C11 标准新特性

## 对齐处理

alignof(T)返回T的对齐方式，aligned\_alloc()以指定字节和对齐方式分配内存，头文件<stdalign.h>定义了这些内容。

std::size\_t alignof(typename)

返回由类型标识所指示的类型的任何实例所要求的对齐字节数，该类型可以为完整类型、数组类型或者引用类型。 若类型为引用类型，则运算符返回被引用类型的对齐；若类型为数组类型，则返回元素类型的对齐要求。

sizeof和alignof对于一般的数据类型返回值是相同的，但是对于下面情况特别：

struct Foo

{

int a;

float b;

char c;

};

alignof(Foo) *//值为4，对齐长度*

sizeof(Foo) *//结构体的总大小：12*

分配 size 字节未初始化的存储空间，按照 alignment 指定对齐。 size 参数必须是 alignment 的整数倍：

void \*aligned\_alloc(size\_t alignment, size\_t size);

## \_Noreturn修饰符

\_Noreturn是个函数修饰符，位置在函数返回类型的前面，声明函数无返回值，有点类似于gcc的**attribute**((noreturn))，后者在声明语句尾部：

1. \_Noreturn 关键词出现于函数声明中，指定函数不会由于执行到 return 语句或抵达函数体结尾而返回（可通过执行 longjmp 返回）；
2. 若声明 \_Noreturn 的函数返回，则行为未定义。若编译器能检测此错误，则推荐编译器诊断；
3. 此指定符通常通过便利宏 noreturn 使用，该宏于头文件 stdnoreturn.h 提供。

#include <stdlib.h>

#include <stdio.h>

#include <stdnoreturn.h>

// 在 i <= 0 时导致未定义行为

// 在 i > 0 时退出

noreturn void stop\_now(int i) // 或 \_Noreturn void stop\_now(int i)

{

if (i > 0) exit(i);

}

int main(void)

{

puts("Preparing to stop...");

stop\_now(2);

puts("This code is never executed.");

}

## \_Generic

\_Generic支持轻量级范型编程，可以把一组具有不同类型而却有相同功能的函数抽象为一个接口：

\_Generic (controlling-expression , association-list)

其中 association-list 是逗号分隔的关联列表，每个关联拥有语法：：

type-name : expression

default : expression

“\_Generic”这个关键字类似于switch语法：

\_Generic(‘a’, char: 1, int: 2, long: 3, default: 0);

输出为2 (字符在C中为整型)。

## \_Static\_assert()

\_Static\_assert()，静态断言，在编译时刻进行，断言表达式必须是在编译时期可以计算的表达式，而普通的assert()在运行时刻断言：

\_Static\_assert(expression, string message);

示例：

#include <assert.h>

int main(void)

{

// 测试数学是否正常工作

static\_assert(2 + 2 == 4, "Whoa dude!"); // 或 \_Static\_assert(...

// 这会在编译时产生错误。

static\_assert(sizeof(int) < sizeof(char), "this program requires that int is less than char");

}

## 安全版本的几个函数

主要牵涉到字符串和内存这一块，目的是为了防止缓冲区溢出攻击，如：

char\* gets\_s(char \*str, rsize\_t n);

取代了：

char\* gets(char \*str);

如果在Visual Studio中使用这些函数，一般会提示要么使用宏：

\_CRT\_SECURE\_NO\_WARNINGS

禁用警告，要么使用具体对应的安全版本的函数取代，所以只需要看编译器给出的提示就行了。

## fopen()新模式

函数增加了新的创建、打开模式“x”，在文件锁中比较常用：以x结尾的模式为独占模式，文件已存在或者无法创建(一般都是路径不正确)都会导致fopen失败，文件以操作系统支持的独占模式打开。

## 匿名结构体、联合体

在 C 语言中，可以在结构体中声明某个联合体（或结构体）而不用指出它的名字，如此之后就可以像使用结构体成员一样直接使用其中联合体（或结构体）的成员：

struct person

{

char \*name;

char gender;

int age;

int weight;

struct

{

int area\_code;

long phone\_number;

};

};

等价于：

struct phone

{

int area\_code;

long phone\_number;

};

struct person

{

char \*name;

char gender;

int age;

int weight;

struct phone office;

};

现在假设定义了一个person类型的对象p，那么访问他的电话区域码可以通过：

p.area\_code

取代：

p.office.area\_code

## 多线程

头文件<threads.h>定义了创建和管理线程的函数，新的存储类修饰符\_Thread\_local限定了变量不能在多线程之间共享。

\_Thread\_local 指示线程存储期。它不能用于函数声明。若将它用在对象声明上，则它必须在同一对象的每次声明上都存在。若将它用在块作用域声明上，则必须与 static 或 extern 之一组合以决定链接。

线程存储期：存储期是创建对象的线程的整个执行过程，在启动线程时初始化存储于对象的值。每个线程拥有其自身的相异对象。若执行访问此对象的表达式的线程，不是执行其初始化的线程，则行为是实现定义的。所有声明为 \_Thread\_local 的对象拥有此存储期。

## \_Atomic类型修饰符和头文件<stdatomic.h>

语法：

\_Atomic(type-name)

\_Atomic type-name

用作类型限定符，指代type-name的原子版本。在此作用中，它可以与 const，volatile及restrict混合使用。尽管不同于其他限定符，type-name的原子版本可能拥有不同的大小、对齐以及对象表示：

\_Atomic const int \* p1; // p1 是指向 \_Atomic const int 的指针

const atomic\_int \* p2; // 同上

const \_Atomic(int) \* p3; // 同上

原子类型的对象是仅有的免除数据竞争的对象，即它们可以被两个线程共时修改，或先被一个修改再被另一个读取。通过原自变量，可以实现基于无锁队列的生产者-消费者模型。

每个原子对象都拥有关联于其自身的修改顺序，即对该对象的完整修改顺序。若从某个线程的视角来看，对于某原子对象M的修改A发生先于同一原子对象M的修改B，则在M的修改顺序中A的出现先于B。

注意即使每个原子对象都有其自身的修改顺序，它却不是全序；不同线程可能会观测到相异原子对象有相异的修改顺序。

对于所有原子运算，保证有四种连贯：

1. 写写连贯：若原子对象M的修改操作A发生先于M的修改操作B，则M的修改顺序中A出现早于B;
2. 读读连贯：若原子对象M的值计算A发生先于M的值计算B，且从M上的副效应X求得A值，则B所计算得的值要么是X所存储的值，要么是M上的副效应Y所存储的值，其中Y在M的修改顺序中出现后于X；
3. 读写连贯：若原子对象M的值计算A发生先于M上的操作B，则从M上的副效应X求得A值，这里X在M的修改顺序中出现先于B；
4. 写读连贯：若在原子对象M上的副效应X发生先于M的值计算B，则求值B从X，或从在M的修改顺序中出现后于X的副效应Y求得其值

一些原子运算亦是同步操作：它们可以拥有附加的释放语义、获取语义，或顺序一致语义。见memory\_order。内建的自增减运算符和复合赋值运算符是拥有完全序列一致顺序（如同用memory\_order\_seq\_cst）的读-修改-写操作。若想要更不严格的同步语义，则可以用标准库函数替代。

原子属性仅对左值表达式有意义。左值到右值转换（其模仿从原子区域到CPU寄存器的内存读取）会把原子性及其他限定符剥去。示例：

#include <stdio.h>

#include <threads.h>

#include <stdatomic.h>

atomic\_int acnt;

int cnt;

int f(void\* thr\_data)

{

for(int n = 0; n < 1000; ++n) {

++cnt;

++acnt;

// 对于此例，宽松内存顺序是足够的，例如

// atomic\_fetch\_add\_explicit(&acnt, 1, memory\_order\_relaxed);

}

return 0;

}

int main(void)

{

thrd\_t thr[10];

for(int n = 0; n < 10; ++n)

thrd\_create(&thr[n], f, NULL);

for(int n = 0; n < 10; ++n)

thrd\_join(thr[n], NULL);

printf("The atomic counter is %u\n", acnt);

printf("The non-atomic counter is %u\n", cnt);

}

可能的输出：

The atomic counter is 10000

The non-atomic counter is 8644

## 改进的Unicode支持和头文件<uchar.h>

语法：

”s-char-sequence”

u8 ”s-char-sequence”

u ”s-char-sequence”

U ”s-char-sequence”

L ”s-char-sequence”

该语句用于建立一个指向double的指针fp，且该指针指向这个3元素数组的第一个元素。 在文件域内建立的复合赋值只在程序的整个生存期内有效。在模块内建立的复合赋值是局部对象，在退出模块后不再存在。

1. 字符串字面量：字面量类型为char[]，用执行字符集从s-char-sequence中的下个字符初始化数组中的每个字符；
2. UTF-8字符串字面量：字面量类型为char[]，用UTF-8编码，从s-char-sequence中的下个多字节字符初始化字符数组中的每个字符；
3. 16位宽字符串字面量：字面量类型为char16\_t[]，如同在实现定义的本地环境中通过执行mbrtoc16一般初始化数组中的每个char16\_t元素；
4. 32位宽字符串字面量：字面量类型为char32\_t[]，如同在实现定义的本地环境中通过执行mbrtoc32一般初始化数组中的每个char32\_t元素；
5. 宽字符串字面量：字面量类型为wchar\_t[]，如同在实现定义的本地环境中通过执行mbstowcs一般初始化数组中的每个wchar\_t元素。

示例：

#include <stdio.h>

#include <stdlib.h>

#include <stddef.h>

#include <uchar.h>

#include <locale.h>

int main(void)

{

char s1[] = "a猫?"; // 或 "a\u732B\U0001F34C"

char s2[] = u8"a猫?";

char16\_t s3[] = u"a猫?";

char32\_t s4[] = U"a猫?";

wchar\_t s5[] = L"a猫?";

setlocale(LC\_ALL, "en\_US.utf8");

printf(" \"%s\" is a char[%zu] holding { ", s1, sizeof s1 / sizeof \*s1);

for(size\_t n = 0; n < sizeof s1 / sizeof \*s1; ++n)

printf("%#x ", +(unsigned char)s1[n]);

puts(" }");

printf("u8\"%s\" is a char[%zu] holding { ", s2, sizeof s2 / sizeof \*s2);

for(size\_t n = 0; n < sizeof s2 / sizeof \*s2; ++n)

printf("%#x ", +(unsigned char)s2[n]);

puts(" }");

printf(" u\"a猫?\" is a char16\_t[%zu] holding { ", sizeof s3 / sizeof \*s3);

for(size\_t n = 0; n < sizeof s3 / sizeof \*s3; ++n)

printf("%#x ", s3[n]);

puts(" }");

printf(" U\"a猫?\" is a char32\_t[%zu] holding { ", sizeof s4 / sizeof \*s4);

for(size\_t n = 0; n < sizeof s4 / sizeof \*s4; ++n)

printf("%#x ", s4[n]);

puts(" }");

printf(" L\"%ls\" is a wchar\_t[%zu] holding { ", s5, sizeof s5 / sizeof \*s5);

for(size\_t n = 0; n < sizeof s5 / sizeof \*s5; ++n)

printf("%#x ", (unsigned)s5[n]);

puts(" }");

}

可能的结果：

"a猫?" is a char[9] holding { 0x61 0xe7 0x8c 0xab 0xf0 0x9f 0x8d 0x8c 0 }

u8"a猫?" is a char[9] holding { 0x61 0xe7 0x8c 0xab 0xf0 0x9f 0x8d 0x8c 0 }

u"a猫?" is a char16\_t[5] holding { 0x61 0x732b 0xd83c 0xdf4c 0 }

U"a猫?" is a char32\_t[4] holding { 0x61 0x732b 0x1f34c 0 }

L"a猫?" is a wchar\_t[4] holding { 0x61 0x732b 0x1f34c 0 }

## quick\_exit()

用于exit()函数执行失败的时候：

#include <stdlib.h>

#include <stdio.h>

void f1(void)

{

puts("pushed first");

fflush(stdout);

}

void f2(void)

{

puts("pushed second");

}

int main(void)

{

at\_quick\_exit(f1);

at\_quick\_exit(f2);

quick\_exit(0);

}

输出：

pushed second

pushed first

## 复数宏，浮点数宏

C头文件 <tgmath.h> 包含头文件 <math.h> 及 <complex.h> ，并定义了几种泛型宏。这些宏会根据参数类型决定要调用的实际函数：

#include <stdio.h>

#include <complex.h>

#include <tgmath.h>

int main(void)

{

double complex z1 = I \* I; // 虚数单位平方

printf("I \* I = %.1f%+.1fi\n", creal(z1), cimag(z1));

double complex z2 = pow(I, 2); // 虚数单位平方

printf("pow(I, 2) = %.1f%+.1fi\n", creal(z2), cimag(z2));

double PI = acos(-1);

double complex z3 = exp(I \* PI); // 欧拉公式

printf("exp(I\*PI) = %.1f%+.1fi\n", creal(z3), cimag(z3));

double complex z4 = 1+2\*I, z5 = 1-2\*I; // 共轭

printf("(1+2i)\*(1-2i) = %.1f%+.1fi\n", creal(z4\*z5), cimag(z4\*z5));

}

输出：

I \* I = -1.0+0.0i

pow(I, 2) = -1.0+0.0i

exp(I\*PI) = -1.0+0.0i

(1+2i)\*(1-2i) = 5.0+0.0i

## time.h新增timespec结构体，时间单位为纳秒，原来的timeval结构体时间单位为毫秒

struct timespec 定义：

typedef long time\_t;

#ifndef \_TIMESPEC

#define \_TIMESPEC

struct timespec {

time\_t tv\_sec; // seconds

long tv\_nsec; // and nanoseconds

};

#endif

struct timespec有两个成员，一个是秒，一个是纳秒, 所以最高精确度是纳秒。一般由函数：

int clock\_gettime(clockid\_t, struct timespec\*)

获取特定时钟的时间，常用如下4种时钟：

CLOCK\_REALTIME //系统当前时间，从1970年1.1日算起

CLOCK\_MONOTONIC // 系统的启动时间，不能被设置

CLOCK\_PROCESS\_CPUTIME\_ID // 本进程运行时间

CLOCK\_THREAD\_CPUTIME\_ID // 本线程运行时间

struct tm \*localtime(const time\_t \*clock); // 线程不安全

struct tm\* localtime\_r(const time\_t\* timer, struct tm\* result); // 线程安全

size\_t strftime(char\* ptr, size\_t maxsize, const char\* format, const struct tm\* timeptr);

struct timeval 定义：

struct timeval {

time\_t tv\_sec; // seconds

long tv\_usec; // microseconds

};

struct timezone{

int tz\_minuteswest; //miniutes west of Greenwich

int tz\_dsttime; //type of DST correction

};

struct timeval有两个成员，一个是秒，一个是微秒, 所以最高精确度是微秒。 一般由函数

int gettimeofday(struct timeval \*tv, struct timezone \*tz)

获取系统的时间，示例：

#include<stdio.h>

#include<time.h>

#include<sys/time.h>

void nowtime\_ns()

{

printf("---------------------------struct timespec---------------------------------------\n");

printf("[time(NULL)] : %ld\n", time(NULL));

struct timespec ts;

clock\_gettime(CLOCK\_REALTIME, &ts);

printf("clock\_gettime : tv\_sec=%ld, tv\_nsec=%ld\n", ts.tv\_sec, ts.tv\_nsec);

struct tm t;

char date\_time[64];

strftime(date\_time, sizeof(date\_time), "%Y-%m-%d %H:%M:%S", localtime\_r(&ts.tv\_sec, &t));

printf("clock\_gettime : date\_time=%s, tv\_nsec=%ld\n", date\_time, ts.tv\_nsec);

}

void nowtime\_us()

{

printf("---------------------------struct timeval----------------------------------------\n");

printf("[time(NULL)] : %ld\n", time(NULL));

struct timeval us;

gettimeofday(&us,NULL);

printf("gettimeofday: tv\_sec=%ld, tv\_usec=%ld\n", us.tv\_sec, us.tv\_usec);

struct tm t;

char date\_time[64];

strftime(date\_time, sizeof(date\_time), "%Y-%m-%d %H:%M:%S", localtime\_r(&us.tv\_sec, &t));

printf("gettimeofday: date\_time=%s, tv\_usec=%ld\n", date\_time, us.tv\_usec);

}

int main(int argc, char\* argv[])

{

nowtime\_ns();

printf("\n");

nowtime\_us();

printf("\n");

return 0;

}

执行结果：

—————————struct timespec—————————————

[time(NULL)] : 1400233995

clock\_gettime : tv\_sec=1400233995, tv\_nsec=828222000

clock\_gettime : date\_time=2014-05-16 17:53:15, tv\_nsec=828222000

—————————struct timeval—————————————-

[time(NULL)] : 1400233995

gettimeofday: tv\_sec=1400233995, tv\_usec=828342

gettimeofday: date\_time=2014-05-16 17:53:15, tv\_usec=828342

有关关键字或者接口的参照可以查看以下地址：

<http://zh.cppreference.com/w/%E9%A6%96%E9%A1%B5>