# 基本数据类型

## 1 vs\_32位和64位

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 大小 | 数据类型 | 大小 |
| int=signed int | 4B | char | 1B |
| unsigned int | 4B | bool | 不支持 |
| short=short int=signed short=signed short int | 2B |  |  |
| unsigned short=unsigned short int | 2B | float | 4B |
| long=long int=signed long=signed long int | 4B | double | 8B |
| unsigned long= unsigned long int | 4B | long double | 8B |
| long long =long long int= signed long long =signed long long int | 8B |  |  |
| 指针:32位编译器4B，64位编译器8B |  | sizeof | 32位4B，64位8B |

## 2 GCC\_64位

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 大小 | 数据类型 | 大小 |
| int=signed int | 4B | char | 1B |
| unsigned int | 4B | bool | 不支持 |
| short=short int=signed short=signed short int | 2B |  |  |
| unsigned short=unsigned short int | 2B | float | 4B |
| long=long int=signed long=signed long int | 8B | double | 8B |
| unsigned long= unsigned long int | 4B | long double | 16B |
| long long =long long int= signed long long =signed long long int | 8B |  |  |
| 指针:32位编译器4B，64位编译器8B |  | sizeof | 32位4B，64位8B |

# 数组和指针

赋值运算符的优先级是最低的，赋值的左值必须是变量，如if（j==5||i=3）相当于if（(j==5||i)=3）

## 1 标准输入函数

#include <stdio.h>

int scanf( const char \*format, ... );

功能：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()](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/stdio_details.html#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. 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.也就是说的域宽可以控制输入字符的长度。 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. 控制字符以一个%符号开始,如下:

|  |  |  |  |
| --- | --- | --- | --- |
| %d | 一个有符号十进制int整数 | %c | 一个单一的字符 |
| %i | 一个有符号int整数 | %p | 一个指针 |
| %u | 一个无符号int整数 |  |  |
| %o | 一个无符号int八进制数 | %% | 一个百分号'%' |
| %x | 一个无符号int十六进制数 | %[] | 一个字符集(类似正则表达式) |
| %f，%e，%g | 一个浮点数，可以是小数，指数，默认5位 |  |  |
| %s | 一个字符串，遇到空格、回车、TAB就结束读取，自动添加\0 | | |
| %n | 代表读取的字符数，其大小 | | |

scanf()函数的一般格式为：

scanf("格式字符串"，输入项地址表)

scanf的格式控制的一般形式为：

%[\*][宽度][F|N][h|l] 控制字符 //[]中的控制字符为可选项 。

F 、N、h、l,ll分别表示远指针、近指针、短整和长整型，超长整型

"\*"，表示该输入项读入后不赋予任何变量，即跳过该输入值。

"宽度"，表示输入读入字符的长度，对于整型表示截取相应宽度的数字赋给后面列表中的相应变量；对于字符型表示读入相应长度的字符后把第一个字符赋给相应的变量，其余的自动舍弃。例如scanf("%2d%3d",&a, &b);如果输入为12345则将12赋给a，将345赋给b；scanf("%2c%3c",&a, &b);如果输入为12345则将'1'赋给a，将'3'赋给b .

"%s"，整个输入作为一个串,并设置末尾的'/0'

"%ns"，n为整数,读入的串最长不超过n,然后在末尾补'/0'

"%nf"，读入的浮点数最多有n位整数,位数多于n,会截断。

"%n[a-z]"， 读入最多n个字符,如果遇到非a-z的字符,停止

"%[^=]"， 读入任意多的字符,直到遇到"="停止

"%n[^=]"， 读入"="号前的至多n 个字符

对于输入字符串还有一些比较有用的控制。例如经常需要读入一行字符串，而这串字符里面可能有空格、制表符等空白字符，如果直接用%s是不可以的，于是有些人就想到用gets(),当然这也是一种选择，但是懂C的人基本上都知道gets()是一个很危险的函数，而且很难控制，特别是与scanf()交替使用时前者的劣势更是一览无余，所以gets()一般是不推荐用的，其实用%[^/n]就可以很好的解决这个问题了，^表示"非"，即读入其后面的字符就结束读入。这样想读入一行字符串直接用scanf("%[^\n]%\*c",str);就可以了，%\*c的作用是读入\n,否则后面读入的将一直是\n。

所有对%s起作用的控制都可以用%[],比如%[0-9]表示只读入'0'到'9'之间的字符，%[a-zA-Z]表示只读入字母，'-'是范围连接符，当然也可以直接列出你需要读入的字符。如果你只需要读"abc"里面的字符就可以用%[abc] (或者%[cab]、%[acb]、%[a-c]、%[c-a].....),

如果想读入某个范围之外的字符串就在前面加一个'^',如：%[^a-z]就表示读入小写字母之外的字符。

“\*”在scanf中表示读取后不赋给变量，在printf中表示域宽或小数位数，由第二个或第三个参数决定如

printf(“%-\*.\*s%s”，width，precisely,””,hello”)可在hello前输出width个空格

m域宽，n表示小数位数或者截取字串的长度， -表示左对齐。

注意：scanf输入时必须严格按照格式串中输入。不能加减空格。

## 2 标准输出函数

#include <stdio.h>

int printf( const char \*format, ... );

功能：printf()函数根据format(格式)给出的格式打印输出到STDOUT(标准输出). 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.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

|  |  |  |  |
| --- | --- | --- | --- |
| %d | 带符号int整数 | %c | 字符 |
| %i | 带符号int整数 | %s | 一串字符，遇到\0就停止输出 |
| %u | 无符号int整数 | %o | 八进制int整数 |
| %f | 浮点数 | %x | 无符号十六进制int整数, 用小写字母 |
| %e | 科学计数法, 使用小写"e" | %X | 无符号十六进制int整数, 用大写字母 |
| %E | 科学计数法, 使用大写"E" | %p | 一个指针 |
| %g | 使用%e或%f中较短的一个 | %n | 参数是一个指向一个整数的指针.指向的是字符数放置的位置 |
| %G | 使用%E或%f中较短的一个 | %% | 一个精度符号'%' |

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 的%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](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/escape_sequences.html).

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.

#include <stdio.h>

int getchar( void );

功能：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.

#include <stdio.h>

int putchar( int ch );

功能：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.

scanf 读取数值时是自动忽略\n，而读取字符时不会忽略。%c,每次读取一个字符，%s一次读取字符串，遇到\n，停止读取。Scanf错误时返回EOF，gets 出错时返回NULL。Scanf遇到空格，回车TAB，都会停止读取，而gets则不会。

## 3 转义字符

|  |  |  |
| --- | --- | --- |
| 转义字符 | 意义 | ASCII码值（十进制） |
| \a | 响铃(BEL) | 007 |
| \b | 退格(BS) ，将当前位置移到前一列 | 008 |
| \f | 换页(FF)，将当前位置移到下页开头 | 012 |
| \n | 换行(LF) ，将当前位置移到下一行开头 | 010 |
| \r | 回车(CR) ，将当前位置移到本行开头 | 013 |
| \t | 水平制表(HT) （跳到下一个TAB位置） | 009 |
| \v | 垂直制表(VT) | 011 |
| \\ | 代表一个反斜线字符''\' | 092 |
| \' | 代表一个单引号（撇号）字符 | 039 |
| \" | 代表一个双引号字符 | 034 |
| \? | 代表一个问号 | 063 |
| \0 | 空字符(NULL) | 000 |
| \ooo | 1到3位八进制数所代表的任意字符 | 三位八进制 |
| \xhh | 1到2位十六进制所代表的任意字符 | 二位十六进制 |

## 4 原码、反码、补码

正数原码=反码=补码，负数反码是将符号位以外求反，负数补码是对其反码加1，如出现进位应进位到符号位为止。

补码的作用一是为了运行减法[A-B]补码=A补+（-B）补码，二是统一+0和-0。数字在计算机中是用补码表示的。

## 5 数组与指针

指针变量：存放某内存空间地址的变量，指针的值是所指对象的地址，二级指针的值就是一级指针变量所在内存单元的地址 。

野指针 ：指向不属于自己的内存空间的指针。

指针的使用场景：偏移和传参，否则不需要使用指针。

一维数组作为实参传递的时候会退化为指针。所以一维数组传递时必须同时传入长度；一维数组作为形参，应该空下标；而二维数组传递时会退化为数组指针，因而必须传入数组的第一个下标或者两个下标；而二位数组作形参应该空一维下标

左移运算：算术和逻辑左移一样，右边补0；右移运算>>：算术右移要考虑符号位，逻辑右移直接补0

scanf输入字符串时如果输入的字符数大于数组大小就会越界，非法占用其余内存空间，scanf会自动补\O，并且scanf遇到空格回车制表符就结束读取。

指针存放的是对象的地址。对象可以是函数，数组，数组元素，变量，结构体变量，共用体变量，枚举变量等。

Int \*p; int a[10]; p=a;

则

p[i]；与a[i]；与\*（p+i）；与\*（a+i）;

等价

Char a[]=”I love you”；int \*p=a；与char \*p=“I love you”有本质的差别，前者是将字符串赋值给数组，此时P[i]是可以修改的，而后者是将常量的地址赋值给指针p，而此时P[i]是不可以修改的。即指向常量的指针和指向变量的指针。

一维数组下标必须为常量，或者定义同时赋初值则可以省略下标。数值型数组不赋值默认为0，字符型数值默认为\0。可以一次赋值一个，也可以整体赋值。a[]={1，2，3，4，,5}

动态内存申请和释放 ：

（数据类型 \*）malloc（大小）

free(指针)

（数据类型\*）realloc（原始指针，新的大小）//成返回地址，失败返回NULL；

（数据类型\*）calloc（空间个数，每个空间大小）//申请空间个数\*每个空间大小这么多空间并返回地址，同时置为0

memset（空间起始地址，要赋予的值，空间大小）//将一段空间赋予值

memcpy（目标地址，原地址，大小）//拷贝内存空间

memmove（目标地址，原地址，大小）//移动内存空间

memcmp（目标地址1，目标地址2，字节个数）

memchr（目标地址，待查找字符，字节个数）

注意free（指针）指针必须是申请的堆空间的原始指针，如：

char \*p=(char \*)malloc(10);p++；free（p）；

则会出错。另外调用free（p），p所指的内存空间虽然释放了，但值不会改变。仍可以被使用，这时p就是野指针，因而要把指针置为NULL。

当返回值为指针时，该指针不能指向栈空间应该指向堆空间。

注意：malloc（0）是合法的，返回的地址不是NULL，可以用malloc\_usable\_size（void\*）查看真真实的大小，但是非常危险的 ,很可能会覆盖正在使用的内存。

memset函数 对指针所指向的空间大小得每个内存单元赋初始值。

## 6 几个概念的区别：

(1) 指针与数组

数组名相当于地址，但绝对不是一级指针，因为指针可以++或者--，数组不可以，但是都可以进行加法运算，其原因是因为数组名是一个地址常量，不可改变。而指针是一个变量，他有自己的地址，其值才是就是地址。

此外对指针取地址和指针指向的地址是不同的。但是对数组取地址的是相同的。并且数组名的加法和数组名的地址的加法运算不同，前者移动的单位长度是数组元素的大小，而后者移动的是整个数组长度。

（2）指针数组和数组指针：

指针数组是数组元素为指针的数组，数据类型 \*数组名[size]，主要用于对指针所指内容进行排序而不移动元素。如：char \*a[10]

数组指针是指向数组的指针，数据类型 （\*指针名）[size]，指针+1，偏移整个数组的大小.如int a[3][3],int（\*p）[3],p=a;

（3）指针函数与函数指针

指针函数 存储类型 返回值类型 \*函数名（形参表）；

函数指针 存储类型 返回值类型 （\*指针名）（形参表）；

函数指针的作用在于选择不同的函数来执行。如对两个数进行加减乘除运算，就可以定义一个指针分别指向这四个函数。或者使用函数指针作为函数参数，以传递不同的行为。回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。

对于实参是空指针，形参也为指针，则函数调用时候会出错，因而c中只能用二级指针在子函数中来对主函数的实参指针进行修改。否则就不需要使用二级指针。

如：

void fun（char \*p）

{

int i;

p=&i;

}

void main()

{

char \*p;

fun(p);

}

这里报错原因是因为传递的是p的值，而p的值是空的。正确的用法应该是用二级指针，这样尽管p的值是空的，但由于传递的是p所在地址因而不会报错。

Void fun（char \*\*p）

{

int i;

\*pp=&i

}

void main()

{

char \*p；

fun（&p）；

}

# 函数与自定义数据类型

## 1 变量的作用域和生存期

变量的定义格式：

存储类型 数据类型 变量名

函数或语句块内部：

auto 局部作用域，动态生存期，存放在动态存储区；(默认)

Static 局部作用域，静态生存期，存放在静态存储区；

register 局部作用域，动态生存期，放在cpu的寄存器中，register变量不能取址。

函数外部：

static 文件作用域（全局作用域），静态生存期，存放在静态存储区；限制该变量不可被extern扩展到其他源文件中使用

extern 扩大作用域（默认，实质是变量的声明而不是定义，因为不需申请空间）

## 2 函数的作用域和生存期

函数的定义格式：

存储类型 返回值类型 函数名（形参表）

static 文件作用域（全局作用域），静态生存期；限制该函数不可被extern扩展到其他源文件中使用。

extern 扩大作用域，（默认，可以省略）

## 3 C中的参数传递

传值：默认的传值，即在函数被调用的时候，给形参申请一个空间，再将实参的值传递给形参，对形参的任何改变不会影响实参数的值；无论传址、传址、传输组都是传实参的内容（变量的值或指针的值）而不会传地址。传址只是人为的传递将地址的值传递。

传址：所谓传址又叫做传指针，即在函数被调用的时候，给形参指针开辟一个空间用来存放传递过来的地址，将实参所在的内存地址传递给形参，对形参的任何改变也不会影响实参所指的内容，但是对形参所指的内容的改变将会影响到实参所指的内容（因为这两个指针都指向同一个内存空间）

传数组：实质是传址，将数组的起始地址传给形参，会退化为指针丢失数组长度，对形参的改变会改变实参。

可以将不同的函数模块放在不同的源文件中，但每个源文件都是分别编译，因而必须分别添加头文件。分别编译可以提高编译效率，在对某个源文件修改编译的时候只需单独修改编译某一个源文件。

①尽量不要把函数放在循环中

②尽量不要在子函数中定义数据结构，而通过指针传给子函数

③能用栈区就不用堆区，因为malloc申请堆区的时间开销远大于在栈区定义一个变量的的时间开销。

## 4 结构体、共用体、枚举

结构体：每个成员独享一个空间，大小为所有成员大小之和；struct。

共用体：所用成员共享一个空间，大小为最大成员的大小；所有成员的地址都相同，共用体的值等于最新赋值的成员，union。

枚举：枚举中每个成员代表一个常量，不可对成员赋值，默认从0开始依次递增，或者从指定值依次递增。enum。

### 4.1 结构体定义：

struct 结构体名

{

数据成员；

成员函数；

}；

结构体变量的定义：

（1）显式定义，struct 结构体名 变量名；（c++中科可以省略struct）

（2）隐式定义：定义结构体的同时定义变量；

如果结构体只使用一次，还可以省去结构体名，采取隐式定义结构体变量。

初始化：

（1）定义的同时用初始化列表初始化；

（2）定义后再初始化

### 4.2 联合体定义：

union 结构体名

{

数据成员；

成员函数；

}；

联合体与结构体不同的是空间大小的计算和存放的成员，联合体的大小是数据成员中最大的那个成员的大小，而结构体的大小至少是所有成员大小之和。联合体所有成员共享一个空间，任何时候都只有一个成员有效，所有成员的地址都是相同的，而结构体中每个成员都有独立的空间，每个成员都有效。

### 4.3 枚举定义

enum 枚举名{枚举常量1[=初值]，枚举常量2[=初值]，......};

（1）显式定义，enum 枚举名 枚举变量；（c++中科可以省略struct）

（2）隐式定义：定义枚举的同时定义变量；

如果枚举只使用一次，还可以省去枚举名，采取隐式定义枚举变量。

枚举不同于结构体和联合体，枚举是一系列常量的组合，没有名字。枚举中的数据成员可以直接使用，不能用成员访问符访问。

默认情况下，枚举常量从0开始，依次增加，如果某个枚举常量设置了初始值，则其后面的没有设置初始值的枚举常量就按该值依次增加，直到最后或者遇到下一个设了处置的枚举常量。

对于枚举变量只能赋值枚举常量，不能直接付给数值，如果必须这样，则需要强制转换。

## 5 函数声明与函数定义

函数声明时可以只写参数类型，不写参数名。函数定义时，不会访问的参数也可以只写参数类型，不写参数名

# 文件I/O

## 1 相关概念

文件：程序和数据的集合就叫文件。

文件缓冲区：由于内存和I/O设备的的速度差异很大。因而在内存中设置输入、输出缓冲区。读取时，先从输入设备中读取一批数据到输入缓冲区，然后在依次送给变量。输出时先存放在输出缓冲区，待缓冲区满或程序结束或主动刷新时才输出到输出设备。

文件信息区：每个打开一个文件，都会在内存中开辟一个文件信息区用于存放文件的相关相关属性。这些信息存放在一个名为FILE的结构体中。

标准I/O流与文件流：标准输入流、标准输出流、标出出错输出流（键盘，屏幕，屏幕）如putchar，getchar，putc，getc，puts，gets，printf，scanf。文件流（文本文件与二进制文件）如fputc，fgetc，fputs，fgets，fprintf，fscanf。实质就是流指针的不同，前者流指针为stdin或stdout，后者则为FILE型指针。

文件的存储形式：

1. ASCII文件，又叫文本文件，内存数据（二进制）——>ASCII码——>硬盘
2. 二进制文件，又叫映像文件，内存数据（二进制）——>硬盘

## 2 相关函数

### 2.1 打开/关闭/删除文件

|  |
| --- |
| **FILE \*fopen( const char \*fname, const char \*mode );** |

功能：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) ，文件名包括路径、主文件名、扩展名。

|  |  |  |
| --- | --- | --- |
| "r" | 以只读方式打开文本文件 | 该文件必须存在，否则打开时出错。 |
| "w" | 以只写方式打开文本文件 | 打开时若文件存在则清空该文件，若文件不存在则建立该文件。 |
| "a" | 以附加方式打开只写文件 | 打开时若文件不存在则建立该文件，如果存在则写入的数据会被加到文件尾。（原EOF保留） |
| "rb" | 以只读方式打开二进制文件 | 该文件必须存在，否则打开时出错。 |
| "wb" | 以只读方式打开二进制文件 | 打开时若文件存在则清空该文件，若文件不存在则建立该文件。 |
| "ab" | 以只读方式打开二进制文件 | 打开时若文件不存在则建立该文件，如果存在则写入的数据会被加到文件尾。（原EOF保留） |
| "r+" | 以可读写方式打开文本文件 | 该文件必须存在，否则打开时出错。 |
| "w+" | 以可读写方式打开文本文件 | 打开时若文件存在则清空该文件，若文件不存在则建立该文件。 |
| "a+" | 以附加方式打开可读写文本文件 | 打开时若文件不存在则建立该文件，如果存在则写入的数据会被加到文件尾。（原EOF不保留） |
| "rb+" | 以可读写方式打开二进制文件 | 该文件必须存在，否则打开时出错。 |
| "wb+" | 以可读写方式打开二进制文件 | 打开时若文件存在则清空该文件，若文件不存在则建立该文件。 |
| "ab+" | 以附加方式打开可读写二进制文件 | 打开时若文件不存在则建立该文件，如果存在则写入的数据会被加到文件尾。（原EOF不保留） |

|  |
| --- |
| **int fclose( FILE \*stream );** |

功能：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.

FILE \*fp；//定义一个文件指针

if（!(fp=fopen（”文件名”，打开方式）)）

printf(”打开出出错”)

…………

fclose（fp）//

|  |
| --- |
| **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.

### 2.2 顺序读/写文件

（1）对文本文件的顺序读写

|  |
| --- |
| **int fgetc( FILE \*stream );** |

功能：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.

注意： fgetc与getc一样自动将文本文件的\r\n转为\n读入，将文本文件的\0读入，将标准输入的\n读入。

|  |
| --- |
| **int fputc( int ch, FILE \*stream );** |

功能：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.

注意： fputc会把\0输出到文本文件和屏幕；将\n输出到文本文件作为\r\n，输出到屏幕作为\n。

|  |
| --- |
| **char \*fgets( char \*str, int num, FILE \*stream );** |

功能：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。若当前行字符数大于[num - 1]则下次接着读，自动将str[num]置为\0；若当前行字符少于[num - 1]，fgets()在到达行末时停止,在这种情况下,str(字符串)将会被一个换行符\n结束并且自动将str[num]置为\0。 如果fgets()遇到EOF, str(字符串)将会以\0结束.fgets()成功时返回str(字符串),失败时返回NULL.

注意： gets()读入时将舍去换行符\n，仅用于标准输入，仅有一个参数； fgets()可用于标准输入和文本文件读入，自动将文本文件的\r\n和标准输入的\n转为\n，且读入换行符\n，有3个参数。

|  |
| --- |
| **int fputs( const char \*str, FILE \*stream );** |

功能：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.

注意：fputs输出\n到文本文件作为\r\n，到屏幕作为\n，遇到\0就停止输出且不会输出\0，有两个参数。fputs向文件和屏幕输出时不会自动换行。

puts仅用于标准输出，遇到\0就停止输出且不会输出\0。puts向屏幕输出时会自动换行。

|  |
| --- |
| **int fscanf( FILE \*stream, const char \*format, ... );** |

功能：函数fscanf()以[scanf()](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/stdio_details.html#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.

|  |
| --- |
| **int fprintf( FILE \*stream, const char \*format, ... );** |

功能：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()](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/stdio_details.html#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.

|  |
| --- |
| **int feof( FILE \*stream );** |

功能：函数feof()在到达给出的文件流的文件尾时返回一个非零值. 否则返回零。

（2）对二进制文件的顺序读写

|  |
| --- |
| **int fread( void \*buffer, size\_t size, size\_t num, FILE \*stream );** |

功能：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()](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/stdio_details.html#feof)或[ferror()](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/stdio_details.html#ferror)判断到底发生哪个错误.

|  |
| --- |
| **int fwrite( const void \*buffer, size\_t size, size\_t count, FILE \*stream );** |

功能：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.

### 2.3 随机读写文件

随机读写文件的实质就是改变文件标记位置的顺序读取

|  |
| --- |
| **void rewind( FILE \*stream );** |

功能：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.

|  |
| --- |
| **int fsetpos( FILE \*stream, const fpos\_t \*position );** |

功能：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.

|  |
| --- |
| **int fgetpos( FILE \*stream, fpos\_t \*position );** |

功能：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.

|  |
| --- |
| **int fseek( FILE \*stream, long offset, int origin );** |

功能：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):

|  |  |  |
| --- | --- | --- |
| 0 | SEEK\_SET | 从文件的开始处开始搜索 |
| 1 | SEEK\_CUR | 从当前位置开始搜索 |
| 2 | 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. offset必须是长整型。

注意：如果连续读或连续写不需用fseek刷新文件标记位置，但如果读后写或者写后读就必须刷新文件标记位置。

|  |
| --- |
| **long ftell( FILE \*stream );** |

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

1. 文件读/写出错

|  |
| --- |
| **int ferror( FILE \*stream );** |

功能：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()](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/stdio_details.html#perror)检测发生什么错误.

|  |
| --- |
| **void perror( const char \*str );** |

功能：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.

|  |
| --- |
| **void clearerr( FILE \*stream );** |

功能：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()](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/stdio_details.html#perror)判断实际上发生了何种错误

### 2.4 刷新输入/输出缓冲区

|  |
| --- |
| **int fflush( FILE \*stream );** |

功能：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(STDIN)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.

### 2.5 二进制和文本模式的区别

在windows系统中，文本文件下，文件以"\r\n"代表换行。若以文本模式打开文件，并用fputs等函数写入换行符"\n"时，函数会自动在"\n"前面加上"\r"。即实际写入文件的是"\r\n" 。

在类Unix/Linux系统中文本模式下，文件以"\n"代表换行。所以Linux系统中在文本模式和二进制模式下并无区别。

### 2.6 命令行参数

main(int argc,char \*argv[]) //命令行参数在属性-》调试-》命令行参数中设置

argc：由系统传给函数的程序清单数。若argc==3则：

argv[0] 指向程序运行的全路径名

argv[1] 指向在DOS命令行中执行程序名后的第一个字符串

argv[2] 指向执行程序名后的第二个字符串

# C参考手册

## 1 stdarg.h

<stdarg.h>主要作用是提供可变参数函数机制。

标准库提供的一些参数的数目可以有变化的函数。例如我们很熟悉的printf。这种函数被称作“具有变长度参数表的函数”，或简称为“变参数函数”。要定义这类函数，就必须使用标准头文件<stdarg.h>，使用该文件提供的一套机制，并需要按照规定的定义方式工作。

一个变参数函数至少需要有一个普通参数，其普通参数可以具有任何类型。在函数定义中，这种函数的最后一个普通参数除了一般的用途之外，还有其他特殊用途。下面从一个例子开始说明有关的问题。

假设我们想定义一个函数sum，它可以用任意多个整数类型的表达式作为参数进行调用，希望sum能求出这些参数的和。这时我们应该将sum定义为一个只有一个普通参数，并具有变长度参数表的函数，这个函数的头部应该是（函数原型与此类似）：int sum(int n, ...)，我们实际上要求在函数调用时，从第一个参数n得到被求和的表达式个数，从其余参数得到被求和的表达式。在参数表最后连续写三个圆点符号，说明这个函数具有可变数目的参数。凡参数表具有这种形式（最后写三个圆点），就表示定义的是一个变参数函数。注意，这样的三个圆点只能放在参数表最后，在所有普通参数之后。

为了能在变参数函数里取得并处理不定个数的“其他参数”，头文件<stdarg.h>提供了一套机制。这里提供了一个特殊类型va\_list。在每个可变参数函数的函数体里必须定义一个va\_list类型的局部变量，它将成为访问由三个圆点所代表的实际参数的媒介。

va\_list 媒介变量

下面假设函数sum里所用的va\_list类型的变量的名字是vap。在能够用vap访问实际参数之前，必须首先用“函数”va\_start做这个变量初始化。宏va\_start的类型特征可以大致描述为：

void va\_start(va\_list 媒介变量, 最后一个普通参数)

实际上va\_start通常并不是函数，而是用宏定义实现的一种功能。在函数sum里对vap初始化的语句应当写为：

va\_start(vap, n);

在完成这个初始化之后，我们就可以通过另一个宏va\_arg访问函数调用的各个实际参数了。宏va\_arg的类型特征可以大致地描述为：

T va\_arg(va\_list 媒介变量, T)

T 为va\_list中下一个参数的类型，在调用宏va\_arg时必须提供有关实参的实际类型，这一类型也将成为这个宏调用的返回值类型。对va\_arg的调用不仅返回了一个实际参数的值（“当前”实际参数的值），同时还完成了某种更新操作，使对这个宏va\_arg的下次调用能得到下一个实际参数。对于我们的例子，其中对宏va\_arg的一次调用应当写为：v = va\_arg(vap, int);这里假定v是一个有定义的int类型变量。

宏va\_copy()可以从源参数列表中复制参数值到已初始化的变量中。

va\_copy(目标，源)

在变参数函数的定义里，函数退出之前必须做一次结束动作。这个动作通过对局部的va\_list变量调用宏va\_end完成。这个宏的类型特征大致是：

void va\_end(va\_list 媒介变量);

下面是函数sum的完整定义，从中可以看到各有关部分的写法：

int sum(int n, ...) {

va\_list vap;

int i, s = 0;

va\_start(vap, n);

for (i = 0; i < n; i++) s += va\_arg(vap, int);

va\_end(vap);

return s;

}

这里首先定义了va\_list变量vap，而后对它初始化。循环中通过va\_arg取得顺序的各个实参的值，并将它们加入总和。最后调用va\_end结束。

下面是调用这个函数的几个例子：

k = sum(3, 5+8, 7, 26\*4);

在编写和使用具有可变数目参数的函数时，有几个问题值得注意。

首先，因为没办法写出来有关的类型，系统在预处理时进行宏展开，编译时即使发现错误，也无法提供关于这些宏调用的错误信息。所以，在使用这些宏的时候必须特别注意类型的正确性，系统通常无法自动识别和处理其中的类型转换问题。

第二：调用va\_arg将更新被操作的va\_list变量（如在上例的vap），使下次调用可以得到下一个参数。在执行这个操作时，va\_arg并不知道实际有几个参数，也不知道参数的实际类型，它只是按给定的类型完成工作。因此，写程序的人应在变参数函数的定义里注意控制对实际参数的处理过程。上例通过参数n提供了参数个数的信息，就是为了控制循环。标准库函数printf根据格式串中的转换描述的数目确定实际参数的个数。如果这方面信息有误，函数执行中就可能出现严重问题。编译程序无法检查这里的数据一致性问题，需要写程序的人自己负责。在前面章节里，我们一直强调对printf等函数调用时，要注意格式串与其他参数个数之间一致性，其原因就在这里。

第三：编译系统无法对变参数函数中由三个圆点代表的那些实际参数做类型检查，因为函数的头部没有给出这些参数的类型信息。因此编译处理中既不会生成必要的类型转换，也不会提供类型错误信息。考虑标准库函数printf，在调用这个函数时，不但实际参数个数可能变化，各参数的类型也可能不同，因此不可能有统一方式来描述它们的类型。对于这种参数，C语言的处理方式就是不做类型检查，要求写程序的人保证函数调用的正确性。

## 2 limits.h

<limits.h>的主要作用是给出基本整数类型的极限值

|  |  |  |
| --- | --- | --- |
| CHAR\_BIT | | 字节的位数 (宏常量) |
| MB\_LEN\_MAX | | 多字节字符的最大字节数 (宏常量) |
| CHAR\_MIN | | char的最小值 (宏常量) |
| CHAR\_MAX | | char的最大值 (宏常量) |
| SCHAR\_MIN SHRT\_MIN INT\_MIN | LONG\_MIN LLONG\_MIN(C99) | 各为signed char、 short、 int、 long和long long的最小值 (宏常量) |
| SCHAR\_MAX SHRT\_MAX INT\_MAX | LONG\_MAX LLONG\_MAX(C99) | 各为signed char、 short、 int、 long和long long的最大值 (宏常量) |
| UCHAR\_MAX USHRT\_MAX UINT\_MAX | ULONG\_MAX ULLONG\_MAX(C99) | 各为unsigned char、 unsigned short、 unsigned int、unsigned long和unsigned long long的最大值 (宏常量) |

## 3 非局部跳转—setjmp.h

|  |
| --- |
| **#include <setjmp.h>**  **int setjmp( jmp\_buf envbuf );** |

功能： 函数将系统栈保存于jmp\_buf 类型的变量*envbuf*中，以供以后调用[longjmp()](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/stdother_details.html#longjmp)。返回值是记录保存系统栈的次数。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。

|  |
| --- |
| **#include <setjmp.h>**  **void longjmp( jmp\_buf envbuf, int status );** |

功能： 函数使程序从前次对[setjmp()](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/stdother_details.html#setjmp)的调用处继续执行。参数*envbuf*一般通过调用[setjmp()](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/stdother_details.html#setjmp)设定。参数*status* 为[setjmp()](mk:@MSITStore:C:\Users\LIAOWEIZHI\Desktop\C／C++函数大全.chm::/cppreference.com/stdother_details.html#setjmp)的返回值，用来指示不同地点longjmp()的执行. *status* 不能设定为零。

使用setjmp和longjmp要注意以下几点：

（1）setjmp与longjmp结合使用时，它们必须有严格的先后执行顺序，也即先调用setjmp函数，之后再调用longjmp函数，以恢复到先前被保存的“程序执行点”。否则，如果在setjmp调用之前，执行longjmp函数，将导致程序的执行流变的不可预测，很容易导致程序崩溃而退出。

（2）longjmp必须在setjmp调用之后，而且longjmp必须在setjmp的作用域之内。具体来说，在一个函数中使用setjmp来初始化一个全局标号，然后只要该函数未曾返回（结束），那么在其它任何地方都可以通过longjmp调用来跳转到 setjmp的下一条语句执行。实际上setjmp函数将发生调用处的局部环境保存在了一个jmp\_buf的结构当中，只要主调函数中对应的内存未曾释放 （函数返回时局部内存就失效了），那么在调用longjmp的时候就可以根据已保存的jmp\_buf参数恢复到setjmp的地方执行。

（3）函数不能反悔动态生存期的对象。因为动态生存期的对象在函数调用结束后会即刻销毁。

## 4 原子操作库（C11）

原子操作：是一种无法再进行细分的操作,用于解决并发操作中数据竞争的问题.观察一个atomic变量时,观察者只会得到变量的原值或变量被他人修改后的新值,而不会观察到一个修改中的状态.

例如,现有一个变量值为二进制10,需要将其修改为01,写操作每次修改一位数字,需要进行两次操作(先把首位1修改成0,再将第二位0改成1),对于一个非atomic变量,观察者可能会看到00值(修改进行了一半),使用atomic则相当于对写操作进行了打包,保证观察者要么看到10,要么看到01,从而保证了值的完整性和操作结果的确定性. (这里只是举个例子,实际在x86/64的机器上,对于整型的操作本身就是原子的)

### 4.1 类型

#### 4.1.1 内存模型

在我们编写的 C/C++程序和它在 CPU 上运行,按照一些规则,代码的内存交互顺序（执行顺序）会被乱序.内存乱序同时由编译器(编译时候)和处理器(运行时)造成,都为了使代码运行的更快.

因为C++03与c99标准是单线程的，所以即便是完全符合标准的编译器也只考虑单个线程，于是在对代码作优化的时候总是一不小心就可能做出危害多线程正确性的优化来。编译开发者和处理器制造商遵循的中心内存排序准则是:不能改变单线程程序的行为.

因为这条规则,在写单线程代码时内存乱序被普遍忽略.即使在多线程程序中,它也被时常忽略,因为有 mutexes,semaphores 等来防止它们调用中的内存乱序. 在Hans Boehm的paper中提到，编译器可以运用“Register Promotion”的技术进行优化，对此，POSIX线程库也无能为力。当 lock-free 技术被使用时,内存在不受任何互斥保护下被多个线程共享,内存乱序的影响能被看到.

为了在多核cpu上多线程编程，最简单的办法就是禁止编译器作任何优化：所有的操作严格按照“Program Order”执行，所有的操作都触发“Cache Coherence”以确保它们的副作用在跨线程间的“Memory Visibility（内存可见）”顺序。但这样做显然是不切实际的，需要付出巨大的效率代价。于是编译器说：“不如这样，你来告诉我哪些数据是线程间共享的。这样，我就可以在必要的时候保守优化，一般情况下全力优化。只要你保证自己的程序是正确同步的，那我保证程序执行时就是你要的那个样子。”这样一个在程序员和编译器之间的约定，就是“Memory Model”。

C++11与C11新增了 Atomic operations library和Thread support library。 “线程支持库”可以简单想象成POSIX线程库的OO版本，对常用的Threads、Mutex、Condition Variables、Futures等概念进行了很好的封装。

“原子操作库”顾名思义，其实就是原子操作库。而在以往，我们往往需要借助汇编语言或者第三方线程库方能实现。atomic对于多线程编程，尤其是lock-free算法，其重要性不言而喻，有了原子操作库，我们可以摆脱那些繁琐的汇编代码了！

PS：乐衷于lock-free编程的读者需要注意的一点，并非所有的atomic内置类型操作均是lock-free的，与具体平台相关，可以调用is\_lock\_free（）接口进行查询。

#### 4.1.2 内存序

enum memory\_order

{

memory\_order\_relaxed,

memory\_order\_consume,

memory\_order\_acquire,

memory\_order\_release,

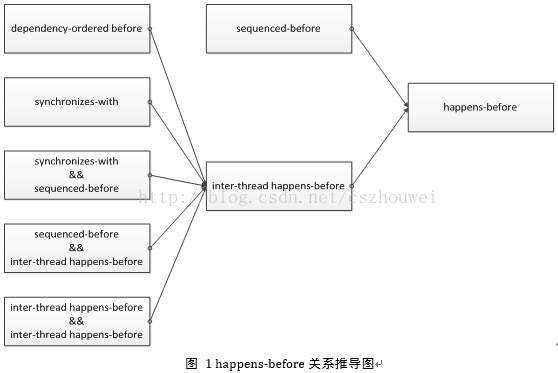
memory\_order\_acq\_rel,

memory\_order\_seq\_cst

};

atomic库的中几乎每个方法都有两个版本，一版本不带内存序参数，即使用默认内存序，一个版本带类型为memory\_order的参数。

C11引入memory order和内存模型本质上是为了解决 “visible side-effects”的问题，用通俗的话来讲：线程1执行写操作A之后，如何可靠并高效地保证线程2执行读操作B时，操作A的结果是完整可见的？为了解决这个问题，C11引入了“happens-before”关系，定义如下：A、B代表多线程中的两个线程，如果A happens-before B, 则A线程的操作对内存的影响在B线程执行操作之前应对B可线程可见。现在问题就转化为：如何在两个操作之间建立起happens-before关系呢？



sequenced-before（线程内）：在同一个线程内，操作A先于操作B

carries-a-dependency-to：如果操作A的结果用于操作B的操作当中，那么A carries-a-dependency-to（将依赖带入）B。

dependency-ordered-before：如果操作B的结果进一步在相同的线程内被操作C使用，那么A的store操作是dependency-ordered-before（在依赖执行序列X之前）B的load操作。

synchronizes-with（线程间）：线程1的操作A对变量M执行“release”写，线程2的操作B对变量M执行“acquire”读，并且操作B读取到的值源于操作A之后的“release”写序列中的任何一个（包括操作A本身）。

我们可以把上述 6 中访存次序(内存序)分为 3 类：

|  |  |  |
| --- | --- | --- |
| 模型 |  |  |
| Sequential consistency | memory\_order\_seq\_cst |  |
| Acquire-Release | memory\_order\_acq\_rel |  |
| memory\_order\_acquire  memory\_order\_release |
| memory\_order\_consume |
| Relax | memory\_order\_relaxed | 只保证在同一个线程内，同一个原子变量的操作的执行序列不会被重排序（reorder），但是线程间执行关系是任意 |

三种不同的内存模型在不同类型的 CPU上(如 X86，ARM，PowerPC等)所带来的代价也不一样。例如，在 X86 或者 X86-64平台下，Acquire-Release 类型的访存序不需要额外的指令来保证原子性，即使顺序一致性类型操作也只需要在写操作(Store)时施加少量的限制，而在读操作(Load)则不需要花费额外的代价来保证原子性。

memory\_order\_relaxed：没有顺序一致性的要求，也就是说同一个线程的原子操作还是按照happens-before（按照程序的代码序执行）关系，但不同线程间的执行关系是任意。

memory\_order\_seq\_cst：以牺牲优化效率，来保证指令的顺序一致执行，指明在线程间建立一个全局的执行序列，所有线程执行指令的顺序都是按照源代码的顺序，每个线程所能看到其他线程的操作的执行顺序都是一样的，相当于不打开编译器优化指令，线程内按照正常的指令序执行(happens-before)，多线程各原子操作也会Synchronized-with（同步），（譬如load()需要等待store()写下元素才能读取），当然这里还必须得保证一致性，读操作需要在“一个写操作对所有处理器可见”的时候才能读，适用于基于缓存的体系结构。对所有的变量的所有原子操作都同步。所有的原子操作就跟由一个线程顺序执行似的。是X86系列cpu默认的。

memory\_order\_acquire、memory\_order\_release、memory\_order\_acq\_rel：这个是对relaxed的加强，relax由于无法限制多线程间的排序，所以引入synchronized-with。在线程间同一个原子变量的release操作和acquire操作间同步，同时也就建立起了执行序列约束。确保所有的读和写动作不能移动到acquire操作之前，所有的读和写动作不能移动到release操作之后。release-acquire操作建立了inter-thread happens-before。所以acquire之后的操作和release之前的操作就能进行同步。同时，release-acquire操作具有传递性。

memory\_order\_consume：这个内存序是 “获取-释放”的一部分，但它引入了数据依赖。包括dependency-ordered-before和carries-a-dependency-to 。

atmoic和memory\_order只有在多CPU、多线程情况下，无锁编程才会用到。在x86下，由于是strong memory order的，所以很多时候只需要考虑编译器优化的memory order；保险起见，可以用atomic，他会同时处理编译器优化和cpu的memory reorder（虽然x86用不到）。在除非必要的情况下，不用使用memory\_order， atmoic默认用的是最强限制。

atomic原子型变量,只保证了对于这个变量本身的修改是无需上锁的,如不添加其他设定(即atomic中memory\_order设为relaxed),其保护范围是这个变量自身. mutex锁,保护的则是代码段,其保证的是两段代码同一时间只能有其中一段在被计算. 也就是说, atomic + 正确的memory\_order = 锁。

memory\_order 规定了普通访存操作和相邻的原子访存操作之间的次序是如何安排的，在多核系统中，当多个线程同时读写多个变量时，其中的某个线程所看到的变量值的改变顺序可能和其他线程写入变量值的次序不相同。同时，不同的线程所观察到的某变量被修改次序也可能不相同。然而，如果保证所有对原子变量的操作都是顺序的话，可能对程序的性能影响很大，因此，我们可以通过memory\_order 来指定编译器对访存次序所做的限制。因此，在原子类型的操作中，可以通过额外的参数指定该原子操作的访存次序(内存序)，默认的内存序是 memory\_order\_seq\_cst。

原子操作所对应的内存模型可分为3大类：

读操作：memory\_order\_acquire, memory\_order\_consume

写操作：memory\_order\_release

读-修改-写操作：memory\_order\_acq\_rel, memory\_order\_seq\_cst

而memory\_order\_relaxed没有定义任何同步语义和顺序一致性约束。

atomic类中所有操作如果不做指定默认使用memory\_order\_seq\_cst 。如果自己指定内存屏障类型：

Store操作只能附带 memory\_order\_relaxed, memory\_order\_release, memory\_order\_seq\_cst ，

Load操作只能附带 memory\_order\_relaxed, memory\_order\_acquire, memory\_order\_seq\_cst , memory\_order\_consume.

Read\_Modify\_Write操作能附带所有六种内存屏障

store和load所附带的memory\_order应是成对使用的, 例如在一个load(acquire)之前,必须要发生一个store(release), 否则memory\_order将不会生效. 这就好比一个lock必须前置一个unlock,否则死锁.

例如：

lock();

b = a.load();

a.store(b+1);

unlock();

如果内存模型是relax，当线程A执行unlock时，因为指令乱序的关系，对变量a的写操作可能尚未完成，而线程B的lock()函数则读出了state\_的值发现已解锁，进入临界区，那么线程B的a.store(b+1)就会跟线程A的a.store(b+1)发生data race。而release模型保证了unlock()之上所有的写语句都已完成才写state\_，acquire模型则保证lock()之后的所有读语句都需等待state\_的值读出后执行。这样就确保了临界区内不会发生data race。

#### 4.1.3 原子旗标类型

struct atomic\_flag

atomic\_flag是原子布尔类型。与其他原子类型不同，它保证是无锁的。与atomic\_bool不同，atomic\_flag不提供加载或存储操作。

### 4.2 宏

ATOMIC\_BOOL\_LOCK\_FREE

ATOMIC\_CHAR\_LOCK\_FREE

ATOMIC\_CHAR16\_T\_LOCK\_FREE

ATOMIC\_CHAR32\_T\_LOCK\_FREE

ATOMIC\_WCHAR\_T\_LOCK\_FREE

ATOMIC\_SHORT\_LOCK\_FREE

ATOMIC\_INT\_LOCK\_FREE

ATOMIC\_LONG\_LOCK\_FREE

ATOMIC\_LLONG\_LOCK\_FREE

ATOMIC\_POINTER\_LOCK\_FREE

### 4.3 函数

\_Bool atomic\_flag\_test\_and\_set( volatile atomic\_flag\* obj );(1)

\_Bool atomic\_flag\_test\_and\_set\_explicit( volatile atomic\_flag\* obj, memory\_order order );(2)

原子性地更改obj所指向的atomic\_flag的状态为已设置（true）并返回先前的值。第一版本根据memory\_order\_seq\_cst排序内存访问，第二版本根据order排序内存访问。

参数obj是指向volatile atomic\_flag的指针，可以接受非volatile和volatile（例如映射到内存的I/O）的atomic\_flag。

void atomic\_flag\_clear( volatile atomic\_flag\* obj );(1)

void atomic\_flag\_clear\_explicit( volatile atomic\_flag\* obj, memory\_order order );(2)

原子性更改obj所指向的atomic\_flag状态，通过清除它（设为false）。第一版本根据memory\_order\_seq\_cst排序内存访问，第二版本根据order排序内存访问。

参数obj是指向volatile原子标志的指针，以接受非volatile和volatile（如内存映射I/O）的原子标志。

void atomic\_init( volatile A\* obj, C desired );

以值desired初始化默认构造的原子对象obj。此函数非原子操作，即使通过原子操作，来自另一线程的同时访问也会造成数据竞争。

这是为所有原子对象类型 A 定义的泛型函数。该参数为指向 volatile 原子类型的指针，以接受非 volatile 和 volatile （例如内存映射 I/O ）原子对象。 C 是与 A 对应的非原子类型。

\_Bool atomic\_is\_lock\_free( const volatile A\* obj );

确定所有A类型对象（obj所指向对象的类型）上的原子操作是否为无锁。在任何给定程序执行中，调用atomic\_is\_lock\_free的结果对于所有同一类型的指针相同。

这是为所有原子对象类型 A 定义的泛型函数。该参数为指向 volatile 原子类型的指针，以接受非 volatile 和 volatile （例如内存映射 I/O ）原子对象。

void atomic\_store( volatile A\* obj , C desired);(1)

void atomic\_store\_explicit( volatile A\* obj, C desired, memory\_order order );(2)

原子地以desired替换obj所指向的原子对象的值。此操作是原子写操作。

第一版本根据memory\_order\_seq\_cst排序内存访问，第二版本根据order排序内存访问。order必须是memory\_order\_relaxed、 memory\_order\_release或memory\_order\_seq\_cst之一。否则为未定义行为。

这是为所有原子对象类型 A 定义的泛型函数。该参数为指向 volatile 原子类型的指针，以接受非 volatile 和 volatile （例如内存映射 I/O ）原子对象。 C 是与 A 对应的非原子类型。

C atomic\_load( const volatile A\* obj );

C atomic\_load\_explicit( const volatile A\* obj, memory\_order order );(2)

原子性加载并返回obj所指向的原子对象的当前值。该操作是原子读操作。

第一版本根据memory\_order\_seq\_cst排序内存访问，第二版本根据order排序内存访问。order必须是memory\_order\_relaxed、 memory\_order\_consume、 memory\_order\_acquire或memory\_order\_seq\_cst之一。否则行为未定义。

这是为所有原子对象类型 A 定义的泛型函数。该参数为指向 volatile 原子类型的指针，以接受非 volatile 和 volatile （例如内存映射 I/O ）原子对象。 C 是与 A 对应的非原子类型。

C atomic\_exchange( volatile A\* obj, C desired );(1)

C atomic\_exchange\_explicit( volatile A\* obj, C desired, memory\_order order );(2)

原子地以desired替换obj所指向的对象的值，并返回obj先前所保有的值,但不改变desired。此操作是读-修改-写操作。第一版本根据memory\_order\_seq\_cst排序内存访问，第二版本根据order排序内存访问。

这是为所有原子对象类型 A 定义的泛型函数。该参数为指向 volatile 原子类型的指针，以接受非 volatile 和 volatile （例如内存映射 I/O ）原子对象。 C 是与 A 对应的非原子类型。.

\_Bool atomic\_compare\_exchange\_strong( volatile A\* obj,C\* expected, C desired );(1)

\_Bool atomic\_compare\_exchange\_weak( volatile A \*obj, C\* expected, C desired );(2)

\_Bool atomic\_compare\_exchange\_strong\_explicit( volatile A\* obj, C\* expected, C desired,

memory\_order succ, memory\_order fail );(3)

\_Bool atomic\_compare\_exchange\_weak\_explicit( volatile A \*obj, C\* expected, C desired,

memory\_order succ, memory\_order fail );(4)

原子性比较obj所指向的值和expected所指向的值，若它们相等，则以desired替换前者（进行读-修改-写操作）。否则，将obj所指向的实际值载入\*expected（进行加载操作）。相等返回true，不等返回false。

读-修改-写和加载操作的内存顺序各自为succ和fail。(1-2)版本默认使用memory\_order\_seq\_cst。

函数的弱形式（(2)与(4)）允许虚假失败，即表现如同即使二者相等也有\*obj != \*expected。当比较与交换在循环中时，某些平台上弱版本将会产出更好的性能。在弱的比较与交换要求循环而强版本不要求时，最好用强版本。

这是为所有原子对象类型 A 定义的泛型函数。该参数为指向 volatile 原子类型的指针，以接受非 volatile 和 volatile （例如内存映射 I/O ）原子对象。 C 是与 A 对应的非原子类型。

C atomic\_fetch\_add/sub/or/xor/and( volatile A\* obj, M arg )(1)

C atomic\_fetch\_add/sub/or/xor/and \_explicit( volatile A\* obj, M arg, memory\_order order );(2)

以arg和obj的旧值的加法/减法/逻辑或/逻辑异或/逻辑与的结果原子性替换obj所指向的值，并返回obj先前保有的值。此操作是读-修改-写操作。第一版本根据memory\_order\_seq\_cst排序内存访问，第二版本根据order排序内存访问。

这是为所有原子对象类型 A 定义的泛型函数。该参数为指向 volatile 原子类型的指针，以接受非 volatile 和 volatile （例如内存映射 I/O ）原子对象。若 A 为原子整数类型，则 M 是与 A 对应的非原子类型；若 A 为原子指针类型，则 M 为 ptrdiff\_t 。

对于有符号整数类型，算术定义为使用补码表示。没有未定义结果。对于指针类型，结果可能是未定义地址，不过此外的操作没有未定义行为。

void atomic\_thread\_fence( memory\_order order );

建立非原子和宽松原子访问的以order指示的内存顺序，而无关联的原子操作。例如，线程A中发生先于memory\_order\_release栅栏的所有非原子和宽松原子访问，将同步于与线程B中发生后于memory\_order\_acquire栅栏的来自同一位置的非原子和宽松原子加载。

void atomic\_signal\_fence( memory\_order order );

在线程和同一线程上执行的信号处理函数间，建立非原子和宽松原子访问的以order指示的内存同步顺序。这等价于atomic\_thread\_fence，除了不为内存顺序放出CPU指令。只有编译器进行的指令重排会被压制作order指示。例如，带释放语义的栅栏阻止读或写被移到后继的写入之后，而带获得语义的栅栏阻止读或写被移动到前接读取之前。

### 4.4 原子数据类型

|  |  |  |  |
| --- | --- | --- | --- |
| typedef name | Full type name | typedef name | Full type name |
| atomic\_bool | \_Atomic \_Bool | atomic\_uint\_least16\_t | \_Atomic [uint\_least16\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_char | \_Atomic char | atomic\_int\_least32\_t | \_Atomic [int\_least32\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_schar | \_Atomic signed char | atomic\_uint\_least32\_t | \_Atomic [uint\_least32\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_uchar | \_Atomic unsigned char | atomic\_int\_least64\_t | \_Atomic [int\_least64\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_short | \_Atomic short | atomic\_uint\_least64\_t | \_Atomic [uint\_least64\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_ushort | \_Atomic unsigned short | atomic\_int\_fast8\_t | \_Atomic [int\_fast8\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_int | \_Atomic int | atomic\_uint\_fast8\_t | \_Atomic [uint\_fast8\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_uint | \_Atomic unsigned int | atomic\_int\_fast16\_t | \_Atomic [int\_fast16\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_long | \_Atomic long | atomic\_uint\_fast16\_t | \_Atomic [uint\_fast16\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_ulong | \_Atomic unsigned long | atomic\_int\_fast32\_t | \_Atomic [int\_fast32\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_llong | \_Atomic long long | atomic\_uint\_fast32\_t | \_Atomic [uint\_fast32\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_ullong | \_Atomic unsigned long long | atomic\_int\_fast64\_t | \_Atomic [int\_fast64\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_char16\_t | \_Atomic char16\_t | atomic\_uint\_fast64\_t | \_Atomic [uint\_fast64\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_char32\_t | \_Atomic char32\_t | atomic\_intptr\_t | \_Atomic [intptr\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_wchar\_t | \_Atomic wchar\_t | atomic\_uintptr\_t | \_Atomic [uintptr\_t](http://en.cppreference.com/w/c/types/integer) |
| atomic\_int\_least8\_t | \_Atomic [int\_least8\_t](http://en.cppreference.com/w/c/types/integer) | atomic\_size\_t | \_Atomic [size\_t](http://en.cppreference.com/w/c/types/size_t) |
| atomic\_uint\_least8\_t | \_Atomic [uint\_least8\_t](http://en.cppreference.com/w/c/types/integer) | atomic\_ptrdiff\_t | \_Atomic [ptrdiff\_t](http://en.cppreference.com/w/c/types/ptrdiff_t) |
| atomic\_int\_least16\_t | \_Atomic [int\_least16\_t](http://en.cppreference.com/w/c/types/integer) | atomic\_intmax\_t | \_Atomic [intmax\_t](http://en.cppreference.com/w/c/types/integer) |
|  |  | atomic\_uintmax\_t | \_Atomic [uintmax\_t](http://en.cppreference.com/w/c/types/integer) |

## 5 固定宽度整数类型及最值—stdint.h

该头文件的主要作用是统一不同平台下同一数据类型的长度。在Linux中其中每种类型都是由：

typedef \_\_类型名大写\_TYPE\_\_ 类型名小写

定义而成。

## 6 随机数

rand()产生0-最大范围RAND\_MAX的随机数,若在调用rand()之前调用 srand（time(NULL)），则每次生成的随机数都不一样。调用time函数应当包含time.h头文件。

▲要取得[a,b)的随机整数，使用a+rand() % (b-a)（结果值含a不含b）。

▲要取得[a,b]的随机整数，使用a+rand() % (b-a+1) （结果值含a和b）。

▲要取得(a,b]的随机整数，使用a+(rand() % (b-a) + a + 1 （结果值不含a含b）。

▲即（通用公式:a + rand() % n；取得[a,a+n) 的随机整数，其中的a是起始值，n是整数的范围。）

▲要取得[a,b) 的随机整数，另一种表示：a + (int)(b-a) \* rand() / (RAND\_MAX + 1)。

▲要取得[a,b] 的随机整数 另一种表示：a + (int)(b-a) \* rand() / (RAND\_MAX )。

▲要取得[0,a] 之间的浮点数 ，可以使用rand() / double(RAND\_MAX/a)。

## 7 其它

## 7.1 快速排序

#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num从1开始计数,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。如对int a[100]排序：

int comp(const void\*a,const void\*b)

return \*(int\*)a-\*(int\*)b;

注意：qsort传给cmp函数的参数是指向数组元素的指针。const void\*a,const void\*b可自己根据需要改变类型。因而数组元素是变量，定义为 1级指针，数组元素是一级指针，则定义为2级指针，数组元素是二级指针，强定义为三级指针。

对于链表来说，由于移动元素复杂， 因而可以对指向各节点的指针进行排序。

## 7.2 头文件处理

可以自己写头文件，头文件以“.h”为扩展名，若放在库中用#include<文件名>来引用，若放在工程下以#include”文件名”来引用。头文件的作用就是编译的时候将头文件的内容来覆盖#include的位置。头文件中可以定义变量,函数，等，但不推荐。因为程序编译是以源文件为单位的，只有当头文件被包含在某个源文件内才参与编译。一个源文件可能包含多个头文件，这些头文件可能都包含了某个头文件“\*.h”，那么这个头文件必须用预处理语句#ifdef，#ifndef, #define, #endif 防止被重复包含，否则这个源文件内就会包含多个该头文件“\*.h”，假设该头文件包含一个变量或者函数定义，那么在该源文件内就存在多个同样内容的定义，这样就会造成重复定义。因而在平常时候，我们不要将函数定义在头文件中。只在头文件中进行声明，在源文件中实现。这样对于大型工程，有助于代码文件的有效管理。所以.h文件中一般只能包含全局变量的声明，函数声明，宏定义一类的，在.h文件中定义变量是不被推荐的。

下面给一个#ifndef/#define/#endif的格式：

#ifndef A\_H //意思是"if not define a.h" 如果不存在a.h

#define A\_H //就引入a.h

#include <math.h> // 引用标准库的头文件

#include “header.h” // 引用非标准库的头文件

void Function1(…); // 全局函数声明

#endif //否则不需要引入

## 6.3 time函数

#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。返回的时距离1970年0时0分0分的时间间隔。

#include <time.h>

clock\_t ( 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.) 返回值应该是cpu的时钟。