û [±êÌâ±àºÅ.]) (map)
- size(¼û [±êÌâ±àºÅ.]) (multimap)
- size(¼û [±êÌâ±àºÅ.]) (multiset)
- size(¼û [±êÌâ±àºÅ.]) (priorityqueue)
- size(¼û [±êÌâ±àºÅ.]) (queue)
- size(¼û [±êÌâ±àºÅ.]) (set)
- size(¼û [±êÌâ±àºÅ.]) (stack)
- size(¼û [±êÌâ±àºÅ.]) (string)
- size(¼û [±êÌâ±àºÅ.]) (vector)
- sort(¼û [±êÌâ±àºÅ.]) (list)
- splice(¼û [±êÌâ±àºÅ.]) (list)
- substr(¼û [±êÌâ±àºÅ.]) (string)
- swap(¼û [±êÌâ±àºÅ.]) (deque)
- swap(¼û [±êÌâ±àºÅ.]) (list)
- swap(¼û [±êÌâ±àºÅ.]) (map)
- swap(¼û [±êÌâ±àºÅ.]) (multimap)
- swap(¼û [±êÌâ±àºÅ.]) (multiset)
- swap(¼û [±êÌâ±àºÅ.]) (set)

- swap(¼û [±êÌâ±à°Å.]) (string)
- swap(¼û [±êÌâ±à°Å.]) (vector)
- sync_with_stdio(¼û [±êÌâ±à°Å.]) (io)
- tellg(¼û [±êÌâ±à°Å.]) (io)
- tellp(¼û [±êÌâ±à°Å.]) (io)
- test(¼û [±êÌâ±à°Å.]) (bitset)
- to_string(¼û [±êÌâ±à°Å.]) (bitset)
- to_ulong(¼û [±êÌâ±à°Å.]) (bitset)
- top(¼û [±êÌâ±à°Å.]) (priorityqueue)
- top(¼û [±êÌâ±à°Å.]) (stack)
- unique(¼û [±êÌâ±à°Å.]) (list)
- unsetf(¼û [±êÌâ±à°Å.]) (io)
- upper_bound(¼û [±êÌâ±à°Å.]) (map)
- upper_bound(¼û [±êÌâ±à°Å.]) (multimap)
- upper_bound(¼û [±êÌâ±à°Å.]) (multiset)
- upper_bound(¼û [±êÌâ±à°Å.]) (set)
- value_comp(¼û [±êÌâ±à°Å.]) (map)
- value_comp(¼û [±êÌâ±à°Å.]) (multimap)
- value_comp(¼û [±êÌâ±à°Å.]) (multiset)
- value_comp(¼û [±êÌâ±à°Å.]) (set)
- width(¼û [±êÌâ±à°Å.]) (io)
- write(¼û [±êÌâ±à°Å.]) (io)

1.18 所有的 C/C++ 函数

- Constructors(¼û [±êÌâ±à°Å.]) (cppstring)
- Constructors(¼û [±êÌâ±à°Å.]) (cppvector)
- Operators(¼û [±êÌâ±à°Å.]) (cppbitset)
- Operators(¼û [±êÌâ±à°Å.]) (cppdeque)
- Operators(¼û [±êÌâ±à°Å.]) (cppstack)
- Operators(¼û [±êÌâ±à°Å.]) (cppstring)
- Operators(¼û [±êÌâ±à°Å.]) (cppvector)
- abort(¼û [±êÌâ±à°Å.]) (stdother)
- abs(¼û [±êÌâ±à°Å.]) (stdmath)
- acos(¼û [±êÌâ±à°Å.]) (stdmath)
- any(¼û [±êÌâ±à°Å.]) (cppbitset)
- append(¼û [±êÌâ±à°Å.]) (cppstring)
- asctime(¼û [±êÌâ±à°Å.]) (stddate)
- asin(¼û [±êÌâ±à°Å.]) (stdmath)
- assert(¼û [±êÌâ±à°Å.]) (stdother)
- assign(¼û [±êÌâ±à°Å.]) (cppdeque)
- assign(¼û [±êÌâ±à°Å.]) (cpplist)
- assign(¼û [±êÌâ±à°Å.]) (cppstring)
- assign(¼û [±êÌâ±à°Å.]) (cppvector)

- count(¼û [±êÌâ±à°Å.]) (cppmultimap)
- count(¼û [±êÌâ±à°Å.]) (cppmultiset)
- count(¼û [±êÌâ±à°Å.]) (cppset)
- ctime(¼û [±êÌâ±à°Å.]) (stddate)
- data(¼û [±êÌâ±à°Å.]) (cppstring)
- #define(¼û [±êÌâ±à°Å.]) (preproc)
- difftime(¼û [±êÌâ±à°Å.]) (stddate)
- div(¼û [±êÌâ±à°Å.]) (stdmath)
- empty(¼û [±êÌâ±à°Å.]) (cppdeque)
- empty(¼û [±êÌâ±à°Å.]) (cpplist)
- empty(¼û [±êÌâ±à°Å.]) (cppmap)
- empty(¼û [±êÌâ±à°Å.]) (cppmultimap)
- empty(¼û [±êÌâ±à°Å.]) (cppmultiset)
- empty(¼û [±êÌâ±à°Å.]) (cpppriorityqueue)
- empty(¼û [±êÌâ±à°Å.]) (cppqueue)
- empty(¼û [±êÌâ±à°Å.]) (cppset)
- empty(¼û [±êÌâ±à°Å.]) (cppstack)
- empty(¼û [±êÌâ±à°Å.]) (cppstring)
- empty(¼û [±êÌâ±à°Å.]) (cppvector)
- end(¼û [±êÌâ±à°Å.]) (cppdeque)
- end(¼û [±êÌâ±à°Å.]) (cpplist)
- end(¼û [±êÌâ±à°Å.]) (cppmap)
- end(¼û [±êÌâ±à°Å.]) (cppmultimap)
- end(¼û [±êÌâ±à°Å.]) (cppmultiset)
- end(¼û [±êÌâ±à°Å.]) (cppset)
- end(¼û [±êÌâ±à°Å.]) (cppstring)
- end(¼û [±êÌâ±à°Å.]) (cppvector)
- eof(¼û [±êÌâ±à°Å.]) (cppio)
- equal_range(¼û [±êÌâ±à°Å.]) (cppmap)
- equal_range(¼û [±êÌâ±à°Å.]) (cppmultimap)
- equal_range(¼û [±êÌâ±à°Å.]) (cppmultiset)
- equal_range(¼û [±êÌâ±à°Å.]) (cppset)
- erase(¼û [±êÌâ±à°Å.]) (cppdeque)
- erase(¼û [±êÌâ±à°Å.]) (cpplist)
- erase(¼û [±êÌâ±à°Å.]) (cppmap)
- erase(¼û [±êÌâ±à°Å.]) (cppmultimap)
- erase(¼û [±êÌâ±à°Å.]) (cppmultiset)
- erase(¼û [±êÌâ±à°Å.]) (cppset)
- erase(¼û [±êÌâ±à°Å.]) (cppstring)
- erase(¼û [±êÌâ±à°Å.]) (cppvector)
- #error(¼û [±êÌâ±à°Å.]) (preproc)
- exit(¼û [±êÌâ±à°Å.]) (stdother)
- exp(¼û [±êÌâ±à°Å.]) (stdmath)
- fabs(¼û [±êÌâ±à°Å.]) (stdmath)

- fail(¼û [±êÌâ±àºÅ.]) (cppio)
- fclose(¼û [±êÌâ±àºÅ.]) (stdio)
- feof(¼û [±êÌâ±àºÅ.]) (stdio)
- ferror(¼û [±êÌâ±àºÅ.]) (stdio)
- fflush(¼û [±êÌâ±àºÅ.]) (stdio)
- fgetc(¼û [±êÌâ±àºÅ.]) (stdio)
- fgetpos(¼û [±êÌâ±àºÅ.]) (stdio)
- fgets(¼û [±êÌâ±àºÅ.]) (stdio)
- fill(¼û [±êÌâ±àºÅ.]) (cppio)
- find(¼û [±êÌâ±àºÅ.]) (cppmap)
- find(¼û [±êÌâ±àºÅ.]) (cppmultimap)
- find(¼û [±êÌâ±àºÅ.]) (cppmultiset)
- find(¼û [±êÌâ±àºÅ.]) (cppset)
- find(¼û [±êÌâ±àºÅ.]) (cppstring)
- find_first_not_of(¼û [±êÌâ±àºÅ.]) (cppstring)
- find_first_of(¼û [±êÌâ±àºÅ.]) (cppstring)
- find_last_not_of(¼û [±êÌâ±àºÅ.]) (cppstring)
- find_last_of(¼û [±êÌâ±àºÅ.]) (cppstring)
- flags(¼û [±êÌâ±àºÅ.]) (cppio)
- flip(¼û [±êÌâ±àºÅ.]) (cppbitset)
- floor(¼û [±êÌâ±àºÅ.]) (stdmath)
- flush(¼û [±êÌâ±àºÅ.]) (cppio)
- fmod(¼û [±êÌâ±àºÅ.]) (stdmath)
- fopen(¼û [±êÌâ±àºÅ.]) (stdio)
- fprintf(¼û [±êÌâ±àºÅ.]) (stdio)
- fputc(¼û [±êÌâ±àºÅ.]) (stdio)
- fputs(¼û [±êÌâ±àºÅ.]) (stdio)
- fread(¼û [±êÌâ±àºÅ.]) (stdio)
- free(¼û [±êÌâ±àºÅ.]) (stdmem)
- freopen(¼û [±êÌâ±àºÅ.]) (stdio)
- frexp(¼û [±êÌâ±àºÅ.]) (stdmath)
- front(¼û [±êÌâ±àºÅ.]) (cppdeque)
- front(¼û [±êÌâ±àºÅ.]) (cpplist)
- front(¼û [±êÌâ±àºÅ.]) (cppqueue)
- front(¼û [±êÌâ±àºÅ.]) (cppvector)
- fscanf(¼û [±êÌâ±àºÅ.]) (stdio)
- fseek(¼û [±êÌâ±àºÅ.]) (stdio)
- fsetpos(¼û [±êÌâ±àºÅ.]) (stdio)
- fstream(¼û [±êÌâ±àºÅ.]) (cppio)
- ftell(¼û [±êÌâ±àºÅ.]) (stdio)
- fwrite(¼û [±êÌâ±àºÅ.]) (stdio)
- gcount(¼û [±êÌâ±àºÅ.]) (cppio)
- get(¼û [±êÌâ±àºÅ.]) (cppio)
- get_allocator(¼û [±êÌâ±àºÅ.]) (cppdeque)

- #line(¼û [±êÌâ±à°Å.]) (preproc)
- localtime(¼û [±êÌâ±à°Å.]) (stddate)
- log(¼û [±êÌâ±à°Å.]) (stdmath)
- log10(¼û [±êÌâ±à°Å.]) (stdmath)
- longjmp(¼û [±êÌâ±à°Å.]) (stdother)
- lower_bound(¼û [±êÌâ±à°Å.]) (cppmap)
- lower_bound(¼û [±êÌâ±à°Å.]) (cppmultimap)
- lower_bound(¼û [±êÌâ±à°Å.]) (cppmultiset)
- lower_bound(¼û [±êÌâ±à°Å.]) (cppset)
- malloc(¼û [±êÌâ±à°Å.]) (stdmem)
- max_size(¼û [±êÌâ±à°Å.]) (cppdeque)
- max_size(¼û [±êÌâ±à°Å.]) (cpplist)
- max_size(¼û [±êÌâ±à°Å.]) (cppmap)
- max_size(¼û [±êÌâ±à°Å.]) (cppmultimap)
- max_size(¼û [±êÌâ±à°Å.]) (cppmultiset)
- max_size(¼û [±êÌâ±à°Å.]) (cppset)
- max_size(¼û [±êÌâ±à°Å.]) (cppstring)
- max_size(¼û [±êÌâ±à°Å.]) (cppvector)
- memchr(¼û [±êÌâ±à°Å.]) (stdstring)
- memcmp(¼û [±êÌâ±à°Å.]) (stdstring)
- memcpy(¼û [±êÌâ±à°Å.]) (stdstring)
- memmove(¼û [±êÌâ±à°Å.]) (stdstring)
- memset(¼û [±êÌâ±à°Å.]) (stdstring)
- merge(¼û [±êÌâ±à°Å.]) (cpplist)
- mktime(¼û [±êÌâ±à°Å.]) (stddate)
- modf(¼û [±êÌâ±à°Å.]) (stdmath)
- none(¼û [±êÌâ±à°Å.]) (cppbitset)
- open(¼û [±êÌâ±à°Å.]) (cppio)
- peek(¼û [±êÌâ±à°Å.]) (cppio)
- perror(¼û [±êÌâ±à°Å.]) (stdio)
- pop(¼û [±êÌâ±à°Å.]) (cppprorityqueue)
- pop(¼û [±êÌâ±à°Å.]) (cppqueue)
- pop(¼û [±êÌâ±à°Å.]) (cppstack)
- pop_back(¼û [±êÌâ±à°Å.]) (cppdeque)
- pop_back(¼û [±êÌâ±à°Å.]) (cpplist)
- pop_back(¼û [±êÌâ±à°Å.]) (cppvector)
- pop_front(¼û [±êÌâ±à°Å.]) (cppdeque)
- pop_front(¼û [±êÌâ±à°Å.]) (cpplist)
- pow(¼û [±êÌâ±à°Å.]) (stdmath)
- #pragma(¼û [±êÌâ±à°Å.]) (preproc)
- precision(¼û [±êÌâ±à°Å.]) (cppio)
- Predefined variables(¼û [±êÌâ±à°Å.]) (preproc)
- printf(¼û [±êÌâ±à°Å.]) (stdio)
- push(¼û [±êÌâ±à°Å.]) (cppprorityqueue)

- push(¼û [±êÌâ±à°Å.]) (cppqueue)
- push(¼û [±êÌâ±à°Å.]) (cppstack)
- push_back(¼û [±êÌâ±à°Å.]) (cppdeque)
- push_back(¼û [±êÌâ±à°Å.]) (cpplist)
- push_back(¼û [±êÌâ±à°Å.]) (cppvector)
- push_front(¼û [±êÌâ±à°Å.]) (cppdeque)
- push_front(¼û [±êÌâ±à°Å.]) (cpplist)
- put(¼û [±êÌâ±à°Å.]) (cppio)
- putback(¼û [±êÌâ±à°Å.]) (cppio)
- putc(¼û [±êÌâ±à°Å.]) (stdio)
- putchar(¼û [±êÌâ±à°Å.]) (stdio)
- puts(¼û [±êÌâ±à°Å.]) (stdio)
- qsort(¼û [±êÌâ±à°Å.]) (stdother)
- raise(¼û [±êÌâ±à°Å.]) (stdother)
- rand(¼û [±êÌâ±à°Å.]) (stdother)
- rbegin(¼û [±êÌâ±à°Å.]) (cppdeque)
- rbegin(¼û [±êÌâ±à°Å.]) (cpplist)
- rbegin(¼û [±êÌâ±à°Å.]) (cppmap)
- rbegin(¼û [±êÌâ±à°Å.]) (cppmultimap)
- rbegin(¼û [±êÌâ±à°Å.]) (cppmultiset)
- rbegin(¼û [±êÌâ±à°Å.]) (cppset)
- rbegin(¼û [±êÌâ±à°Å.]) (cppstring)
- rbegin(¼û [±êÌâ±à°Å.]) (cppvector)
- rdstate(¼û [±êÌâ±à°Å.]) (cppio)
- read(¼û [±êÌâ±à°Å.]) (cppio)
- realloc(¼û [±êÌâ±à°Å.]) (stdmem)
- remove(¼û [±êÌâ±à°Å.]) (cpplist)
- remove(¼û [±êÌâ±à°Å.]) (stdio)
- remove_if(¼û [±êÌâ±à°Å.]) (cpplist)
- rename(¼û [±êÌâ±à°Å.]) (stdio)
- rend(¼û [±êÌâ±à°Å.]) (cppdeque)
- rend(¼û [±êÌâ±à°Å.]) (cpplist)
- rend(¼û [±êÌâ±à°Å.]) (cppmap)
- rend(¼û [±êÌâ±à°Å.]) (cppmultimap)
- rend(¼û [±êÌâ±à°Å.]) (cppmultiset)
- rend(¼û [±êÌâ±à°Å.]) (cppset)
- rend(¼û [±êÌâ±à°Å.]) (cppstring)
- rend(¼û [±êÌâ±à°Å.]) (cppvector)
- replace(¼û [±êÌâ±à°Å.]) (cppstring)
- reserve(¼û [±êÌâ±à°Å.]) (cppstring)
- reserve(¼û [±êÌâ±à°Å.]) (cppvector)
- reset(¼û [±êÌâ±à°Å.]) (cppbitset)
- resize(¼û [±êÌâ±à°Å.]) (cppdeque)
- resize(¼û [±êÌâ±à°Å.]) (cpplist)

- resize(¼û [±êÌâ±àºÅ.]) (cppstring)
- resize(¼û [±êÌâ±àºÅ.]) (cppvector)
- reverse(¼û [±êÌâ±àºÅ.]) (cpplist)
- rewind(¼û [±êÌâ±àºÅ.]) (stdio)
- rfind(¼û [±êÌâ±àºÅ.]) (cppstring)
- scanf(¼û [±êÌâ±àºÅ.]) (stdio)
- seekg(¼û [±êÌâ±àºÅ.]) (cppio)
- seekp(¼û [±êÌâ±àºÅ.]) (cppio)
- set(¼û [±êÌâ±àºÅ.]) (cppbitset)
- setbuf(¼û [±êÌâ±àºÅ.]) (stdio)
- setf(¼û [±êÌâ±àºÅ.]) (cppio)
- setjmp(¼û [±êÌâ±àºÅ.]) (stdother)
- setvbuf(¼û [±êÌâ±àºÅ.]) (stdio)
- #, ##(¼û [±êÌâ±àºÅ.]) (preproc)
- signal(¼û [±êÌâ±àºÅ.]) (stdother)
- sin(¼û [±êÌâ±àºÅ.]) (stdmath)
- sinh(¼û [±êÌâ±àºÅ.]) (stdmath)
- size(¼û [±êÌâ±àºÅ.]) (cppbitset)
- size(¼û [±êÌâ±àºÅ.]) (cppdeque)
- size(¼û [±êÌâ±àºÅ.]) (cpplist)
- size(¼û [±êÌâ±àºÅ.]) (cppmap)
- size(¼û [±êÌâ±àºÅ.]) (cppmultimap)
- size(¼û [±êÌâ±àºÅ.]) (cppmultiset)
- size(¼û [±êÌâ±àºÅ.]) (cpppriorityqueue)
- size(¼û [±êÌâ±àºÅ.]) (cppqueue)
- size(¼û [±êÌâ±àºÅ.]) (cppset)
- size(¼û [±êÌâ±àºÅ.]) (cppstack)
- size(¼û [±êÌâ±àºÅ.]) (cppstring)
- size(¼û [±êÌâ±àºÅ.]) (cppvector)
- sort(¼û [±êÌâ±àºÅ.]) (cpplist)
- splice(¼û [±êÌâ±àºÅ.]) (cpplist)
- sprintf(¼û [±êÌâ±àºÅ.]) (stdio)
- sqrt(¼û [±êÌâ±àºÅ.]) (stdmath)
- srand(¼û [±êÌâ±àºÅ.]) (stdother)
- sscanf(¼û [±êÌâ±àºÅ.]) (stdio)
- strcat(¼û [±êÌâ±àºÅ.]) (stdstring)
- strchr(¼û [±êÌâ±àºÅ.]) (stdstring)
- strcmp(¼û [±êÌâ±àºÅ.]) (stdstring)
- strcoll(¼û [±êÌâ±àºÅ.]) (stdstring)
- strcpy(¼û [±êÌâ±àºÅ.]) (stdstring)
- strcspn(¼û [±êÌâ±àºÅ.]) (stdstring)
- strerror(¼û [±êÌâ±àºÅ.]) (stdstring)
- strftime(¼û [±êÌâ±àºÅ.]) (stddate)
- strlen(¼û [±êÌâ±àºÅ.]) (stdstring)

- strncat(¼û [±êÌâ±à°Å.]) (stdstring)
- strncmp(¼û [±êÌâ±à°Å.]) (stdstring)
- strncpy(¼û [±êÌâ±à°Å.]) (stdstring)
- strpbrk(¼û [±êÌâ±à°Å.]) (stdstring)
- strrchr(¼û [±êÌâ±à°Å.]) (stdstring)
- strspn(¼û [±êÌâ±à°Å.]) (stdstring)
- strstr(¼û [±êÌâ±à°Å.]) (stdstring)
- strtod(¼û [±êÌâ±à°Å.]) (stdstring)
- strtok(¼û [±êÌâ±à°Å.]) (stdstring)
- strtol(¼û [±êÌâ±à°Å.]) (stdstring)
- strtoul(¼û [±êÌâ±à°Å.]) (stdstring)
- strxfrm(¼û [±êÌâ±à°Å.]) (stdstring)
- substr(¼û [±êÌâ±à°Å.]) (cppstring)
- swap(¼û [±êÌâ±à°Å.]) (cppdeque)
- swap(¼û [±êÌâ±à°Å.]) (cpplist)
- swap(¼û [±êÌâ±à°Å.]) (cppmap)
- swap(¼û [±êÌâ±à°Å.]) (cppmultimap)
- swap(¼û [±êÌâ±à°Å.]) (cppmultiset)
- swap(¼û [±êÌâ±à°Å.]) (cppset)
- swap(¼û [±êÌâ±à°Å.]) (cppstring)
- swap(¼û [±êÌâ±à°Å.]) (cppvector)
- sync_with_stdio(¼û [±êÌâ±à°Å.]) (cppio)
- system(¼û [±êÌâ±à°Å.]) (stdother)
- tan(¼û [±êÌâ±à°Å.]) (stdmath)
- tanh(¼û [±êÌâ±à°Å.]) (stdmath)
- tellg(¼û [±êÌâ±à°Å.]) (cppio)
- tellp(¼û [±êÌâ±à°Å.]) (cppio)
- test(¼û [±êÌâ±à°Å.]) (cppbitset)
- time(¼û [±êÌâ±à°Å.]) (stddate)
- tmpfile(¼û [±êÌâ±à°Å.]) (stdio)
- tmpnam(¼û [±êÌâ±à°Å.]) (stdio)
- to_string(¼û [±êÌâ±à°Å.]) (cppbitset)
- to_ulong(¼û [±êÌâ±à°Å.]) (cppbitset)
- tolower(¼û [±êÌâ±à°Å.]) (stdstring)
- top(¼û [±êÌâ±à°Å.]) (cppprorityqueue)
- top(¼û [±êÌâ±à°Å.]) (cppstack)
- toupper(¼û [±êÌâ±à°Å.]) (stdstring)
- #undef(¼û [±êÌâ±à°Å.]) (preproc)
- ungetc(¼û [±êÌâ±à°Å.]) (stdio)
- unique(¼û [±êÌâ±à°Å.]) (cpplist)
- unsetf(¼û [±êÌâ±à°Å.]) (cppio)
- upper_bound(¼û [±êÌâ±à°Å.]) (cppmap)
- upper_bound(¼û [±êÌâ±à°Å.]) (cppmultimap)
- upper_bound(¼û [±êÌâ±à°Å.]) (cppmultiset)

- upper_bound(¼û [±êÌâ±à°Å.]) (cppset)
- va_arg(¼û [±êÌâ±à°Å.]) (stdother)
- value_comp(¼û [±êÌâ±à°Å.]) (cppmap)
- value_comp(¼û [±êÌâ±à°Å.]) (cppmultimap)
- value_comp(¼û [±êÌâ±à°Å.]) (cppmultiset)
- value_comp(¼û [±êÌâ±à°Å.]) (cppset)
- vprintf, vfprintf, (stdio)
- width(¼û [±êÌâ±à°Å.]) (cppio)
- write(¼û [±êÌâ±à°Å.]) (cppio)

1.19 感谢

cppreference.com(¼û [±êÌâ±à°Å.]) -> ¸ĞĞ»

´ËÎÄµµÓ¢ÎÄÔ ÎÄÕª ×Ôwww.cppreference.com

ÒÔÏÂC-FreeµÄÖ§³ÖÕß£¬Îª ÎÄµµµÄ• Òë×ö³ö¹±Ï×£¬ÌØ´Ë¸ĞĞ»£º

Dreamby , sirius , Vic
Zhang , Lauren.Jc , Rexzhou , power , littlestone

¡¡

2. Addenda

2.1 Complexity

There are different measurements of the speed of any given algorithm. Given an input size of **N**, they can be described as follows:

Name	Speed	Description
exponential time	slow	takes an amount of time proportional to a constant raised to the **N**th power (K^**N**)
polynomial time	fast	takes an amount of time proportional to **N** raised to some constant power (**N**^K)
linear time	faster	takes an amount of time directly proportional to **N** (K * **N**)

logarithmic time	much faster	takes an amount of time proportional to the logarithm of **N** (log(**N**))
constant time	fastest	takes a fixed amount of time, no matter how large the input is (K)

2.2 C++ Containers

C++ 容器 (如 vectors(向量), lists(链表), 等.) 是一种能够容纳不同类型数据的通用容器. 例如, 下面的代码常见一个能够容纳 int 类型数据的 vector(见 [标题编号.]):

```
vector<int> v;
```

只要容器中的对象能够满足如下条件, C++容器既能够容纳 C++内建的类型对象 (像上面例子中的 **int** 类型) 也能够自定义的对象:

- 对象必须有默认构造函数,
- 有一个可访问的析构函数, 并且
- 有一个可访问的赋值操作符重载函数(operator=).

当我们在本文档中描述各种容器的功能的时候, 我们用 **TYPE** 来表示容器所包含的数据的类型. 例如, 在上面例子中, **TYPE** 表示 **int**.

2.3 C++ I/O Flags

C++ I/O 标志

C++为标准输入和输出定义了一些格式标志, 它可以通过 flags()(见 [标题编号.]), setf()(见 [标题编号.]), 和 unsetf()(见 [标题编号.]) 三个函数来控制. 例如,

```
cout.setf(ios::left);
```

对所有指向 **cout** 的输出进行左对齐调整.

标志	功能
boolalpha	可以使用单词"true"和"false"进行输入/输出的布尔值.
dec	用十进制格式显示后面的数值.
fixed	用正常的记数方法显示浮点数(与科学计数法相对应).
hex	用十六进制格式显示后面的数值.

internal	将填充字符回到符号和数值之间.
left	输出调整为左对齐.
oct	用八进制格式显示后面的数值.
right	输出调整为右对齐.
scientific	用科学记数法显示浮点数.
showbase	输出时显示所有数值的基数.
showpoint	显示小数点和额外的零，即使不需要.
showpos	在非负数值前面显示"＋".
skipws	当从一个流进行读取时，跳过空白字符(spaces, tabs, newlines).
unitbuf	在每次插入以后，清空缓冲区.
uppercase	以大写的形式显示科学记数法中的"e"和十六进制格式的"x".

也可以通过使用下面的操作符，不直接操作标志。大多数的编程人员都都熟悉 **endl** 操作符，它给我们一个使用操作符的启示。例如：当我们设置 *dec* 标志时，我们可以使用下面的命令：

```
cout << dec;
```

<iostream>中定义的操作符

操作符	描述	输入	输出
boolalpha	启用 boolalpha 标志	X	X
dec	启用 dec 标志	X	X
endl	输出换行标示，并清空缓冲区		X
ends	输出空字符		X
fixed	启用 fixed 标志		X
flush	清空流		X
hex	启用 hex 标志	X	X
internal	启用 internal 标志		X
left	启用 left 标志		X
noboolalpha	关闭 boolalpha 标志	X	X
noshowbase	关闭 showbase 标志		X
noshowpoint	关闭 showpoint 标志		X
noshowpos	关闭 showpos 标志		X
noskipws	关闭 skipws 标志	X	
nounitbuf	关闭 unitbuf 标志		X
nouppercase	关闭 uppercase 标志		X
oct	启用 oct 标志	X	X
right	启用 right 标志		X

scientific	启用 scientific 标志		X
showbase	启用 showbase 标志		X
showpoint	启用 showpoint 标志		X
showpos	启用 showpos 标志		X
skipws	启用 skipws 标志	X	
unitbuf	启用 unitbuf 标志		X
uppercase	启用 uppercase 标志		X
ws	跳过所有前导空白字符	X	

在<iomanip>中定义的操作符

操作符	描述	输入	输出
resetiosflags(long f)	关闭被指定为 *f* 的标志	X	X
setbase(int base)	设置数值的基本数为 *base*		X
setfill(int ch)	设置填充字符为 *ch*		X
setiosflags(long f)	启用指定为 *f* 的标志	X	X
setprecision(int p)	设置数值的精度		X
setw(int w)	设置域宽度为 *w*		X

2.4 Static Returns!

Watch out.

This function returns a variable that is statically located, and therefore overwritten each time this function is called. If you want to save the return value of this function, you should manually save it elsewhere.

Of course, when you save it elsewhere, you should make sure to actually copy the value(s) of this variable to another location. If the return value is a struct, you should make a new struct, then copy over the members of the struct.