C&C++知识点

# 时间和日期处理

<time.h>中定义了库函数得到日期和时间的各种格式。

## time()

time函数返回一个time\_t类型的值(通常是long)，它是由实现决定的对当前日期和时间的编码。可以把这个值传递给其他函数，比如localtime，得到更明确的信息。

time\_t tval = time(&NULL);

等价于

time\_t tval;

time(&tval);

## localtime()

localtime函数把编码的时间分解成下面的部分：

struct tm {

int tm\_sec; /\* 秒 - [0,59] \*/

int tm\_min; /\* 分钟 - [0,59] \*/

int tm\_hour; /\* 一天的第几个小时 - [0,23] \*/

int tm\_mday; /\* 一个月的第几天 - [1,31] \*/

int tm\_mon; /\* 从1月份开始的月份 - [0,11] \*/

int tm\_year; /\* 从1900年开始的年 \*/

int tm\_wday; /\* 从周日开始的一个星期的第几天 - [0,6] \*/

int tm\_yday; /\* 从1月1日开始的一年中的第几天 - [0,365] \*/

int tm\_isdst; /\* 白天的保存是否有效 \*/

};

localtime函数返回一个指向静态结构的指针，因此在没有复制时程序中每次只有一个这样的结构可以使用。

timt\_t tval = time(&NULL);

tm\* now = localtime(&tval);

## ctime()

ctime函数返回一个指向静态字符串的指针，该字符串以标准的格式描述完整的时间和日期并以换行符结束。

time\_t tval;

…

printf(“date: %s\n”, ctime(&tval)); // Mon Dec 04 18:53:02 2017

## strftime()

strftime函数按照用户规范格式化一个字符串。格式描述符如下：

|  |  |
| --- | --- |
| %A，%a | 完整的/缩写的星期名(Saturday/Sat) |
| %B，%b | 完整的/缩写的月份名(April/Apr) |
| %c | 日期和时间(Apr 04 18:48：03 2017) |
| %d | 一个月的第几天(04) |
| %H | 小时(24小时制：18) |
| %I | 小时(12小时制：06) |
| %j | 一年中的几天(102) |
| %m | 月份(从一月开始：12) |
| %M | 1小时的第几分钟(47) |
| %p | AM/PM(PM) |
| %s | 1分钟的第几秒(03) |
| %U | 一年的第几个星期(以星期日为一个星期的开始) |
| %w | 星期几(星期日是0) |
| %W | 一年的第几个星期(以星期一为一个星期的开始) |
| %x | 日期(Apr 12 2017) |
| %X | 时间(12:38:03) |
| %y | 一个世纪的第几年(117) |
| %Y | 年(2017) |
| %Z | 时区(如果支持的话) |

char buf[BUFSIZ];

char\* format = “%A, %B %d, day %j of %Y, %I:%M %p”;

struct tm\* now;

…

strftime(buf, sizeof buf, format, now); // Monday, December 04, day 338 of 2017, 06:53 PM

## mktime()

mktime函数改变一个tm结构以使日期值和时间值都在适当的范围内，之后它更新了星期的天数域(tm\_wday)和年的天数域(tm\_yday)。如果所指定的日期不能被表示，则mktime函数将会失败。当正被讨论的日期先于实现的参考日期时，会发生这种情况。

struct tm\* now;

…

now->tm\_mday += 10;

if(mktime(now) != 1)

{ return EXIT\_FAILURE; }

## asctime()

asctime函数返回在传递的tm结构中代表时间的标准字符串，所以ctime(&tval)等于asctime(localtime(&tval))。

struct tm\* now;

…

printf(“New date: %s”, asctime(now)); // Mon Dec 04 18:53:02 2017

## difftime()

difftime函数以double型返回两个编码中相差的秒数。

time\_t start, stop;

…

difftime(stop, start);

# 动态内存管理

## 标准C中使用堆

头文件<stdlib.h>为使用动态内存声明了4个函数：

void\* malloc(size\_t siz);

返回一个指向siz的第一个字节的指针。通常用于分配单个对象。

void\* calloc(size\_t nelems, size\_t elem\_size);

返回指向nelems \* elem\_size字节的指针，并被初始化为0。通常用于分配对象数组。

void\* realloc(void\* ptr, size\_t siz);

用于扩展或缩小堆栈的分配。除非ptr是NULL指针，否则ptr指针必须源于上次对malloc、calloc、realloc的调用，其结果和malloc(siz)相同。如果有足够的空间用于新的分配，realloc就把原始数据拷贝到新分配的内存中，然后返回这个新的地址。如果ptr不是NULL，而siz是0时，那么realloc就和free的作用一样了。

void free(void\* ptr);

使以前分配的堆栈内存区可以重用。前一次的调用一定会返回ptr指向上述之一的内存分配函数。调用free(NULL)是合法的，但它不起任何作用。

## 浅拷贝和深拷贝

一般来说，当一个对象包含一个指向堆内存的指针时，应该定义一个拷贝构造函数和赋值运算符，以及所需的析构函数和其他的构造函数。

## 处理内存分配失败

C++规定内存分配失败将会导致一个bad\_alloc异常。任何程序应该为处理bad\_alloc异常做好准备：

catch(const bad\_alloc& x)

{

cerr << “out of memory: ” << x.what() << endl;

abort();

}

如果偏爱经典的返回空指针的行为，可以使用new运算符的特殊形式

#include <new>

T\* tp = new (nothrow) T;

if(tp)

// 使用tp…

另一种处理内存不足的方法是取代其本身的部分内存分配机制。当内存分配失败时，C++就会调用缺省的new\_handler，它将按顺序抛出bad\_alloc异常。可以通过把处理函数的地址传递到set\_new\_handler函数来建立自己的处理函数。

inline void my\_handler()

{

cout << “Memory exhausted” << endl;

abort();

}

int main()

{

set\_new\_handler(my\_handler);

…

}

## 重载new和delete

标准C++库定义了12个用于分配和释放内存的函数。这12个函数是：

// 标量形式

void\* operator new(size\_t);

void\* operator new(size\_t, const nothrow\_t);

void operator delete(void\*);

void operator delete(void\*, const nothrow\_t);

// 数组形式

void\* operator new[](size\_t);

void\* operator new[](size\_t, const nothrow\_t);

void operator delete[](void\*);

void operator delete[](void\*, const nothrow\_t);

// 配置new

void\* operator new(size\_t, void\*);

void\* operator new[](size\_t, void\*);

void operator delete(void\*, void\*);

void operator delete[](void\*, void\*);

当用与上面函数的前8个函数相同的形式定义的任何函数时，它将在程序中替换该函数的库函数版本。后面4个配置形式都属于重载，它们不像其他的函数那样可以被替代。

## 配置new

有时需要在一个预定的地址上创建一个对象。这个地址的位置可能在某个设备映射的RAM区域中，或者在一个特殊类的堆中。可以为new运算符构建一个使用配置语法的对象。例如为了在地址p构建T，可以像下面这样做：

#include <new>

T\* tp = new (p) T;

在这种情况下，由于内存的位置已经定下来了，因此没有任何形式的operator new执行。实际上，配置operator new的默认版本忽略了它的size参数，而只是返回其地址参数

void\* operator new(size\_t, void\* p)

{

return p;

}

可以使用配置new来使赋值运算符和它的拷贝构造函数同步，而不用复制初始化new的代码。如下所示：

T& T::operator ==(const T& x)

{

if(this != &x)

{

this->T::~T();

new (this) T(x);

}

return \*this;

}