标准 C++ 库参考手册

这些 C 语言库的头文件是 C++ 标准库的子集，涵盖了很多方面，包括通用工具库、输入/输出函数的宏和动态内存管理的函数。

目　录

FE_INVALID

FE_OVERFLOW

FE_UNDERFLOW

FE_ALL_EXCEPT

FE_DOWNWARD

FE_TONEAREST

FE_TOWARDZERO

FE_UPWARD

FE_DFL_ENV

FENV_ACCESS

(float.h)

tolower

toupper

容器

vector

致谢

当前文档 《标准 C++ 库参考手册》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-11-22。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：chrisniael https://github.com/chrisniael/cplusplus.com

文档地址：http://www.bookstack.cn/books/chrisniael-cpp-plus

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

参考手册

标准 C++ 库参考手册

C 库

这些 C 语言库的头文件是 C++ 标准库的子集，涵盖了很多方面，包括通用工具库、输入/输出函数的宏和动态内存管理的函数。

头文件	描述
<cassert> (assert.h)	C 诊断库 (头文件)
<cctype> (ctype.h)	字符处理函数 (头文件)
<cerrno> (errno.h)	C 错误 (头文件)
<cfenv> (fenv.h)	浮点环境 (头文件)
<cfloat> (float.h)	浮点类型特性 (头文件)
<cinttypes> (inttypes.h)	C 整数类型 (头文件)
<ciso646> (iso646.h)	ISO 646 可选操作符拼写 (头文件)
<climits> (limits.h)	整数类型的大小 (头文件)
<clocale> (locale.h)	C 本地化库 (头文件)
<cmath> (math.h)	C 数学库 (头文件)
<csetjmp> (setjmp.h)	非局部跳转 (头文件)
<csignal> (signal.h)	处理信号的 C 库 (头文件)
<cstdarg> (stdarg.h)	可变数量参数处理 (头文件)
<cstdbool> (stdbol.h)	布尔类型 (头文件)
<cstddef> (stddef.h)	C 标准定义 (头文件)
<cstdint> (stdint.h)	整数类型 (头文件)
<cstdio> (stdio.h)	操作输入/输出的 C 库 (头文件)
<cstdlib> (stdlib.h)	C 标准通用工具库 (头文件)
<cstring> (string.h)	C 字符串 (头文件)
<ctgmath> (tgmath.h)	类型泛化的数学 (头文件)
<ctime> (time.h)	C 时间库 (头文件)
<cuchar> (uchar.h)	Unicode 字符 (头文件)
<cwchar> (wchar.h)	宽字符 (头文件)
<cwctype> (wctype.h)	宽字符类型 (头文件)

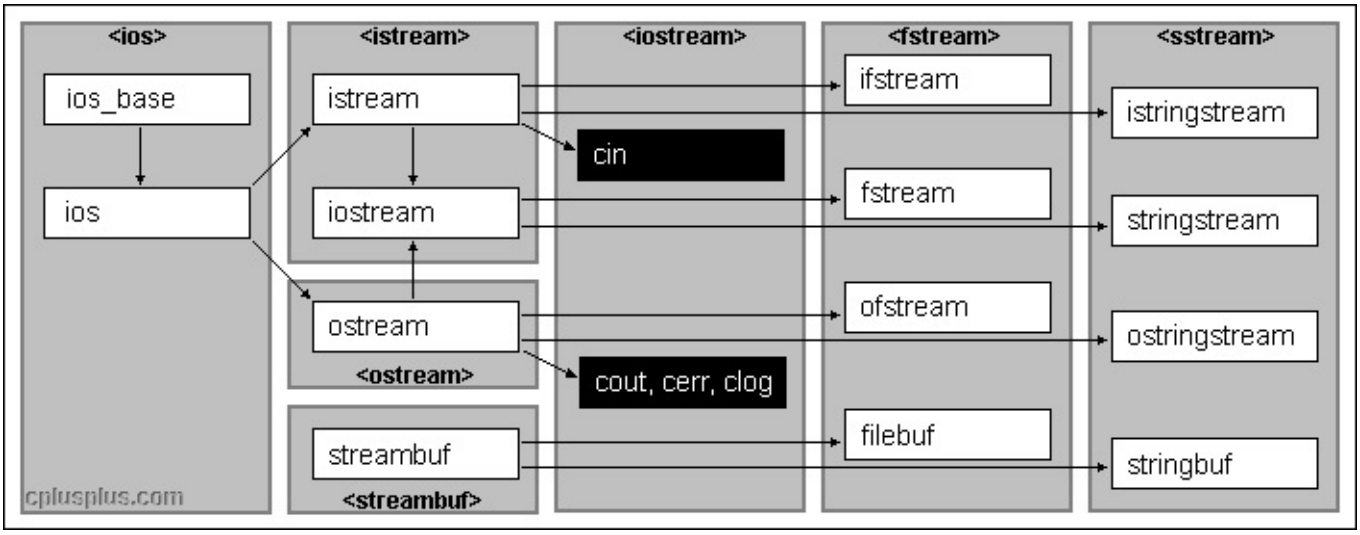
容器

头文件	描述
<array>	Array (头文件)
<bitset>	Bitset (头文件)
<deque>	Deque (头文件)
<forward_list>	Forward list (头文件)
<list>	List (头文件)
<map>	Map (头文件)
<queue>	Queue (头文件)
<set>	Set (头文件)
<stack>	Stack (头文件)
<unordered_map>	Unordered map (头文件)
<unordered_set>	Unordered set (头文件)
<vector>	Vector (头文件)

输入/输出流库

使用 流 这种抽象概念，来执行像文件和字符串这样的序列字符的输入输出操作。

在下面的关系图上，展示了这个功能涉及的多个相关联的类以及对应的头文件名字。



原子和线程库

头文件	描述
<atomic>	Atomic (头文件)
<condition_variable>	Condition variable (头文件)
<future>	Future (头文件)
<mutex>	Mutex (头文件)
<thread>	Thread (头文件)

其他头文件

头文件	描述
<algorithm>	标准模板库 ： 算法 (库)
<chrono>	时间库 (头文件)
<codecvt>	Unicode 转化方面 (头文件)
<complex>	复数库 (头文件)
<exception>	标准异常 (头文件)
<functional>	函数对象 (头文件)
<initializer_list>	初始化列表 (头文件)
<iterator>	迭代器定义 (头文件)
<limits>	数值范围 (头文件)
<locale>	本地化库 (头文件)
<memory>	内存元件 (头文件)
<new>	动态内存 (头文件)
<numeric>	泛型的数值操作 (头文件)
<random>	随机 (头文件)
<ratio>	比例头文件 (头文件)
<regex>	正则表达式 (头文件)
<stdexcept>	异常类 (头文件)
<string>	字符串 (头文件)
<system_error>	系统错误 (头文件)
<tuple>	Tuple 库 (头文件)
<typeindex>	类型索引 (头文件)
<typeinfo>	类型信息 (头文件)
<type_traits>	type_traits (头文件)
<utility>	工具组件 (头文件)
<valarray>	数值数组库 (头文件)

库

C 库

C 语言库

C++ 库被组织在 C 语言库相同结构的头文件中，并包括了相同的定义，但有以下的不同之处 ：

- 每个头文件的名字和 C 语言版本一样，但是多了 *“c“* 前缀。例如，C++ 头文件 *<cstdlib>* 等价于 C 语言头文件 *<stdlib.h>*。
- 库中所有元素都被定义在了 *std* 命名空间中了。

虽然这样，但为了兼容 C，传统头文件 *name.h*（比如 *stdlib.h* ）在全局作用域中同样提供了定义。这个手册中所有的例子就是使用这个版本，所以这些例子是完全与 C 兼容的，即使它在 C++ 中被废弃了。

在 C++ 的实现中当然也有某些特定的改变：

- *wchar_t*，*char16_t*，*char32_t* 和 *bool* 是 C++ 中的基本类型，因此，它们没有被定义在 C 语言中应该出现的头文件中。<iso646.h> 中的宏也一样，成了 C++ 中的关键字。
- 下面这些函数的参数常量性定义有所改变：*strchr*，*strpbrk*，*strrchr*，*strstr*，*memchr*。
- 头文件 <cstdlib> 中的函数 *atexit*，*exit* 和 *abort*，在 C++ 中增加了行为。
- 提供了一些重载版本的函数，使用额外的类型作为参数，但有相同的语义，例如，在头文件 <cmath> 中的的 *flot* 和 *long double* 版本的函数，*long* 版本的 *abs* 和 *div*。

注解版本

C++ 98 包括了 1990 ISO C 标准和它的修正案 #1 (ISO/IEC 9899:1990 和 ISO/IEC 9899:1990/DAM 1) 描述的 C 库。

C++ 11 包括了 1990 ISO C 标准和它的 Technical Corrigenda 1, 2, 3 (ISO/IEC 9899:1999 和 ISO/IEC 9899:1999/Cor.1,2,3) 描述的 C 库，加上 <cuchar> (ISO/IEC 19769:2004)。

头文件 C90 (C++98)

头文件	描述
<cassert> (assert.h)	C 诊断库
<cctype> (ctype.h)	字符处理函数 (头文件)
<cerrno> (errno.h)	C 错误 (头文件)
<cfenv> (fenv.h)	浮点环境 (头文件)
<cfloat> (float.h)	浮点类型特性 (头文件)
<cinttypes> (inttypes.h)	C 整数类型 (头文件)
<ciso646> (iso646.h)	ISO 646 可选操作符拼写 (头文件)
<climits> (limits.h)	整数类型的大小 (头文件)
<clocale> (locale.h)	C 本地化库 (头文件)

<cmath> (math.h)	C 数学库 (头文件)
<csetjmp> (setjmp.h)	非局部跳转 (头文件)
<csignal> (signal.h)	处理信号的 C 库 (头文件)
<cstdarg> (stdarg.h)	可变数量参数处理 (头文件)
<cstddef> (stddef.h)	C 标准定义 (头文件)
<cstdio> (stdio.h)	操作输入/输出的 C 库 (头文件)
<cstdlib> (stdlib.h)	C 标准通用工具库 (头文件)
<cstring> (string.h)	C 字符串 (头文件)
<ctime> (time.h)	C 时间库 (头文件)

ISO-C 90 修正案 1 添加了两个额外的头文件: <cwchar> 和 <cwctype>。

头文件 C99 (C++11)

头文件	描述
<cstdbool> (stdbol.h)	布尔类型 (头文件)
<cstdint> (stdint.h)	整数类型 (头文件)
<ctgmath> (tgmath.h)	类型泛化的数学 (头文件)
<cuchar> (uchar.h)	Unicode 字符 (头文件)
<cwchar> (wchar.h)	宽字符 (头文件)
<cwctype> (wctype.h)	宽字符类型 (头文件)

头文件

<cassert> (assert.h)

C诊断库

assert.h 定义了一个可以被用来作为标准调试工具的宏函数

宏函数

函数	描述
assert	评估断言 （宏）

宏

assert

<cassert>

```
void assert(int expression);
```

评估断言

如果这个函数形式的宏的参数表达式等于 0（例如 expression 等于 false），那么编译器会调用 `abort` 函数来终止程序，并将消息写入标准错误设备。

虽然消息内容依赖于特定的库实现，但是它至少包括：断言失败的 *expression* ，源文件的名字，和对应的行号。通常格式如下：

> Assertion failed: *expression*, file *filename*, line *line number*

参数

`expression`

expression 会被评估。如果这个 expression 等于 0，则会导致断言失败，并终止程序。

如果在包含 `<assert.h>` 的时候，一个名为 `NDEBUG` 的宏已经被定义，那么 `assert` 宏将被关闭。这个功能使的开发人在调试程序的时候，能在源代码中包含很多 `assert` 调用，而发布版本的时候能关闭所有 `assert` 宏，只需要在代码开始部分，并在包含 `<assert.h>` 前写上这样一行代码：

```
1. #define NDEBUG
```

因此， `assert` 的设计是用来捕捉程序错误的，而不是用户或者运行时错误，在程序退出调试阶段后，通常都会使用 `NDEBUG` 来关闭这个宏。

返回值

无

例子

```
1. /* assert example */
2. #include <stdio.h>    /* printf */
3. #include <assert.h> /* assert */
4.
5. void print_number(int *myInt)
6. {
7.     assert(myInt != NULL);
```

```
8.      printf("%d\n", *myInt);
9.  }
10.
11. int main()
12. {
13.     int a = 0;
14.     int * b = NULL;
15.     int * c = NULL;
16.
17.     b = &a;
18.
19.     print_number(b);
20.     print_number(c);
21.
22.     return 0;
23. }
```

在这个例子中，如果 `print_number` 使用一个空指针作为参数被调用，那么 `assert` 会终止程序执行。这种情况发生在第二次调用 `print_number` 时，它触发了一个断言失败，提示了bug的存在。

头文件

<cctype> (ctype.h)

字符处理函数

这个头文件定义了分类和转化字符的函数集

函数

这些函数把等价于一个字符的 *int* 型变量作为参数，并且返回一个 *int* 型值，这个返回值即可以作为一个字符，又可以代表一个布尔值：一个值为 *0* 的 *int* 型变量意味着 *false*，而非 *0* 的 *int* 型变量代表 *true* 。

这里有两个函数集：

字符分类函数

这些函数会检查作为参数传递进来的字符是否属于某一特定类别：

函数名	描述
isalnum	检查字符是否是字母或数字(alphanumeric) (函数)
isalpha	检查字符是否是字母(alphabetic) (函数)
isblank *(c++11)*	检查字符是否是空白符(blank) (函数)
iscntrl	检查字符是否是控制字符(control character) (函数)
isdigit	检查字符是否是十进制数字(dicimal digit) (函数)
isgraph	检查字符是否有图形表示(graphical representation) (函数)
islower	检查字符是否是小写字母(lowercase letter) (函数)
isprint	检查字符是否可打印(printable) (函数)
ispunct	检查字符是否是标点符号(punctuation) (函数)
isspace	检查字符是否是空格符(white-space) (函数)
isupper	检查字符是否是大写字母(uppercase letter) (函数)
isxdigit	检查字符是否是十六进制数字(hexadecimal) (函数)

字符转化函数

这是两个转化字母大小写的函数：

函数名	描述
tolower	将大写字母转化为小写 (函数)
toupper	将小写字母转化为大写 (函数)

对于第一个函数集，这里有一张各个函数将原始的127个ASCII字符集作为参数的返回值表 (表格中的 x 表明这个函

数将相应字符作为参数时返回 *true*)

ASCII 值	字符	iscntrl	isblank	isspace	isupper	islower	isa
0x00 .. 0x008	NUL,(其他控制码)	x					
0x09	tab('\t')	x	x	x			
0x0A .. 0x0D	(空格控制码 : '\f','\v','\n','\r')	x		x			
0x0E .. 0x1F	(其他控制码)	x					
0x20	空格(' ')		x	x			
0x21 .. 0x2F	!"#$%&'()*+,-./						
0x30 .. 0x39	0123456789						
0x3a .. 0x40	:;\<=>?@						
0x41 .. 0x46	ABCDEF				x		
0x47 .. 0x5A	GHIJKLMNOPQRSTUVWXYZ				x		
0x5B .. 0x60	[\]^_`						
0x61 .. 0x66	abcdef					x	
0x67 .. 0x7A	ghijklmnopqrstuvwxyz					x	
0x7B .. 0x7E	{\	}					
0x7F	(DEL)	x					

扩展字符集 (大于 0x7F) 可能会因为环境和平台的缘故而属于不同的种类。通常规则是，在大多数支持扩展字符集的平台下，标准 C 环境的 *isgraph* 和 *isprint* 函数返回 *true* 。

函数

isalnum

<cctype>

```
int isalnum ( int c );
```

检查字符是否是字母或数字(alphanumeric)

检查 *c* 是否是一个十进制数字或者是大写或小写字母。

函数返回值是 *true*，那么 isalpha 和 isdigit 也返回 *true*。

注意，判别一个字符是否是字母取决于使用环境。在默认的 “C” 环境中，只有当 isupper 和 islower 返回 *true* 的时候才是字母。

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 isalnum 在头文件 <locale>中。

参数

`c`

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个数字或字母，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 *false*)。

例子

```
1.  /* isalnum example */
2.  #include <stdio.h>
3.  #include <ctype.h>
4.
5.  int main()
6.  {
7.      int i;
8.      char str[] = "c3po...";
9.      i = 0;
10.     while(isalnum(str[i])) i++;
11.     printf("The first %d characters are alphanumeric.\n", i);
12.     return 0;
13. }
```

输出：

```
1. The first 4 characters are alphanumeric.
```

另请参阅

函数名	描述
isalpha	检查字符是否是字母(alphabetic) (函数)
isdigit	检查字符是否是十进制数字(dicimal digit) (函数)

函数

isalpha

<cctype>

```
int isalpha ( int c );
```

检查字符是否是字母(alphabetic)

检查 *c* 是否是一个字母。

注意，判别一个字符是否是字母取决于使用环境。在默认的 “C” 环境中，只有当 isupper 和 islower 返回 *true* 的时候才是字母。

使用其他的环境，只有当 isupper 或 islower 返回 *true* 时才是字母，其他还有一些被环境特定认为是字母的一些字符（在中情况下，这个字母字符不可能在函数 iscntrl，isdigit，ispunct 或 isspace 中返回 *true* 。

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 isalpha 在头文件 <locale>中。

参数

```
c
```

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个字母，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 *false*)。

例子

```
1.  /* isalpha example */
2.  #include <stdio.h>
3.  #include <ctype.h>
4.
5.  int main()
6.  {
7.      int i = 0;
8.      char str[] = "C++";
9.      while(str[i])
10.     {
11.         if(isalpha(str[i]))
12.             printf("character %c is alphabetic\n", str[i]);
13.         else
```

```
            printf("character %c is not alphabetic\n", str[i]);
        i++;
    }
    return 0;
}
```

输出：

```
character C is alphabetic
character + is not alphabetic
character + is not alphabetic
```

另请参阅

函数名	描述
isalnum	检查字符是否是字母或数字(alphanumeric) (函数)
isdigit	检查字符是否是十进制数字(dicimal digit) (函数)

函数

isblank (*C++11*)

<cctype>

```
int isblank ( int c );
```

检查字符是否是空白符(blank)

检查 *c* 是否是一个空白字符(blank character)。

标准 “C” 环境把水平制表符 (‘\t’) 和空格符 (‘ ‘) 认为是空白字符。

其他环境认定的空白符可能会不一样，但是它们必须是在函数 isspace 中返回 *true* 的空格字符。

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 isblank 在头文件 <locale>中。

参数

`c`

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个空白字符，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 false)。

例子

```
/* isblank example */
#include <stdio.h>
#include <ctype.h>

int main()
{
    char c;
    int i = 0;
    char str[] = "Example sentence to test is blank\n";
    while(str[i])
    {
        c = str[i];
        if(isblank(c))
            c = '\n';
        putchar(c);
        i++;
```

```
    }
    return 0;
}
```

这段代码把 C 字符串中所有的空白字符替换为换行字符，并追个字符的输出。

输出：

```
Example
sentence
to
test
isblank
```

另请参阅

函数名	描述
isspace	检查字符是否是空格符(white-space) (函数)
isgraph	检查字符是否可打印(graphical representation) (函数)
ispunct	检查字符是否是标点字符(punctuation) (函数)
isalnum	检查字符是否是字母或数字(alphanumeric) (函数)
isblank (locale)	使用环境检查字符是否是空白符(blank) (函数模板)

函数

iscntrl

<cctype>

```
int iscntrl ( int c );
```

检查字符是否是控制字符(control character)

检查 *c* 是否是一个控制字符。

控制字符并不占据显示的打印位置（这和在函数 isprint 中返回 *true* 的可打印字符相反）。

对于标准 ASCII 字符集（在 “C” 环境中），控制字符是 ASCII 值在 0x00 (NUL) 到 0x1f (US) 之间的，加上 0x7f (DEL) 的字符。

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 iscntrl 在头文件 <locale>中。

参数

`c`

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个控制字符，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 *false*)。

例子

```
1.  /* iscntrl example */
2.  #include <stdio.h>
3.  #include <ctype.h>
4.
5.  int main()
6.  {
7.      int i;
8.      char str[] = "first line \n second line \n";
9.      while(!iscntrl(str[i]))
10.     {
11.         putchar(str[i]);
12.         i++;
13.     }
14.     return 0;
15. }
```

这段代码追个字符输出一个字符串，直到遇到一个控制字符才跳出 while 循环。在这个例子中，只有第一行被输出，因为第一行以控制字符 '\n' (ASCII 值是 0x0a) 结尾。

另请参阅

函数名	描述
isgraph	检查字符是否有图形表示(graphical representation) (函数)
ispunct	检查字符是否是标点符号(punctuation) (函数)

函数

isdigit

<cctype>

```
int isdigit ( int c );
```

检查字符是否是十进制数字(decimal digit)

检查 *c* 是否是一个十进制数字。

十进制数字有：*0 1 2 3 4 5 6 7 8 9*

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 isdigit 在头文件 <locale>中。

参数

`c`

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个十进制数字，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 *false*)。

例子

```c
/* isdigit example */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main()
{
    char str[] = "1776ad";
    int year;
    if(isdigit(str[0]))
    {
        year = atoi(str);
        printf("The year that followed %d was %d.\n", year, year + 1);
    }
    return 0;
}
```

输出：

```
1. The year that followed 1776 was 1777.
```

isdigit 被用来检查 *str* 的第一个字符是否是一个十进制数字，来成为一个有效的候选者被 atoi 转化为一个整型的值。

另请参阅

函数名	描述
isalnum	检查字符是否是字母或数字(alphanumeric) (函数)
isalpha	检查字符是否是字母(alphabetic) (函数)

函数

isgraph

<cctype>

```
int isgraph ( int c );
```

检查字符是否有图形表示(graphical representation)

检查 *c* 是否是一个图形表示的字符

图形表示的字符是那些能被打印的字符 (isprint 决定)，除了空格字符 (‘ ‘)。

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 isgraph 在头文件 <locale>中。

参数

`c`

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个有图形表示的字符，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 *false*)。

例子

```
1.  /* isgraph example */
2.  #include <stdio.h>
3.  #include <ctype.h>
4.
5.  int main()
6.  {
7.      FILE * pFile;
8.      int c;
9.      pFile = fopen("myfile.txt", "r")
10.     if(pFile)
11.     {
12.         do
13.         {
14.             c = fgetc(pFile);
15.             if(isgraph(c))
16.                 putchar(c);
17.         }while(c != EOF);
18.         fclose(pFile);
```

```
    }
}
```

这个例子输出文件 “myfile.txt” 中除了空格字符和特殊字符外的内容，也就是说，只输出满足函数 isgraph 的字符。

另请参阅

函数名	描述
isprint	检查字符是否可打印 (函数)
isspace	检查字符是否是空格符(white-space) (函数)
isalnum	检查字符是否是字母或数字(alphanumeric) (函数)

函数

islower

<cctype>

```
int islower ( int c );
```

检查字符是否是小写字母 (lowercase letter)

检查 *c* 是否是一个小写字母。

注意，判别一个字符是否是小写字母取决于使用环境。在默认的 “C” 环境中，小写字母有: *a b c d e f g h i j k l m n o p q r s t u v w x y z*。

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 islower 在头文件 <locale>中。

参数

`c`

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个小写字母，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 *false*)。

例子

```
1.  /* islower example */
2.  #include <stdio.h>
3.  #include <ctype.h>
4.
5.  int main()
6.  {
7.      int i = 0;
8.      char str[] = "Test String.\n";
9.      char c;
10.     while(str[i])
11.     {
12.         c = str[i];
13.         if(islower(c))
14.             c = toupper(c);
15.         putchar(c);
16.         i++;
17.     }
```

```
18. }
```

输出：

```
1. TEST STRING
```

另请参阅

函数名	描述
isupper	检查字符是否是大写字母(isupper) (函数)
isalpha	检查字符是否是字母(alphabetic) (函数)
toupper	将小写字母转化为大写 (函数)
tolower	将大写字母转化为小写 (函数)

函数

isprint

<cctype>

```
int isprint ( int c );
```

检查字符是否可打印(printable)

检查 *c* 是否是一个可打印字符。

可打印字符会占据显示的打印位置（这和在函数 iscntrl 中返回 *true* 的控制字符相反）。

对于标准 ASCII 字符集（在 “C” 环境中），可打印字符是 ASCII 值大于 0x1f (US)，但除了 0x7f (DEL) 的字符。

isgraph 返回 *true* 的情况和 isprint 一样，除了空格字符 (‘ ‘) 外，空格字符 (‘ ‘) 在 isprint 中返回 *true*，但在 isgraph 中返回 *false*。

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 isprint 在头文件 <locale>中。

参数

`c`

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个数字或字母，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 *false*)。

例子

```
1.  /* isprint example */
2.  #include <stdio.h>
3.  #include <ctype.h>
4.
5.  int main()
6.  {
7.      int i = 0;
8.      char str[] = "first line \n second line \n";
9.      while(isprint(str[i]))
10.     {
11.         putchar(str[i]);
12.         i++;
```

```
    }
    return 0;
}
```

这段代码追个字符输出一个字符串，直到遇到一个控制字符才跳出 while 循环。在这个例子中，只有第一行被输出，因为第一行以控制字符 '\n' (ASCII 值是 0x0a) 结尾，它不是一个可打印字符。

另请参阅

函数名	描述
iscntrl	检查字符是否是控制字符(control character) (函数)
isspace	检查字符是否是空格符(white-space) (函数)
isalnum	检查字符是否是字母或数字(alphanumeric) (函数)

函数

ispunct

<cctype>

```
int ispunct ( int c );
```

检查字符是否是标点字符(punctuation)

检查 *c* 是否是一个标点字符。

标准 “C” 环境把所有是图形字符 (as in isgraph) 但不是字母或数字 (as in isalnum) 的字符认为是标点字符。

其他环境可能会把不同的字符当作标点字符，但无论哪种情况，它们肯定是 isgraph 但不是 isalnum。

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 ispunct 在头文件 <locale>中。

参数

`c`

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个数字或字母，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 *false*)。

例子

```
1.  /* ispunct example */
2.  #include <stdio.h>
3.  #include <ctype.h>
4.
5.  int main()
6.  {
7.      int i = 0;
8.      int cx = 0;
9.      char str[] = "Hello, welcome!";
10.     while(str[i])
11.     {
12.         if(ispunct(str[i]))
13.             cx++;
14.         i++;
15.     }
```

```
16.     printf("Sentence contains %d punctuation characters.\n", cx);
17. }
```

输出：

```
1. Sentence contains 2 punctuation characters.
```

另请参阅

函数名	描述
isgraph	检查字符是否有图形表示(graphical representation) (函数)
iscntrl	检查字符是否是控制字符(control character) (函数)

函数

isspace

<cctype>

```
int isspace ( int c );
```

检查字符是否是一个空格

检查 *c* 是否是一个空格字符。

标准 “C” 环境中，空白字符有：

字符	ASCII值	描述
‘ ’	(0x20)	空格(SPC)
‘\t’	(0x09)	水平制表符(TAB)
‘\n’	(0x0a)	换行符(LF)
‘\v’	(0x0b)	垂直制表符(VT)
‘\f’	(0x0c)	换页(FF)
‘\r’	(0x0d)	回车(CR)

在其他的环境中，可能会有不同的字符被认为是空格，但是它们不可能让函数 isalnum 返回 *true*。

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 isspace 在头文件 <locale>中。

参数

`c`

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个空格字符，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 *false*)。

例子

```
1. /* isspace example */
2. #include <stdio.h>
3. #include <ctype.h>
4.
5. int main()
```

```
{
    char c;
    int i = 0;
    char str[] = "Example sentence to test isspace\n";
    while(str[i])
    {
        c=str[i];
        if(isspace(c))
            c = '\n';
        putchar(c);
        i++;
    }
}
```

这段代码替换了 C 字符串中的空白字符为换行符，并追个字符的将其输出。

输出：

```
Example
sentence
to
test
isspace
```

另请参阅

函数名	描述
isgraph	检查字符是否有图形表示(graphical representation) (函数)
ispunct	检查字符是否是标点字符(punctuation) (函数)
isalnum	检查字符是否是字母或数字(alphanumeric) (函数)
isspace	检查字符是否是空格符(white-space) (函数)

函数

isupper

<cctype>

```
int isupper ( int c );
```

检查字符是否是大写字母 (uppercase letter)

检查 *c* 是否是一个大写字母。

注意，判别一个字符是否是大写字母取决于使用环境。在默认的 “C” 环境中，大写字母有: *A B C D E F G H I J K L M N O P Q R S T U V W X Y Z*。

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 isupper 在头文件 <locale>中。

参数

`c`

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个大写字母，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 *false*)。

例子

```
1.  /* isupper example */
2.  #include <stdio.h>
3.  #include <ctype.h>
4.
5.  int main()
6.  {
7.      int i = 0;
8.      char str[] = "Test String.\n";
9.      char c;
10.     while(str[i])
11.     {
12.         c = str[i];
13.         if(isupper(c))
14.             c = tolower(c);
15.         putchar(c);
16.         i++;
17.     }
```

```
18.     return 0;
19. }
```

输出：

```
1. test string.
```

另请参阅

函数名	描述
islower	检查字符是否是小写字母(islower) (函数)
isalpha	检查字符是否是字母(alphabetic) (函数)
toupper	将小写字母转化为大写 (函数)
tolower	将大写字母转化为小写 (函数)

函数

isxdigit

<cctype>

```
int isxdigit ( int c );
```

检查字符是否是十六进制数字(decimal digit)

检查 *c* 是否是一个十六进制数字字符。

十进制数字有: *0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F*

头文件 <cctype> 的参考中，有标准 ASCII 字符集的各个字符在不同 *ctype* 函数的返回值的详细图表。

在 C++ 中，这个函数的 locale-specific 模板版本 isxdigit 在头文件 <locale>中。

参数

`c`

被检查的字符，被转化为 *int* 型或 *EOF*。

返回值

如果 *c* 的确是一个十六进制数字，则返回一个非0值 (也就是 *true*)，否则返回0 (也就是 *false*)。

例子

```c
/* isxdigit example */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main()
{
    char str[] = "ffff";
    long int number;
    if(isxdigit(str[0]))
    {
        number = strtol(str, NULL, 16);
        printf("The hexadecimal number %lx is %ld.\n", number, number);
    }
    return 0;
}
```

输出：

```
1. The hexadecimal number ffff is 65535.
```

isxdigit 被用来检查 *str* 的第一个字符是否是一个十六进制数字，来成为一个有效的候选者被 strtol 转化为一个整型的值。

另请参阅

函数名	描述
isdigit	检查字符是否是十进制数字(decimal digit) (函数)
isalnum	检查字符是否是字母或数字(alphanumeric) (函数)
isalpha	检查字符是否是字母(alphabetic) (函数)

头文件

<cerrno> (errno.h)

C 错误

这个头文件定义了下面的宏：

宏	描述
errno	最后一个错误号（宏）

加上至少3个附加的常量宏：*EDOM*，*ERANGE* 和 *EILSEQ*（具体细节请查看 errno）。

宏

errno

`int`

最后一个错误号

这个宏被展开成为一个可改变的 *int* 型的左值。因此，它可以被程序读取和改变。

程序开始的时候，*errno* 被设置成为0，标准 C 库的任何函数可以把它改变成为任何非0的值，通常用来通知特定类型的错误（*errno* 一旦改变，则没有库函数会把它设置为0）。

程序同样可以改变 *errno* 的值。事实上，如果它的值是打算用来检查库函数调用后的错误的话，那么它应该在函数被调用前被设置为0（因为之前调用的任何函数都可能会改变它的值）。

声明 *errno* 的头文件 (<cerrno>) 中至少还定义了下面这些非0的常量宏:

宏	何时 *errno* 会被设置
EDOM	定义域错误 : 某些数学函数仅仅能被用于某些特定的值，这些值被称为定义域，例如平方根函数仅仅能被用于非负的数字，因此当 sqrt 使用负数作为参数调用时，会设置 *errno* 为 *EDOM* 。
ERANGE	值域错误 : 用变量表示的值的范围是有限的。例如，像 pow 这样的数学函数很容易超出浮点变量能表示的范围，还有像 strtod 这样的函数，可能会遇到数字序列长于值表示的范围。这些情况下，*errno* 被设置为 *ERANGE* 。
EILSEQ	非法序列 : 多字节字符序列可能会有一个有效序列的有限集。当一个多字节字符集被像 mbrtowc 这样的函数转化时，如果遇到无效序列，则 *errno* 会被设置为 *EILSEQ* 。

标准库的函数可能会设置 error 为任何值（不单单是上面列出的可移植的值）。某些特定的库实现可能会这个头文件中定义额外的名字。

C++ 11 通过包含很多可以在 POSIX 环境中获得的名字，扩展了必须定义在这个头文件中的基本的值集合，可移植的 errno 值的数量增加到 78 个。详细列表，请查看 errc。

与 errno 值相关的特定错误消息可以使用 strerror 获得，或使用 perror 直接打印。

在 C++ 中，errno 总是被定义成一个宏，但是在 C 中可能会使用外部链接实现成一个 *int* 对象。

数据竞争

支持多线程的库应该在每个线程基础上实现 *errno* : 每个线程拥有它自己的局部 *errno* 。依从 C11 和 C++ 11 标准，在库中这是一个必要条件。

头文件

<cfenv> (fenv.h) (C++11)

浮点环境

这个头文件声明了一系列函数和宏来访问浮点环境，以及特定类型。

这个浮点环境维持了一系列 状态标志 和特定的 控制模式 。浮点环境的特定内容依赖于实现，但 状态标志 通常包含了 浮点异常 和它们关联的信息， 控制模式 至少包含了舍入方向。

函数

浮点异常

函数名	描述
feclearexcept	清除浮点异常 (函数)
feraiseexcept	触发浮点异常 (函数)
fegetexceptflag	获得浮点异常标志 (函数)
fesetexceptflag	设置浮点异常标志 (函数)

舍入方向

函数	描述
fegetround	获得舍入方向模式 (函数)
fesetround	设置舍入方向模式 (函数)

整个环境

函数名	描述
fegetenv	获得浮点环境 (函数)
fesetenv	设置浮点环境 (函数)
feholdexcept	保留浮点环境 (函数)
feupdateenv	更新浮点环境 (函数)

其他

函数名	描述
fetestexcept	测试浮点环境异常 (函数)

类型

类型名	描述
fenv_t	浮点环境类型 （类型）
fexcept_t	浮点异常类型 （类型）

宏常量

浮点异常

宏名	描述
FE_DIVBYZERO	极异常 （宏）
FE_INEXACT	不精确的结果异常 （宏）
FE_INVALID	无效参数异常 （宏）
FE_OVERFLOW	向上溢出错误异常 （宏）
FE_UNDERFLOW	向下溢出错误异常 （宏）
FE_ALL_EXCEPT	所有异常 （宏）

舍入方向

宏名	描述
FE_DOWNWARD	向下舍入模式 （宏）
FE_TONEAREST	四舍五入模式 （宏）
FE_TOWARDZERO	朝零舍入模式 （宏）
FE_UPWARD	向上舍入模式 （宏）

整个环境

宏名	描述
FE_DFL_ENV	默认环境 （宏）

编译指示

编译指示名	描述
FENV_ACCESS	访问浮点环境 （编译指示）

函数

feclearexcept (*C++11*)

<cfenv>

```
int feclearexcept(int excepts);
```

清除浮点异常

试图清除 *excepts* 指定的浮点异常。

调用这个函数的程序需要确保在本次函数调用时，编译指示 FENV_ACCESS 已经开启。

参数

`excepts`

位掩码值：支持的任何浮点异常数字的组合（按位 OR）：

宏值	描述
FE_DIVBYZERO	极错误：被 0 除，或一些其他渐进无限的结果（从有限的参数）。
FE_INEXACT	不精确：结果不准确。
FE_INVALID	作用域错误：至少一个参数是函数没有定义的值。
FE_OVERFLOW	上溢错误：结果太大了，超出了返回值类型能表示的数量级。
FE_UNDERFLOW	下一错误：结果太小了，超出了返回值类型能表示的数量级。
FE_ALL_EXCEPT	所有异常（选择实现支持的所有异常）

特定的库实现可能会支持附加的 浮点异常 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

返回值

如果所有在 *excepts* 中的异常都被清除的话（或者 *excepts* 为 0 ），则返回 0 ，否则返回非 0 。

例子

```c
/* feclearexcept, fetestexcept example */
#include <stdio.h> /* printf */
#include <math.h> /* sqrt */
#include <fenv.h> /* feclearexcept, fetestexcept, FE_ALL_EXCEPT, FE_INVALID */
#pragma STDC FENV_ACCESS on

int main()
{
    feclearexcept(FE_ALL_EXCEPT);
    sqrt(-1);
    if(fetestexcept(FE_INVALID))
        printf("sqrt(-1) raises FE_INVALID\n");
    return 0;
}
```

可能的输出：

```
sqrt(-1) raises FE_INVALID
```

数据竞争

每个线程都保持着分离的、拥有自己状态的 浮点环境 。产生一个新线程就复制当前状态。[这个适用于 C11 和 C++11 的实现]

异常

不抛出异常的保证：这个函数从不抛出异常。
注意 C 浮点环境异常 不是 C++ 异常，因此不能被 *try/catch* 块捕捉。
调用这个函数的时候，如果编译指示 FENV_ACCESS 关闭的话，则会导致未定义行为。

另请参见

函数	描述
feraiseexcept	触发浮点异常 (函数)
fetestexcept	测试浮点异常 (函数)

函数

feraiseexcept (C++11)

<cfenv>

```
int feraiseexcept(int excepts);
```

触发浮点异常

尝试通过 *excepts* 触发浮点异常。

如果指定了多个异常，那么它们触发的顺序是不确定的。

调用这个函数的程序需要确保在本次函数调用时，编译指示 FENV_ACCESS 已经开启。

参数

`excepts`

位掩码值：支持的任何浮点异常数字的组合（按位 OR）：

宏值	描述
FE_DIVBYZERO	极错误：被 0 除，或一些其他渐进无限的结果（从有限的参数）。
FE_INEXACT	不精确：结果不准确。
FE_INVALID	作用域错误：至少一个参数是函数没有定义的值。
FE_OVERFLOW	上溢错误：结果太大了，超出了返回值类型能表示的数量级。
FE_UNDERFLOW	下一错误：结果太小了，超出了返回值类型能表示的数量级。
FE_ALL_EXCEPT	所有异常（选择实现支持的所有异常）

特定的库实现可能会支持附加的 浮点异常 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

返回值

如果所有在 *excepts* 中的异常都被成功触发的话（或者 *excepts* 为 0 ），则返回 0 ，否则返回非 0 。

例子

```c
/* feraiseexcept example */
#include <stdio.h>    /* printf */
#include <fenv.h>    /* feraiseexcept, fetestexcept, FE_ALL_EXCEPT, FE_INVALID */
#pragma STDC FENV_ACCESS on

double fn(double x)    /* some function for which zero is a domain error */
{
    if(x == 0.0)
        feraiseexcept(FE_INVALID);
    return x;
}

int main()
{
    feclearexcept(FE_ALL_EXCEPT);
    fn(0.0);
    if(fetestexcept(FE_INVALID))
        printf("FE_INVALID raised\n");
    return 0;
}
```

可能的输出：

```
FE_INVALID raised
```

数据竞争

每个线程都保持着分离的、拥有自己状态的 浮点环境 。产生一个新线程就复制当前状态。[这个适用于 C11 和 C++11 的实现]

异常

不抛出异常的保证：这个函数从不抛出异常。
注意 C 浮点环境异常 不是 C++ 异常，因此不能被 *try/catch* 块捕捉。
调用这个函数的时候，如果编译指示 FENV_ACCESS 关闭的话，则会导致未定义行为。

另请参见

函数	描述
feclearexcept	清楚浮点异常 (函数)
fetestexcept	测试浮点异常 (函数)

函数

fegetexceptflag (C++11)

```
int fegetexceptflag(fexcept_t *flagp, int excepts);
```

获取浮点异常标志

尝试把浮点异常 *excepts* 存储在 fexcept_t 对象 *flagp* 中。

参数

`flagp`

指明存储标志的 fexcept_t 对象。

`excepts`

位掩码值：支持的任何浮点异常数字的组合（按位 OR）：

宏值	描述
FE_DIVBYZERO	极错误：被 0 除，或一些其他渐进无限的结果（从有限的参数）。
FE_INEXACT	不精确：结果不准确。
FE_INVALID	作用域错误：至少一个参数是函数没有定义的值。
FE_OVERFLOW	上溢错误：结果太大了，超出了返回值类型能表示的数量级。
FE_UNDERFLOW	下一错误：结果太小了，超出了返回值类型能表示的数量级。
FE_ALL_EXCEPT	所有异常（选择实现支持的所有异常）

特定的库实现可能会支持附加的 浮点异常 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

返回值

如果标志被成功存储的话（或者 *excepts* 为 0 ），则返回 0 ，否则返回非 0 。

数据竞争

同时调用这个函数是安全的，不导致数据竞争。

异常

不抛出异常的保证：这个函数从不抛出异常。

另请参见

函数	描述
fesetexceptflag	设置浮点异常标志 (函数)
feholdexcept	保留浮点异常 (函数)

函数

fesetexceptflag (C++11)

```
int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

设置浮点异常标志

Attempts to set the exceptions indicated by excepts with the states stored in the object pointed by flagp.

如果成功，则函数会改变当前 浮点环境 的状态，设置请求的异常标志，但不会真正 触发 异常。

调用这个函数的程序需要确保在本次函数调用时，编译指示 FENV_ACCESS 已经开启。

参数

`flagp`

指向 fexcept_t 对象的指针，用来表示浮点异常。*flagp* 指向的对象应该在之前被函数 fegetexceptflag 通过参数 *excepts* 已经设置了值。

`excepts`

位掩码值：支持的任何浮点异常数字的组合（按位 OR）：

宏值	描述
FE_DIVBYZERO	极错误：被 0 除，或一些其他渐进无限的结果（从有限的参数）。
FE_INEXACT	不精确：结果不准确。
FE_INVALID	作用域错误：至少一个参数是函数没有定义的值。
FE_OVERFLOW	上溢错误：结果太大了，超出了返回值类型能表示的数量级。
FE_UNDERFLOW	下一错误：结果太小了，超出了返回值类型能表示的数量级。
FE_ALL_EXCEPT	所有异常（选择实现支持的所有异常）

特定的库实现可能会支持附加的 浮点异常 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

返回值

如果函数成功 set the flags in the (or if *excepts* was zero)，则返回 0 ，否则返回非 0 。

数据竞争

每个线程都保持着分离的、拥有自己状态的 浮点环境 。产生一个新线程就复制当前状态。[这个适用于 C11 和 C++11 的实现]

异常

不抛出异常的保证：这个函数从不抛出异常。
注意 C 浮点环境异常 不是 C++ 异常，因此不能被 *try/catch* 块捕捉。
调用这个函数的时候，如果编译指示 FENV_ACCESS 关闭的话，则会导致未定义行为。

另请参见

函数	描述
fegetexceptflag	获取浮点异常标志 (函数)
feraiseexcept	触发浮点异常 (函数)

函数

fegetround (C++11)

```
int fegetround(void);
```

获取舍入方向模式

返回当前 浮点环境 中表明舍入方向模式的值。

这个函数的返回值不一定和 cfloat 中 *FLT_ROUNDS* 的值相同。

参数

无

返回值

如果这个函数能决定当前舍入模式，并且被当前实现支持，那么函数返回值对应的宏定义如下：

宏值	描述
FE_DOWNWARD	向下舍入模式（宏）
FE_TONEAREST	四舍五入模式（宏）
FE_TOWARDZERO	朝零舍入模式（宏）
FE_UPWARD	向上舍入模式（宏）

特定的库实现可能会支持附加的 浮点舍入方向 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

例子

```
1. /* fegetround / rint example */
2. #include <stdio.h>    /* printf */
3. #include <fenv.h>    /* fegetround FE_* */
4. #include <math.h>    /* rint */
5.
6. int main()
7. {
```

```
    printf("Rounding using ");
    switch(fegetround())
    {
        case FE_DOWNWARD:
            printf("downward");
            break;
        case FE_TONEAREST:
            printf("to-nearset");
            break;
        case FE_TOWARDZERO:
            printf("toward-zero");
            break;
        case FE_UPWARD:
            printf("upward");
            break;
        default:
            printf("unknown");
    }
    printf(" rounding:\n");

    printf("rint (2.3) = %.1f\n", rint(2.3));
    printf("rint (3.8) = %.1f\n", rint(3.8));
    printf("rint (-2.3) = %.1f\n", rint(-2.3));
    printf("rint (-3.8) = %.1f\n", rint(-3.8));

    return 0;
}
```

可能的输出：

```
Rounding using to-nearset rounding:
rint (2.3) = 2.0
rint (3.8) = 4.0
rint (-2.3) = -2.0
rint (-3.8) = -4.0
```

数据竞争

每个线程都保持着分离的、拥有自己状态的 浮点环境 。产生一个新线程就复制当前状态。[这个适用于 C11 和 C++11 的实现]

异常

不抛出异常的保证：这个函数从不抛出异常。

另请参见

函数	描述

fesetround	设置舍入方向模式（函数）
fegetenv	获取浮点环境（函数）
rint	舍入至整数值（函数）

函数

fesetround (C++11)

```
int fesetround(int rdir);
```

设置舍入方向模式

设置 *rdir* 为 当前 浮点环境 的 舍入方向。

这个函数的返回值不一定和 cfloat 中 *FLT_ROUNDS* 的值相同。

参数

`rdir`

以下定义为 舍入方向模式 的值之一：

宏值	描述
FE_DOWNWARD	向下舍入模式（宏）
FE_TONEAREST	四舍五入模式（宏）
FE_TOWARDZERO	朝零舍入模式（宏）
FE_UPWARD	向上舍入模式（宏）

特定的库实现可能会支持附加的 浮点舍入方向 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

返回值

如果请求的浮点方向被成功设置的话，则返回 0 ，否则返回非 0 。

例子

```
1. /* fesetround example */
2. #include <stdio.h>    /* printf */
3. #include <fenv.h>    /* fesetround, FE_* */
4. #include <math.h>    /* rint */
5. #pragma STDC FENV_ACCESS on
```

```c

int main()
{
    printf("rounding -3.8:\n");

    fesetround(FE_DOWNWARD);
    printf("FE_DOWNWARD: %.1f\n", rint(-3.8));

    fesetround(FE_TONEAREST);
    printf("FE_TONEAREST: %.1f\n", rint(-3.8));

    fesetround("FE_TOWARDZERO: %.1f\n", rint(-3.8));
    printf("FE_TOWARDZERO: %.1f\n", rint(-3.8));

    fesetround(FE_UPWARD);
    printf("FE_UPWARD: %.1f\n", rint(-3.8));

    return 0;
}
```

可能的输出：

```
rounding -3.8:
FE_DOWNWARD: -4.0
FE_TONEAREST: -4.0
FE_TOWARDZERO: -3.0
FE_UPWARD: -3.0
```

数据竞争

同时调用这个函数是安全的，不导致数据竞争。

异常

不抛出异常的保证：这个函数从不抛出异常。

另请参见

函数	描述
fegetround	获取浮点方向模式 (函数)
fesetenv	设置浮点环境 (函数)
rint	舍入至整数值 (函数)

函数

fegetenv (C++)

```
int fegetenv(fenv_t *envp);
```

获取浮点环境

尝试将当前 浮点环境 的状态存储在 *envp* 指向的对象中。

浮点环境 是影响 浮点计算（包括 浮点异常 和 舍入方向模式_ ）的状态标志和控制模式的集合。

调用这个函数的程序需要确保在本次函数调用时，编译指示 FENV_ACCESS 已经开启。

参数

`envp`

指向存储浮点环境状态的 fenv_t 对象。

返回值

如果状态被成功存储，则返回0，否则返回非0。

数据竞争

每个线程都保持着分离的、拥有自己状态的 浮点环境 。产生一个新线程就复制当前状态。[这个适用于 C11 和 C++11 的实现]

异常

不抛出异常的保证：这个函数从不抛出异常。

另请参见

函数	描述
feholdexcep	保留浮点环境 (函数)
fesetenv	设置浮点环境 (函数)

函数

fesetenv (C++11)

```
int fesetenv(const fenv_t *envp);
```

设置浮点环境

尝试用 *envp* 指向的对象建立 浮点环境 的状态。

浮点环境 是影响 浮点计算（包括 浮点异常 和 舍入方向模式_ ）的状态标志和控制模式的集合。

如果成功的话，这个函数会改变浮点环境的当前状态，但不会真的 触发 状态中的异常。

调用这个函数的程序需要确保在本次函数调用时，编译指示 FENV_ACCESS 已经开启。

参数

`envp`

要么是指向 fenv_t 对象的指针，要么是 浮点环境 的宏值之一：

宏名	描述
FE_DFL_ENV	默认的浮点环境（和程序启动时一样）

特定的库实现可能会支持附加的 浮点环境 状态值（它们对应的宏同样以 FE_ 开头的宏）。

返回值

如果状态被成功建立，则返回0，否则返回非0。

数据竞争

每个线程都保持着分离的、拥有自己状态的 浮点环境 。产生一个新线程就复制当前状态。[这个适用于 C11 和 C++11 的实现]

异常

不抛出异常的保证：这个函数从不抛出异常。

另请参见

函数	描述
feupdateenv	更新浮点环境 (函数)

fegetenv	获取浮点环境（函数）
fesetenv	设置浮点环境（函数）

函数

feholdexcept (C++11)

```
int feholdexcept(fenv_t *envp);
```

保留浮点异常

保存 浮点环境 的当前状态至 *envp* 指向的对象中。然后会重置当前状态，并且如果支持的话会设置环境为 *non-stop* 模式。

调用这个函数的程序需要确保在本次函数调用时，编译指示 FENV_ACCESS 已经开启。

参数

`envp`

指向存储浮点环境状态的 fenv_t 对象的指针。

返回值

如果函数成功执行的话，包括设置 浮点环境 为 *non-stop* 模式，则返回 0， 否则返回非 0。

例子

```
/* feholdexcept / feupdateenv example */
#include <stdio.h>    /* printf, puts */
#include <fenv.h>    /* feholdexcept, feclearexcept, fetestexcept, feupdateenv, FE_* */
#include <math.h>    /* log */
#pragma STDC FENV_ACCESS on

double log_zerook(double x)
{
    fenv_t fe;
    feholdexcept(&fe);
    x = log(x);
    feclearexcept(FE_OVERFLOW | FE_DIVBYZERO);
    feupdateenv(&fe);
    return x;
}

int main()
{
    feclearexcept(FE_ALL_EXCEPT);
    printf("log(0.0): %f\n", log_zerook(0.0));
    if(!fetestexcept(FE_ALL_EXCEPT));
        puts("no exceptions raised");

```

```
    return 0;
}
```

可能的输出：

```
log(0.0): -inf
no exceptions raised
```

数据竞争

每个线程都保持着分离的、拥有自己状态的 浮点环境 。产生一个新线程就复制当前状态。[这个适用于 C11 和 C++11 的实现]

异常

不抛出异常的保证：这个函数从不抛出异常。
注意 C 浮点环境异常 不是 C++ 异常，因此不能被 *try/catch* 块捕捉。

另请参见

函数	描述
fegetenv	获取浮点环境 (函数)
fesetenv	设置浮点环境 (函数)
feclearexcept	清除浮点异常 (函数)

函数

feupdateenv (C++11)

```
int feupdateenv(const fenv_t *envp);
```

更新浮点环境

尝试用 *envp* 指向的对象建立 浮点环境 的状态。然后它会尝试触发在函数调用前设置在 浮点环境 中的异常。

调用这个函数的程序需要确保在本次函数调用时，编译指示 FENV_ACCESS 已经开启。

参数

要么是指向 fenv_t 对象的指针，要么是 浮点环境 的宏值之一：

宏名	描述
FE_DFL_ENV	默认的浮点环境（和程序启动时一样）

特定的库实现可能会支持附加的 浮点环境 状态值（它们对应的宏同样以 FE_ 开头的宏）。

返回值

如果函数成功，则返回 0， 否则返回非 0。

例子

```
/* feholdexcept / feupdateenv example */
#include <stdio.h>    /* printf, puts */
#include <fenv.h>    /* feholdexcept, feclearexcept, fetestexcept, feupdateenv, FE_* */
#include <math.h>    /* log */
#pragma STDC FENV_ACCESS on

double log_zerook(double x)
{
    fenv_t fe;
    feholdexcept(&fe);
    x = log(x);
    feclearexcept(FE_OVERFLOW | FE_DIVBYZERO);
    feupdateenv(&fe);
    return x;
}

int main()
{
    feclearexcept(FE_ALL_EXCEPT);
    printf("log(0.0): %f\n", log_zerook(0.0));
```

```
    if(!fetestexcept(FE_ALL_EXCEPT));
        puts("no exceptions raised");

    return 0;
}
```

可能的输出：

```
log(0.0): -inf
no exceptions raised
```

数据竞争

每个线程都保持着分离的、拥有自己状态的 浮点环境 。产生一个新线程就复制当前状态。[这个适用于 C11 和 C++11 的实现]

异常

不抛出异常的保证：这个函数从不抛出异常。
注意 C 浮点环境异常 不是 C++ 异常，因此不能被 *try/catch* 块捕捉。

另请参见

函数	描述
feupdateenv	更新浮点环境 (函数)
fegetenv	获取浮点环境 (函数)
fesetenv	设置浮点环境 (函数)

函数

fetestexcept (C++11)

```
int fetestexcept(int excepts);
```

测试浮点异常

返回在 *excepts* 中当前设置的异常。

返回值是用按位或表示的 *excepts* 中被当前 浮点环境 设置的异常的集合。如果 *excepts* 中的异常一个没有被设置，则返回 0。

调用这个函数的程序需要确保在本次函数调用时，编译指示 FENV_ACCESS 已经开启。

参数

`excepts`

位掩码值：支持的任何浮点异常数字的组合（按位 OR）：

宏值	描述
FE_DIVBYZERO	极错误：被 0 除，或一些其他渐进无限的结果（从有限的参数）。
FE_INEXACT	不精确：结果不准确。
FE_INVALID	作用域错误：至少一个参数是函数没有定义的值。
FE_OVERFLOW	上溢错误：结果太大了，超出了返回值类型能表示的数量级。
FE_UNDERFLOW	下一错误：结果太小了，超出了返回值类型能表示的数量级。
FE_ALL_EXCEPT	所有异常（选择实现支持的所有异常）

特定的库实现可能会支持附加的 浮点异常 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

返回值

如果 *excepts* 中的异常没有一个被设置的话，则返回 0，否则返回当前设置的异常（在 *excepts* 中的）。

例子

```c
/* fetestexcept example */
#include <stdio.h>    /* puts */
#include <fenv.h>    /* feraiseexcept, fetestexcept, FE_* */
#pragma STDC FENV_ACCESS on

double fn(double x)
{
    /* some function for which zero is a domain and range error */
    if(x == 0.0)
        feraiseexcept(FE_INVALID | FE_OVERFLOW);
    return x;
}

int main()
{
    int fe;

    feclearexcept(FE_ALL_EXCEPT);
    fn(0.0);

    /* testing for single exception: */
    if(fetestexcept(FE_OVERFLOW))
        puts("FE_OVERFLOW is set");

    /* testing multiple exceptions: */
    fe = fetestexcept(FE_ALL_EXCEPT);

    puts("The following exceptions are set:");
    if(fe & FE_DIVBYZERO)
        puts("FE_DIVBYZERO");
    if(fe & FE_INEXACT)
        puts("FE_INEXACT");
    if(fe &FE_INVALID)
        puts("FE_INVALID");
    if(fe & FE_OVERFLOW)
        puts("FE_OVERFLOW");
    if(fe & FE_UNDERFLOW)
        puts("FE_UNDERFLOW");

    return 0;
}
```

可能的输出：

```
FE_OVERFLOW is set
The following exceptions are set:
FE_INVALID
FE_OVERFLOW
```

数据竞争

每个线程都保持着分离的、拥有自己状态的 浮点环境 。产生一个新线程就复制当前状态。[这个适用于 C11 和 C++11 的实现]

异常

不抛出异常的保证：这个函数从不抛出异常。
注意 C 浮点环境异常 不是 C++ 异常，因此不能被 *try/catch* 块捕捉。
调用这个函数的时候，如果编译指示 FENV_ACCESS 关闭的话，则会导致未定义行为。

另请参见

函数	描述
feraiseexcept	触发浮点异常 (函数)
feclearexcept	清楚浮点异常 (函数)
feholdexcept	保留浮点异常 (函数)

类型

fenv_t (C++11)

浮点环境类型

表示整个 浮点环境 状态的类型，包括它的 状态标志（例如有效的 浮点异常 ）和 控制模式 （例如 舍入方向 ）。

这个类型的具体细节取决与库实现：它的值会被 fegetenv 或 feholdexcept 设置，还可以被应用于 fesetenv 或 feupdateenv 的调用。

另请参阅

函数/类型	描述
fegetenv	获取浮点环境 (函数)
fesetenv	设置浮点环境 (函数)
fexcept_t	浮点异常类型 (类型)

类型

fexcept_t (C++11)

浮点异常类型

可以表示所有 浮点状态标志 的类型，包括有效的 浮点异常 和任何实现相关状态的附加信息。

这个类型的具体细节取决与库实现：它的值会被 `fegetexceptflag` 设置，还可以被应用于 `fesetexceptflag` 的调用。

另请参阅

函数/类型	描述
`fegetenv`	获取浮点环境 (函数)
`fesetenv`	设置浮点环境 (函数)
`fenv_t`	浮点环境类型 (类型)

宏

FE_DIVBYZERO (C++11)

```
int
```

极错误

这个宏展开成一个 *int* 型的值，用来表示触发 极错误 时的 浮点异常 。

极错误 出现在操作结果渐进无限时，例如，被 0 除，或者 log(0.0)。

它被定义为 2 的整数次方，允许和多个 浮点异常 组合（使用按位 OR 操作：| ）成为单个值：

宏值	描述
FE_DIVBYZERO	极错误：被 0 除，或一些其他渐进无限的结果（从有限的参数）。
FE_INEXACT	不精确：结果不准确。
FE_INVALID	作用域错误：至少一个参数是函数没有定义的值。
FE_OVERFLOW	上溢错误：结果太大了，超出了返回值类型能表示的数量级。
FE_UNDERFLOW	下一错误：结果太小了，超出了返回值类型能表示的数量级。
FE_ALL_EXCEPT	所有异常（选择实现支持的所有异常）

特定的库实现可能会支持附加的 浮点异常 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。
FE_DIVBYZERO 总是被定义，如果 math_errhandling 有 *MATH_ERREXCEPT* 集合。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

另请参见

宏名	描述
FE_INEXACT	不精确的结果异常 (宏)
FE_INVALID	无效参数异常 (宏)
FE_OVERFLOW	向上溢出错误异常 (宏)
FE_UNDERFLOW	向下溢出错误异常 (宏)
FE_ALL_EXCEPT	所有异常 (宏)

宏

FE_INEXACT (C++11)

```
int
```

不精确的异常

这个宏展开成一个 *int* 型的值，用来表示触发 不精确的结果 时的 浮点异常 。

不精确异常 被触发用来提醒操作的返回类型不能表示准确的结果，或者当函数因为某些其他原因不能产生一个精确的结果。

它被定义为 2 的整数次方，允许和多个 浮点异常 组合（使用按位 OR 操作: | ）成为单个值:

宏值	描述
FE_DIVBYZERO	极错误: 被 0 除，或一些其他渐进无限的结果（从有限的参数）。
FE_INEXACT	不精确: 结果不准确。
FE_INVALID	作用域错误: 至少一个参数是函数没有定义的值。
FE_OVERFLOW	上溢错误: 结果太大了，超出了返回值类型能表示的数量级。
FE_UNDERFLOW	下一错误: 结果太小了，超出了返回值类型能表示的数量级。
FE_ALL_EXCEPT	所有异常（选择实现支持的所有异常）

特定的库实现可能会支持附加的 浮点异常 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。
FE_INEXACT 总是被定义，如果 math_errhandling 有 *MATH_ERREXCEPT* 集合。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

另请参见

宏名	描述
FE_DIVBYZERO	极异常 (宏)
FE_INVALID	无效参数异常 (宏)
FE_OVERFLOW	向上溢出错误异常 (宏)
FE_UNDERFLOW	向下溢出错误异常 (宏)
FE_ALL_EXCEPT	所有异常 (宏)

宏

FE_INVALID (C++11)

`int`

无效参数异常

这个宏展开成一个 *int* 型的值，用来表示触发 无效参数 时的 浮点异常 。

无效参数异常 被触发用来提醒 传递给函数的参数超出了它的定义域（也就是，函数不是为那个值而定义的），比如 `sqrt(-1.0)`。

触发这个异常的函数的返回值是不明确的。

FE_INVALID 被定义为 2 的整数次方，允许和多个 浮点异常 组合（使用按位 OR 操作：| ）成为单个值：

宏值	描述
FE_DIVBYZERO	极错误：被 0 除，或一些其他渐进无限的结果（从有限的参数）。
FE_INEXACT	不精确：结果不准确。
FE_INVALID	作用域错误：至少一个参数是函数没有定义的值。
FE_OVERFLOW	上溢错误：结果太大了，超出了返回值类型能表示的数量级。
FE_UNDERFLOW	下一错误：结果太小了，超出了返回值类型能表示的数量级。
FE_ALL_EXCEPT	所有异常（选择实现支持的所有异常）

特定的库实现可能会支持附加的 浮点异常 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。
*FE_INVALID*总是被定义，如果 math_errhandling 有 *MATH_ERREXCEPT* 集合。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

另请参见

宏名	描述
FE_DIVBYZERO	极异常 (宏)
FE_INEXACT	不精确的结果异常 (宏)
FE_OVERFLOW	向上溢出错误异常 (宏)
FE_UNDERFLOW	向下溢出错误异常 (宏)
FE_ALL_EXCEPT	所有异常 (宏)

宏

FE_OVERFLOW (C++11)

```
int
```

上溢错误异常

这个宏展开成一个 *int* 型的值，用来表示触发 上溢错误 时的 浮点异常 。

上溢错误 出现在因为操作结果的数量级太大（符号为正或负）而不能被返回值类型表示的时候。

上溢的操作返回一个正或负的 HUGE_VAL （或 HUGE_VALF），或 HUGE_VALL，并且会影响默认的 舍入模式 。

FE_OVERFLOW 被定义为 2 的整数次方，允许和多个 浮点异常 组合（使用按位 OR 操作：| ）成为单个值：

宏值	描述
FE_DIVBYZERO	极错误：被 0 除，或一些其他渐进无限的结果（从有限的参数）。
FE_INEXACT	不精确：结果不准确。
FE_INVALID	作用域错误：至少一个参数是函数没有定义的值。
FE_OVERFLOW	上溢错误：结果太大了，超出了返回值类型能表示的数量级。
FE_UNDERFLOW	下一错误：结果太小了，超出了返回值类型能表示的数量级。
FE_ALL_EXCEPT	所有异常（选择实现支持的所有异常）

特定的库实现可能会支持附加的 浮点异常 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。
FE_OVERFLOW 总是被定义，如果 math_errhandling 有 *MATH_ERREXCEPT* 集合。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

另请参见

宏名	描述
FE_DIVBYZERO	极异常 (宏)
FE_INEXACT	不精确的结果异常 (宏)
FE_INVALID	无效参数异常 (宏)
FE_UNDERFLOW	向下溢出错误异常 (宏)
FE_ALL_EXCEPT	所有异常 (宏)

宏

FE_UNDERFLOW (C++11)

```
int
```

下溢错误异常

这个宏展开成一个 *int* 型的值，用来表示触发 下溢错误 时的 浮点异常 。

下溢错误 出现在因为操作结果的数量级太小（符号为正或负）而不能被返回值类型表示的时候。

下溢操作返回一个数量级不大于最小常规化正数的未确定的值。

操作是否触发这个异常是实现定义的：没有操作必须需要出发这个异常，但实现可以选择这么做。

FE_UNDERFLOW 被定义为 2 的整数次方，允许和多个 浮点异常 组合（使用按位 OR 操作：| ）成为单个值：

宏值	描述
FE_DIVBYZERO	极错误：被 0 除，或一些其他渐进无限的结果（从有限的参数）。
FE_INEXACT	不精确：结果不准确。
FE_INVALID	作用域错误：至少一个参数是函数没有定义的值。
FE_OVERFLOW	上溢错误：结果太大了，超出了返回值类型能表示的数量级。
FE_UNDERFLOW	下一错误：结果太小了，超出了返回值类型能表示的数量级。
FE_ALL_EXCEPT	所有异常（选择实现支持的所有异常）

特定的库实现可能会支持附加的 浮点异常 值（它们对应的宏同样以 FE_ 开头的宏）。

C99

库可能定义在 <fenv.h>，仅仅支持上面这些宏值（其他可能没有被定义）。

C++11

至少上面所有的宏值都定义在 <fenv.h> 中（即使实现不支持）。

另请参见

宏名	描述
FE_DIVBYZERO	极异常 (宏)
FE_INEXACT	不精确的结果异常 (宏)
FE_INVALID	无效参数异常 (宏)
FE_OVERFLOW	向上溢出错误异常 (宏)
FE_ALL_EXCEPT	所有异常 (宏)

宏

FE_ALL_EXCEPT （C++11）

`int`

所有异常

这个宏展开成一个 *int* 型的值，它组合了所有定义在 <cfenv> 中的 浮点异常 值（用按位 OR ）。

如果实现不支持 浮点异常 ，那么这个宏被定义为 0 。

它可以被用于哪些期望用 浮点异常 的位掩码作为参数的函数：
feclearexcept，fegetexceptflag，feraiseexcept，fesetexceptflag，或者 fetestexcept。

C99

它是所有实现的 浮点异常 宏值的组合，可能包括下面这些（加上其他特定实现的异常）：

C++11

它是所有实现的 浮点异常 宏值的组合，包括下面这些（加上其他特定实现的异常）：

宏值	描述
FE_DIVBYZERO	极错误：被 0 除，或一些其他渐进无限的结果（从有限的参数）。
FE_INEXACT	不精确：结果不准确。
FE_INVALID	作用域错误：至少一个参数是函数没有定义的值。
FE_OVERFLOW	上溢错误：结果太大了，超出了返回值类型能表示的数量级。
FE_UNDERFLOW	下一错误：结果太小了，超出了返回值类型能表示的数量级。
FE_ALL_EXCEPT	所有异常（选择实现支持的所有异常）

另请参见

宏名	描述
FE_DIVBYZERO	极异常 （宏）
FE_INEXACT	不精确的结果异常 （宏）
FE_INVALID	无效参数异常 （宏）
FE_OVERFLOW	向上溢出错误异常 （宏）
FE_UNDERFLOW	向下溢出错误异常 （宏）
feraiseexcept	触发浮点异常 （函数）

宏

FE_DOWNWARD (C++11)

这个宏展开成一个 *int* 型值，来为函数 fegetround 和 fesetround 表示 向下舍入方向模式 。

向下舍入 x 就是选择不大于 x 的最大的值。

可能的 舍入方向模式 是：

宏值	描述
FE_DOWNWARD	向下舍入
FE_TONEAREST	四舍五入
FE_TOWARDZERO	向零舍入
FE_UPWARD	向上舍入

另请参阅

宏/函数	描述
FE_TONEAREST	四舍五入模式 (宏)
FE_TOWARDZERO	朝零舍入模式 (宏)
FE_UPWARD	向上舍入模式 (宏)
fegetround	获得舍入方向模式 (函数)
fesetround	设置舍入方向模式 (函数)

宏

FE_TONEAREST (C++11)

这个宏展开成一个 *int* 型值，来为函数 fegetround 和 fesetround 表示 四舍五入方向模式 。

四舍五入 x 就是尽可能选择接近 x 的值，with halfway cases rounded away from zero.

可能的 舍入方向模式 是：

宏值	描述
FE_DOWNWARD	向下舍入
FE_TONEAREST	四舍五入
FE_TOWARDZERO	向零舍入
FE_UPWARD	向上舍入

另请参阅

宏/函数	描述
FE_DOWNWARD	向下舍入模式 (宏)
FE_TOWARDZERO	朝零舍入模式 (宏)
FE_UPWARD	向上舍入模式 (宏)
fegetround	获得舍入方向模式 (函数)
fesetround	设置舍入方向模式 (函数)

宏

FE_TOWARDZERO (C++11)

这个宏展开成一个 *int* 型值，来为函数 fegetround 和 fesetround 表示 向零舍入方向模式 。

向零舍入 x 就是尽可能选择数量级不大于 x ，但最接近它的值。

可能的 舍入方向模式 是：

宏值	描述
FE_DOWNWARD	向下舍入
FE_TONEAREST	四舍五入
FE_TOWARDZERO	向零舍入
FE_UPWARD	向上舍入

另请参阅

宏/函数	描述
FE_DOWNWARD	向下舍入模式 (宏)
FE_TONEAREST	四舍五入模式 (宏)
FE_UPWARD	向上舍入模式 (宏)
fegetround	获得舍入方向模式 (函数)
fesetround	设置舍入方向模式 (函数)

宏

FE_UPWARD (C++11)

这个宏展开成一个 *int* 型值，来为函数 fegetround 和 fesetround 表示 向上舍入方向模式 。

向上舍入 x 就是尽可能选择不小于 x 的最小的值。

可能的 舍入方向模式 是：

宏值	描述
FE_DOWNWARD	向下舍入
FE_TONEAREST	四舍五入
FE_TOWARDZERO	向零舍入
FE_UPWARD	向上舍入

另请参阅

宏/函数	描述
FE_DOWNWARD	向下舍入模式 (宏)
FE_TONEAREST	四舍五入模式 (宏)
FE_TOWARDZERO	朝零舍入模式 (宏)
fegetround	获得舍入方向模式 (函数)
fesetround	设置舍入方向模式 (函数)

宏

FE_DFL_ENV (C++11)

`fenv_t *`

默认环境

这个宏展开成一个指向 `fenv_t` 对象的指针，这个对象是用来为函数 `fesetenv` 和 `feupdateenv` 选择 默认环境 的。

默认环境 是程序刚启动时的 浮点环境 的状态。

另请参见

函数/类型	描述
`fesetenv`	设置浮点环境 (函数)
`feupdateenv`	更新浮点环境 (函数)
`fenv_t`	浮点环境类型 (类型)

编译指示

FENV_ACCESS (C++11)

on(1) `#pragma STDC FENV_ACCESS on`
off(2) `#pragma STDC FENV_ACCESS off`

访问浮点环境

如果设置为 *on*，则程序会通知编译器它可能会访问 浮点环境 来测试它的 状态标志 （异常）或者运行在 控制模式 下而不是默认模式。

如果设置为 *off*，则编译器可能会做一些特定的优化来破坏这些测试和模式的改变，因此访问之前描述的 浮点环境 的话，会导致 未定义 行为。

这个编译指示的状态是 *on* 或 *off* 取决于编译器设置和库实现。

这个编译指示声明应该出现在：

- 在任何外部声明外 ：它的作用持续到遇到另一个 *FENV_ACCESS* 编译指示，或直到 编译单元 结束。
- 在复合语句中 ：这种情况下，它会优先于所有显示的声明和语句。它的作用持续到遇到另一个 *FENV_ACCESS* 编译指示（例如在一个内嵌的复合语句中），或直到复合语句的结束。复合语句结束后，编译指示的状态会重新被存储为进入它之前的状态。

如果这个编译指示出现在其他上下文中，则行为未定义。

当状态被这个编译指示直接改变时，浮点控制模式 （例如 舍入方向 ）拥有它们默认的设置，但 浮点标志 的状态是不确定的。

另请参见

函数	描述
fegetenv	获得浮点环境 (函数)
fesetenv	设置浮点环境 (函数)

头文件

<cfloat> (float.h)

浮点类型的特性

这个头文件描述了特定系统和编译器实现的浮点类型特性。

一个浮点数由四个元素组成：

- 符号：正或负
- 基底（或基数）：表示不同的数字，可以用单个数表示（二进制用 2，十进制用 10，十六进制用 16，等等）
- 有效数字（或尾数）：一系列上述提到的基底数字。这个序列中数字的个数被称作 精度。
- 指数（又称作特性值，或范围数）：表示有效数字的偏移量，通过下面的方式影响值：
  浮点值 = 有效数字 x 基底^{指数}，再加上它的符号。

宏常量

下面的表格显示了在这个头文件中定义的不同值的名字，以及在所有实现中它们的最大最小值。

当一组宏存在 *FLT_* ，*DBL_* 和 *LDBL_* 的前缀时，以 *FLT_* 开头的应用于 *float* 类型，以 *DBL_* 开头的适用于 *double* ，以 *LDBL__* 开头的适用于 *long double*。

名字	值	代表	描述
FLT_RADIX	2 或 > 2	基数(RADIX)	所有浮点类型的基底（*float* ，*double* 和 *long double*）
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG		尾数数字(MANTissa DIGits)	有效数字 的精度，也就是说，数字的个数和 有效数字 保持一致。
FLT_DIG DBL_DIG LDBL_DIG	6 或 > 6 10 或 > 10 10 或 > 10	数字(DIGits)	四舍五入成浮点数和还原后不改变的 十进制数字的个数。

函数

tolower

<cctype>

```
int tolower ( int c );
```

将大写字母转化为小写

如果 *c* 是一个大写字母并且存在对应的小写字母的话，则将 *c* 转化为对应的小写字母，否则返回 *c* 的原始值。

注意，判别一个字母是什么取决于使用环境。在默认的 “C” 环境中，大写字母有：*A B C D E F G H I J K L M N O P Q R S T U V W X Y Z*，对应转化成的小写字母分别是：*a b c d e f g h i j k l m n o p q r s t u v w x y z*。

在其他环境中，如果一个大写字母有多个对应的小写字母，那么对于同一个值 *c*，这个函数总是返回同样的字符。

在 C++ 中，这个函数的 locale-specific 模板版本 tolower 在头文件 <locale>中。

参数

`c`

被转化的字符，被转化为 *int* 型或 *EOF*。

返回值

返回 *c* 对应的小写字符，如果存在的话，否则返回 *C* 本身（未改变）。返回值是能被隐式转化为 *char* 的一个 *int* 型值。

例子

```
/* tolower example */
#include <stdio.h>
#include <ctype.h>

int main()
{
    int i = 0;
    char str[] = "Test String.\n";
    char c;
    while(str[i])
    {
        c = str[i];
        putchar(tolower(c));
        i++;
    }
```

```
16.     return 0;
17. }
```

输出：

```
1. test string.
```

另请参阅

函数名	描述
toupper	将小写字母转化为大写 (函数)
isupper	检查字符是否是大写字母(uppercase letter) (函数)
islower	检查字符是否是小写字母(lowercase letter) (函数)
isalpha	检查字符是否是字母(alphabetic) (函数)

函数

toupper

<cctype>

```
int toupper ( int c );
```

将小写字母转化为大 写

如果 *c* 是一个小写字母并且存在对应的大写字母的话，则将 *c* 转化为对应的大写字母，否则返回 *c* 的原始值。

注意，判别一个字母是什么取决于使用环境。在默认的 “C” 环境中，小写字母有: *a b c d e f g h i j k l m n o p q r s t u v w x y z*，对应转化成的大写字母分别是*A B C D E F G H I J K L M N O P Q R S T U V W X Y Z*。

在其他环境中，如果一个小写字母有多个对应的大写字母，那么对于同一个值 *c*，这个函数总是返回同样的字符。

在 C++ 中，这个函数的 locale-specific 模板版本 toupper 在头文件 <locale>中。

参数

`c`

被转化的字符，被转化为 *int* 型或 *EOF*。

返回值

返回 *c* 对应的大写字符，如果存在的话，否则返回 *C* 本身（未改变）。返回值是能被隐式转化为 *char* 的一个 *int* 型值。

例子

```
1.  /* toupper example */
2.  #include <stdio.h>
3.  #include <ctype.h>
4.
5.  int main()
6.  {
7.      int i = 0;
8.      char str[] = "Test String.\n";
9.      char c;
10.     while(str[i])
11.     {
12.         c = str[i];
13.         putchar(toupper(c));
14.         i++;
15.     }
```

```
16.     return 0;
17. }
```

输出：

```
1. TEST STRING.
```

另请参阅

函数名	描述
tolower	将大写字母转化为小写 (函数)
islower	检查字符是否是小写字母(lowercase letter) (函数)
isupper	检查字符是否是大写字母(uppercase letter) (函数)
isalpha	检查字符是否是字母(alphabetic) (函数)

- <vector>

头文件

<vector>

Vector 头文件

定义 `vector` 容器类的头文件

类

类名	描述
`vector`	向量（类模板）
`vector<bool>`	bool 向量（类模板特化）

函数

类名	描述
`begin`	指向头部的迭代器（函数模板）
`end`	指向尾部的迭代器（函数模板）

类模板

std::vector

```
templete < class T, class Alloc = allocator<T> > class vector; //generic template
```

vector

Vector 是序列式容器，表示大小可变的数组。

和数组一样的地方是，vector 使用连续的空间位置存储元素，这就意味着可以使用带偏移量的普通指针来访问其中的元素，就和像在数组中一样高效。但是和数组不一样的是，vector 的大小可以动态改变，容器会自动处理它们的存储。

vectors 内部使用一个动态分配的数组来存储它们的元素。当新元素被插入时，这个数组为了增加大小可能需要被重新分配，这意味着分配一个新数组，并将所有元素移动到这个新数组中。在处理时间方面这是一个相对昂贵的开销，因此，当元素被添加到容器时，vectors 不会每次都重新分配空间。

vector 容器可能会分配一些额外的存储空间来适应可能的增长，因此容器真实的 capacity 可能会大于它当前包含的所有元素的大小(也就是它的 size)。不同库可以使用不同的增长策略来平衡内存使用和重新分配，但无论如何，重新分配的区间大小应该呈对数级增长，这样单个元素插入 vector 的尾部才能分摊成常数时间复杂度(请查看 push_back)。

因此，和数组相比，vector 会消耗更多的内存来换取管理存储以及动态增长的高效性。

与其他动态序列式容器(deques，lists 和 forward_lists)相比，vector 访问它的元素还是很高效的，在尾部添加和移除元素相对也很高效。在除了尾部以外的位置插入或移除元素，vector 都没有其他容器高效，并且比 lists 和 forward_lists 拥有更少的稳定的迭代器（迭代器会失效）。

容器属性

序列化

序列式容器中的元素排列在一个严格线性的序列中。元素可以通过它们在序列中的位置来访问。

动态数组

允许直接访问序列中的任何元素，即使通过指针算数，并且能相对快速地从序列尾部添加/删除元素

内存分配器感知的

容器使用一个内存分配器对象来动态处理它的存储需求。

模板参数

T

元素的类型。
仅仅当 T 保证移动的时候不抛出异常，实现才能在重新分配内存的时候进行优化，用移动元素的方式代替拷贝。
别名是成员类型 vector::value_type

`Alloc`

内存分配器对象的类型，用来定义存储分配器模型。默认使用 allocator 类模板，它定义了最简单的内存分配模型并且是与值无关的。
别名是成员类型 vector::allocator_type

成员类型

C++98

类型名	定义	注释
value_type	第一个模板参数 (T)	
allocator_type	第二个模板参数 (Alloc)	默认值为: allocator<v
reference	allocator_type::reference	对于默认的 alloc value_type&
const_reference	allocator_type::const_reference	对于默认的 alloc const value_typ
pointer	allocator_type::pointer	对于默认的 alloc value_type*
const_pointer	allocator_type::const_pointer	对于默认的 alloc const value_typ
iterator	一个指向 value_type 的随机访问迭代器	可以转化为 const_
const_iterator	一个指向 const value_type 的随机访问迭代器	
reverse_iterator	reverse_iterator<iterator>	
const_reverse_iterator	reverse_iterator<const_iterator>	
difference_type	一个有符号整数类型，相当于: iterator_traits<iterator>::difference_type	通常和 ptrdiff_t
size_type	一个无符号整数类型，可以表示任何 difference_type 的非负值	通常和 size_t 一

C++11

类型名	定义	
value_type	第一个模板参数 (T)	
allocator_type	第二个模板参数 (Alloc)	默认值为: alloca
reference	value_type&	
const_reference	const value_type&	
pointer	allocator_traits<allocator_type>::pointer	对于默认的 value_type

const_pointer	allocator_traits<allocator_type>::const_pointer	对于默认的 const valu
iterator	一个指向 value_type 的随机访问迭代器	可以转化为
const_iterator	一个指向 const value_type 的随机访问迭代器	
reverse_iterator	reverse_iterator<iterator>	
const_reverse_iterator	reverse_iterator<const_iterator>	
difference_type	一个有符号整数类型，相当于: iterator_traits<iterator>::difference_type	通常和 ptrd
size_type	一个无符号整数类型，可以表示任何 difference_type 的非负值	通常和 size

成员函数

非成员函数重载

模板特殊化