2016年，C语言该怎样写

文 / Matt Stancliff　译 / 贾子甲

本文所讲述的内容基于x86-64 Unix/Linux/POSIX编程模型，如果你在8位嵌入式系统，十年以上未更新的编译器，或者兼顾跨平台程序兼容的平台上写程序，则需要参考与这些平台特定建议，本文所介绍的经验不一定有效。

C语言的首要原则是——能不写C语言就不写。如果一定要写，请遵守现代规则。

C语言诞生于20世纪70年代初。人们在其发展的各个阶段都在“学习C语言”，但在学习C语言之后，知识往往停滞不前，从开始学习它的那年起积攒起不同观点。

很重要的一点是，用C语言开发，不要再继续认为“那是我在八零或者九零年代学习的东西”。

本文首先假定处于符合现代设计标准的平台上，并且不存在需要兼容过多遗留约束。我们当然不该仅因为某些公司拒绝更新其二十年前的老系统就完全遵守陈旧的标准。

准备

C99标准（C99和C11分别是指1999年和2011年诞生的C语言标准，显然C11标准比较新）。

clang（一种C语言编译器），默认情况

- clang使用C11的一种扩展版本（GNU C11模式），所以不需要多余的配置就可以适应现代特性。
- 如果你想使用标准的C11，需要指定-std=c11；如果你想使用标准C99，则指定-std=c99。
- 与gcc相比，clang编译文件速度更快。

gcc需要指定-std=c99或-std=c11

gcc构建源文件的速度一般慢于clang，但某些时候却能保证编码速度更快。性能比较和回归测试也非常重要。

gcc-5默认使GNU C11模型（与clang相同），但如果你确切地需要C11或C99，仍应指定-std=c11或-std=c99。

优化

-O2，-O3

- 通常想使用-O2，但有时也使用-O3。在两个级别下（通过编译器）分别进行测试并且保持最佳性能。

-Os

- -Os如果你关注缓存效率（本该如此），这个选项能帮上你。

警告

-wall -Wextra -pedantic

- 最新版本的编译器支持-Wpedantic，但为了向后兼容其也接受古老的-pedantic。

在测试过程中，应该在所有的平台上都添加-Werror和-Wshadow。

- 因为不同的平台、编译器和库会发出不同警

告，通过使用-Werror可更有针对性地部署生产资源。你或许并不想仅因为某个平台上从未见过的GCC版本在新的运行方式下时报错就毁掉用户的全部构建。

额外选择包括-Wstrict-overflow -fno-strict-aliasing。

- 要么指定-fno-strict-aliasing，要么就确保只以对象创建时的类型对其进行访问。因为许多C语言代码拥有跨类型的别名，当不能控制整个底层源代码树时，使用-fno-strict-aliasing是个不错的选择。

到目前为止，clang有时会对一些有效语法发出警告，所以需要添加-Wno-missing-field-initializers。GCC在4.7.0版本后修正了这些不必要的警告。

构建

编译单元

- 构建C程序项目的最常见方式是将每个C源文件分解成目标文件，最后将所有目标文件链接到一起。这个过程在增量开发中表现很好，但从性能和优化角度看，并非最优。因为在这种方式下编译器不能检测出跨文件的潜在优化之处。

LTO——链接时优化

- LTO通过使用中间表示方式对目标文件进行处理，因此解决了“跨编译单元源文件分析和优化问题”，所以source-aware优化可在链接时跨编译单元实现。
- LTO明显减缓了链接过程，但make-j会有所帮助。
- clang LTO（http://llvm.org/docs/LinkTimeOptimization.html）
- gcc LTO（https://gcc.gnu.org/onlinedocs/gccint/LTO-Overview.html）
- 到2016年为止，clang和gcc都可通过在目标文件编译和最后库/程序链接时，向命令行选项中添加-flto来支持LTO。
- 不过LTO仍需监管。有时，程序中一些代码没有被直接使用，而是被附加的库使用了。因为LTO会在全局性链接时检测出没有使用或者不可访问的代码，也可能将其删除，所以这些代码将不会包含到最后的链接结果中。

架构

-march = native

- 允许编译器使用CPU完整的特性集。
- 再一次，性能测试和回归测试都非常重要（之后在多个编译器以及多个版本中对结果进行比较）以确保任何启用的优化都不会产生副作用。

如果你使用not-your-build-machine特性，-msse2和-msse4.2可能起到作用。

编写代码

类型

如果你将char、int、short、long或unsigned类型写入新代码中，就把错了。

对于现代程序，应该先输入#include<stdint.h>，之后再使用标准类型。

更多细节，参见“stdint.h规范”（http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/stdint.h.html）。

常见的标准类型有：

- int8_t，int16_t，int32_t，int64_t——符号数
- uint8_t，uint16_t，uint32_t，uint64_t——无符号数
- float——标准32位浮点数
- double——标准64位浮点数

到底用不用int

一些读者说他们真的非常喜爱int类型，但我们不得不把它从他们僵硬的手指中撬出来。在技术层面我想指出的是，如果在你的控制下类型的位数发生了改变，这时还希望程序正确运行是不可能的。

当然你也可以找出inttypes.h文件中的RATIONALE

（http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/inttypes.h.html#tag_13_19_06）来看看为何使用长度不固定的类型是不安全的。

如果你够聪明到可以在平台上对整个开发过程中的int类型统一定为16或32位，并且也在每个使用了它的地方测试所有16位和32位边界情况，那就可以随意使用int类型了。

对于我们中不能在编程时将多层决策树规范层次清晰地记住的人，可以使用固定位数的类型，这样在编程时就可以不必忧心那些麻烦的概念和测试开销。

或者，规范中更加简洁地指出：“ISO C语言标准中整数扩展规则将会造成意外改变。”

所以当使用int时，祝你好运！

使用char的例外情况

到2016年，char类型唯一的用处就是旧API中需要（比如strncat，printf中）。或者初始化一个只读字符串（比如const char * hello = "hello";），这是因为C语言字符串类型字面值（"hello"）是char []。

同样，在C11中，我们也有原生的unicode支持，并且UTF-8的字符串类型仍然是char [],甚至是像const char * abcgrr =u8"abc😊"; 这样的多字节序列也一样。

使用int、long等的例外情况

如果你正在使用带有原生返回类型或原生参数的函数，那么请使用函数原型或者API规范中描述的类型。

符号

无论何时都不应该将unsigned这个词写进代码。丑陋的C语言规定往往会因多字类型而损坏了程序可读性和使用性，而现在我们编写的代码不再需要遵守那些规定了。当能够输入uint64_t时又有谁想用unsigned long long int呢？<stdint.h>中的类型更加明确，含义更加确切，也能够更好地表达意义，并且具有更好的排版，提高了程序的使用性和可读性。

指针作为整型

但你可能会说“当遇到麻烦的指针数学时，我不得不将指针转换为long！”。这是错误的。

正确的指针数学类型是使用<stdint.h>中定义的unitptr_t，当然同样非常有用的ptrdiff_t也定义在stddef.h中。

而并非：

```
long diff = (long)ptrOld - (long)ptrNew;
```

你可以使用：

```
ptrdiff_t diff = (uintptr_t)ptrOld -
(uintptr_t)ptrNew;
```

也可以用：

```
printf("%p is unaligned by %" PRIuPTR
" bytes.\n", (void *)p, ((uintptr_t)
somePtr & (sizeof(void *) - 1)));
```

系统依赖的类型

你继续争辩：“在32位平台上我想要32位字长，而在64位平台上我要64位字长。”

这时思路将故意引入使用平台依赖的两种不同字长，会对编程造成困难。但是如果跳过这个问题，你仍然不想使用依赖系统的类型。

在此情况下，该使用intptr_t——当前平台定义为字长的整型。在32位平台上，intptr_t = int32_t；64位平台上，intptr_t = int64_t。intptr_t也有uintptr_t的形式。

对管理指针的偏移来说，我们还有ptrdiff_t，该类型适于存储指针减少的数值。

最大值容器

你需要一种能承载系统中所有可能的整数的类型吗？在这种情况下，人们倾向于使用已知的最大类型，比如将较小的无符号类型转换成uint64_t。从实际技术上讲，有一种更正确的方式来保证可以装载任何值。

对于所有整数来说，最安全的容器是intmax_t（或者uintmax_t）。可以无精度损失地将任何符号整数（或无符号整数）赋值或者转换为这种

形式。

其他类型

被广泛使用的系统依赖类型是stdded.h中提供的size_t。它在程序中非常基础，因为它是“能够容纳最大的数组索引的整数”，这也意味着它能容纳最大的内存偏移。在实际应用中，size_t是运算符sizeof的返回类型。

size_t实际上与uintptr_t一样都是定义在所有现代平台上的。所以在32位的平台上，size_t = uint32_t，而在64位平台上，size_t = uint64_t。

还有一个带有符号的size_t，也就是ssize_t，它用来表示在库函数出现错误时返回-1（注意，sszie_t是POSIX，不适用于Windows接口）。

所以，难道你不该在自己的函数参数中使用size_t来适应任何字长的系统么？从技术上讲，它是sizeof的返回值，所以任何接受多字节数值的函数都允许表示成size_t类型。

其他用途包括：size_t类型可作为malloc的参数，并且是read()和write()的返回类型（除了Windows平台上ssize_t不存在，其返回值是int类型）。

输出类型

你不应在输出时进行类型转换，而是永远使用inttypes.h中定义的合适的类型说明符。

包括但不限于以下情况：

- size_t——%zu
- ssize_t——%zd
- ptrdiff_t——%td
- 原始指针值%p（现代编译器中输出16进制数；首先将指针转换为(void *)）
- int64_t——"%" PRId64
- uint64_t——"%" PRIu64（64位类型应该只能使用PRI[udixXo]64风格的宏）
- intptr_t——"%" PRIdPTR
- uintptr_t——"%" PRIuPTR
- intmax_t——"%" PRIdMAX
- uintmax_t——"%" PRIuMAX

关于PRI* 格式化说明符需要注意：它们都是宏和宏在特定的平台基础上为了合适的输出而做的扩展。这意味着你不能这样：

```
printf("Local number: %PRIdPTR\n\n",
someIntPtr);
```

并且因为它们是宏，应该：

```
printf("Local number: %" PRIdPTR "\n\n",
someIntPtr);
```

注意应将%放进格式化字符串内，但类型指示符在格式化字符串外部，这是因为所有相邻的字符串都被预处理器连接成为最后的结合字符串。

C99允许在任何地方进行变量声明

所以，千万不要这样做：

```
void test(uint8_t input) {
    uint32_t b;
    if (input > 3) {
        return;
    }
    b = input;
}
```

而是这样：

```
void test(uint8_t input) {
    if (input > 3) {
        return;
    }
    uint32_t b = input;
}
```

警告：如果存在紧密的循环，请测试初始值的配置。有时分散的声明会造成意外减速。对于没有选择最快路径的代码（事情往往这样），最好还是尽可能清晰。而且，与初始化一起定义类型会极大提升可读性。

C99允许在 for 循环中声明内联计数器

所以不要这样：

```
uint32_t i;
for (i = 0; i < 10; i++)
```

而应该：

```
for (uint32_t i = 0; i < 10; i++)
```

一个例外：如果你需要在循环结束后仍保持计数

器的值，那当然不要在循环域内声明计数器。

现代编译器支持 #pragma once

所以不要：

```
#ifndef PROJECT_HEADERNAME
#define PROJECT_HEADERNAME
.
.
.
#endif /* PROJECT_HEADERNAME */
```

而应该：

```
#pragma once
```

#pragma once告诉编译器只包含头文件一次，且不再需要那三行头部约束了。所有编译器和平台都支持并且推荐这样的编译方式，而不建议使用手动编写头部约束。更多细节，参见pragma once中编译器支持的列表（https://en.wikipedia.org/wiki/Pragma_once）。

C语言允许对自动分配的数组进行静态初始化

所以，不要：

```
uint32_t numbers[64];
memset(numbers, 0, sizeof(numbers));
```

而应该：

```
uint32_t numbers[64] = {0};
```

C语言允许对自动分配的结构体进行静态初始化

不要：

```
struct thing {
     uint64_t index;
     uint32_t counter;
};
struct thing localThing;
void initThing(void) {
    memset(&localThing, 0,
sizeof(localThing));
}
```

而应该：

```
struct thing {
     uint64_t index;
     uint32_t counter;
};
struct thing localThing = {0};
```

重要提示：如果结构体存在填充，{0}方法不会将额外的字节归零。例如结构体struct thing在counter（64位平台上）后需要填充4字节，并且结构体以字长度进行增量填充。如果需要将包含没有使用的填充在内的整个结构体置0，即使可寻址内容只有8+4=12字节，也需要使用memset(&localThing, 0, sizeof(localThing))，并且其中sizeof(localThing) == 16 字节。

如果需要重新初始化已存在并且完成了分配的结构体，那么就为后面的任务声明一个全0的结构体：

```
struct thing {
    uint64_t index;
    uint32_t counter;
};
static const struct thing localThingNull
= {0};
...
struct thing localThing = {.counter =
3};
...
localThing = localThingNull;
```

如果你足够幸运是在C99（或者更新）的开发环境中，就可以使用复合文字而不是全0结构体，参见The New C: Compound Literals（http://www.drdobbs.com/the-new-c-compound-literals/184401404）。复合文字允许编译器自动创建临时匿名结构体，并将其值拷贝到目标中：

```
localThing = (struct thing){0};
```

C99添加可变长数组（C11可选）

因此不要这样：

```
uintmax_t arrayLength =
strtoumax(argv[1], NULL, 10);
void *array[];
array = malloc(sizeof(*array) *
arrayLength);
/* remember to free(array) when you're
done using it */
```

而该：

```
uintmax_t arrayLength =
strtoumax(argv[1], NULL, 10);
void *array[arrayLength];
/* no need to free array */
```

重要警告：通常在栈区分配空间时，变长数组和普通数组一样。如果不想静态地创建一个含300万

个元素的普通数组，那就不要试图在运行时使用这种语法来创建如此巨大的数组。这可不是灵活的Python、Ruby中自动增长的列表。如果运行时指定了一个数组的长度，而这个长度对于栈区而言过大的话，程序将会出现可怕的事情（崩溃、安全问题）。对于简单的，用途单一的情况，变长数组非常方便，但它不能支撑大规模软件开发。如果你需要3个元素的数组，而有时又需要300万个，那么千万不要使用变长数组。

了解变长数组的语法的确不错，因为没准你就会遇到（或者想来一次快速测试）。但有可能因为忘记边界检查，或忘记了目标平台上没有足够的栈空间而使程序彻底崩溃，这是非常危险的反模式语法。

注意：必须确定数长度是合理的值。（比如小于几KB，有时平台上的栈最大空间为4KB）。你不能在栈中分配巨大的数组（数百万条项目），但如果数组长度是有限的，使用C99的变长数组功能比通过malloc手动请求堆内存要简单得多。

再次注意：如果没有对用户的输入进行检查，那么用户就能够通过分配一个巨大的变长数组轻松地杀死程序。一些人指出变长数组与模式相抵触，但是如果严格检查边界，在某些情况下，或许会表现得更好。

C99允许注释非重叠的指针参数

参看限制关键字 (https://en.wikipedia.org/wiki/Restrict，通常是 _restrict)

参数类型

如果函数能接受任意输入数据和任意长度的程序，那就不要限制参数的类型。

所以，不要这样：

```
void processAddBytesOverflow(uint8_t
*bytes, uint32_t len) {
    for (uint32_t i = 0; i < len; i++) {
        bytes[0] += bytes[i];
    }
}
```

而应该：

```
void processAddBytesOverflow(void *input,
uint32_t len) {
    uint8_t *bytes = input;
    for (uint32_t i = 0; i < len; i++) {
        bytes[0] += bytes[i];
    }
}
```

函数的输入类型只是描述了代码的接口，而不是带有参数的代码在做什么。上述代码接口意思是“接受一个字节数组和一个长度”，所以不并想限定函数调用者只能使用uint_8字节流。也许用户甚至想将一个老式的char*值或者其他什么意想不到的东西传递给你的函数。

通过声明输入类型为void*然后重新指定或转换成为实际想在函数中使用的类型，这样就将用户从费力思考你自己的函数库文件的内部抽象中拯救出来。

一些读者指出这个例子中的对齐问题，但我们访问的是输入的单字节元素，所以一切都没问题。如果不是这样的话，我们就是在将输入转换为更大的类型，这时候就需要注意对齐问题了。对于处理跨平台对齐的问题的不同写操作，参看Unaligned Memory Access部分章节（https://www.kernel.org/doc/Documentation/unaligned-memory-access.txt，记住：这页概述细节不是关于跨体系结构的C，所以扩展的知识和经验完全适用于任何示例。）

返回参数类型

C99给我们了将true定义为1，false定义为0的<stdbool.h>的权利。对于成功或失败的返回值，函数应该返回true或false，而不是手动指定1或0的int32_t返回类型（或者更糟糕1和-1；又或者0代表成功，1代表失败？或0带代表成功，-1代表失败）。

如果函数将输入参数类型的范围变得更大的操作无效，那么在任何参数可能无效的地方，所有API都应强制使用双指针作为参数，而不应返回改变的指针。对于大规模应用，那样“对于某些调用，返回值使得输入无效”的程序太容易出错。

所以，不要这样：

```
void *growthOptional(void *grow, size_t
currentLen, size_t newLen) {
    if (newLen > currentLen) {
        void *newGrow = realloc(grow,
newLen);
        if (newGrow) {
            /* resize success */
            grow = newGrow;
        } else {
            /* resize failed, free
existing and signal failure through NULL
*/
            free(grow);
            grow = NULL;
        }
    }
    return grow;
}
```

而应该这样：

```
/* Return value:
 *  - 'true' if newLen > currentLen and
attempted to grow
 *    - 'true' does not signify success
here, the success is still in '*_grow'
 *  - 'false' if newLen <= currentLen */
bool growthOptional(void **_grow, size_t
currentLen, size_t newLen) {
    void *grow = *_grow;
    if (newLen > currentLen) {
        void *newGrow = realloc(grow,
newLen);
        if (newGrow) {
            /* resize success */
            *_grow = newGrow;
            return true;
        }
        /* resize failure */
        free(grow);
        *_grow = NULL;
        /* for this function,
         * 'true' doesn't mean success,
it means 'attempted grow' */
        return true;
    }
    return false;
}
```

如果这样的话就更好了：

```
typedef enum growthResult {
    GROWTH_RESULT_SUCCESS = 1,
    GROWTH_RESULT_FAILURE_GROW_NOT_
NECESSARY,
    GROWTH_RESULT_FAILURE_ALLOCATION_
FAILED
} growthResult;

growthResult growthOptional(void **_
grow, size_t currentLen, size_t newLen)
{
    void *grow = *_grow;
    if (newLen > currentLen) {
        void *newGrow = realloc(grow,
newLen);
        if (newGrow) {
            /* resize success */
            *_grow = newGrow;
            return GROWTH_RESULT_
SUCCESS;
        }

        /* resize failure, don't remove
data because we can signal error */
        return GROWTH_RESULT_FAILURE_
ALLOCATION_FAILED;
    }

    return GROWTH_RESULT_FAILURE_GROW_
NOT_NECESSARY;
}
```

格式化

编码风格非常重要，但同时也没任何价值。如果项目有50页编码风格指南，没人会帮你。但如果你的代码没有可读性，也没人会帮你。解决方案就是永远使用自动的编码格式化。

2016年，唯一可用的C格式化程序是clang-format。它拥有最好的默认C语言格式并且仍在持续改进。

这是我运行clang-format的脚本：

```
#!/usr/bin/env bash

clang-format-style="{BasedOnStyle:llvm,I
ndentWidth:4,AllowShortFunctionsOnASingl
eLine: None, KeepEmptyLinesAtTheStartOfB
locks: false}" "$@"
```

然后调用（假设你称之为cleanup-format）：

```
matt@foo:•/repos/badcode% cleanup-format
-i *.{c,h,cc,cpp,hpp,cxx}
```

选项-i通过覆盖已有的文件进行格式化，而不是写入新文件或者创建备份文件。如果有很多文件，可以采取并行方式递归处理整个源文件树。

```
#!/usr/bin/env bash
```

```
# note: clang-tidy only accepts one file
at a time, but we can run it
#       parallel against disjoint
collections at once.
find . \( -name \*.c -or -name \*.cpp -or
-name \*.cc \) |xargs -n1 -P4 cleanup-
tidy


# clang-format accepts multiple files
during one run, but let's limit it to 12
# here so we (hopefully) avoid excessive
memory usage.
find . \( -name \*.c -or -name \*.cpp -or
-name \*.cc -or -name \*.h \) |xargs
-n12 -P4 cleanup-format -i
```

如今新cleanup-tidy脚本出现了，其内容是：

```
#!/usr/bin/env bash
clang-tidy \
    -fix \
    -fix-errors \
    -header-filter=.* \
    --checks=readability-braces-around-
statements,misc-macro-parentheses \
    $1 \
-- -I.
```

Clang-tidy（http://clang.llvm.org/extra/clang-tidy/）是策略驱动型的代码重构工具。上述选项使用了两个修正：

- readability-braces-around-statements——要求所有if/while/for的申明语句全部包含在大括号内。C语言允许在循环和条件语句后面的单条语句的“大括号可选”存在历史原因。但在编写现代代码时不使用大括号来包含循环和条件将是不可原谅的。也许你会说“但是编译器承认它”并且这样不会影响代码的可读性、可维护性、可理解性和适用性。但你不是为了取悦编译器而编程，编程是为了取悦未来数年后不能了解你当时思维的人。
- misc-macro-parentheses——自动添加宏中使用的所有参数的父类。

clang-tidy运行时非常棒，但却会因为某些复杂的代码卡住。clang-tidy不进行格式化，所以需要在排列整齐大括号和宏之后再运行clang-format。

其他想法

永远不使用malloc，而应该用calloc，当得到为0的内存时将不会有性能损失。如果你不喜欢函数原型calloc(object count, size per object)，可以用#define mycalloc(N) calloc(1, N) 将其封装起来。

能不用memset就不用

当能静态初始化结构体或数组为0时，不要用memset(ptr, 0, len)。不过，如果要对包括填充字节在内的整个结构体进行置0时，memset()是你唯一的选择。

写在最后

对于编写大规模代码来说，想要不出现任何错误是不可能的。就算是我们不必担心那些像RAM中比特位随机翻转的问题或者设备以未知的几率出现错误的可能性，也会有多种多样的操作系统、运行状态、库文件以及硬件平台的问题需要担心。

所以我们能做的最好的事情就是——编写简单又易于理解的代码。

原作者在文章结尾列出了对本文提出意见和建议的读者，以及深入了解这些技术细节的在线文档，受篇幅所限，译文不再赘述。感谢Matt Stancliff授权《程序员》翻译，原文链接https://matt.sh/howto-c。