

## 第 3 章 0x300—解读 C 语言标准（上）

注意：这并不最后完整的章节，仅仅是用来测试。请自由发布，但不要用于任何商业用途！如果你在里面发现什么错误，请及时通知我。我的联系方式是：

`xiyouwang@gmail.com`。

我曾一度踌躇于是否该加入本章内容。一位朋友好心地提醒我：既然这本书是要讲述一门艺术，把标准这种死板的东西放进来会不会显得不够“艺术性”呢？我思考了很久，在编写本章的过程中，我发现标准并不会影响其艺术性，对标准理解得越深刻，我们越能创造出优雅的代码，艺术性反而更多。而且，标准本身也是另一种形式的艺术。

### 0x310. 关于 C 语言标准

20 世纪 80 年代，C 语言风靡一时，它被业界广泛接受和应用，人们发现了 C 优于 BASIC 的诸多长处，使得各种各样的 C 编译器兴起。Microsoft 为 IBM PC 制作了一个 C 编译器，里面引入了 `near` 和 `far` 这两个新的关键字，来处理 Intel 80x86 不规则的构架。随着各个厂家的编译器的兴起，C 语言面临着严峻的威胁，就像当年 BASIC 那样，那就是缺乏一个统一的标准。于是，一个统一的国际 C 语言标准成为迫切需要。

C 语言并不是第一个拥有国际标准的语言，在此之前有不少先例，很多成功的编程语言最终都作了标准化。制定标准的问题是：到底该怎么描述这门语言？目的非常清楚，就是得让人们清楚标准到底在讲些什么。可如果用日常生活的语言讲述，那么越追求精确反而倒会使得标准越冗长，如果用数学式的语言来描述，那可能没多少人能看得懂。标准委员会做了个折衷，采用了一种接近日常用语但又精确的描述方式，这就是我们将要看到的这种语言风格。

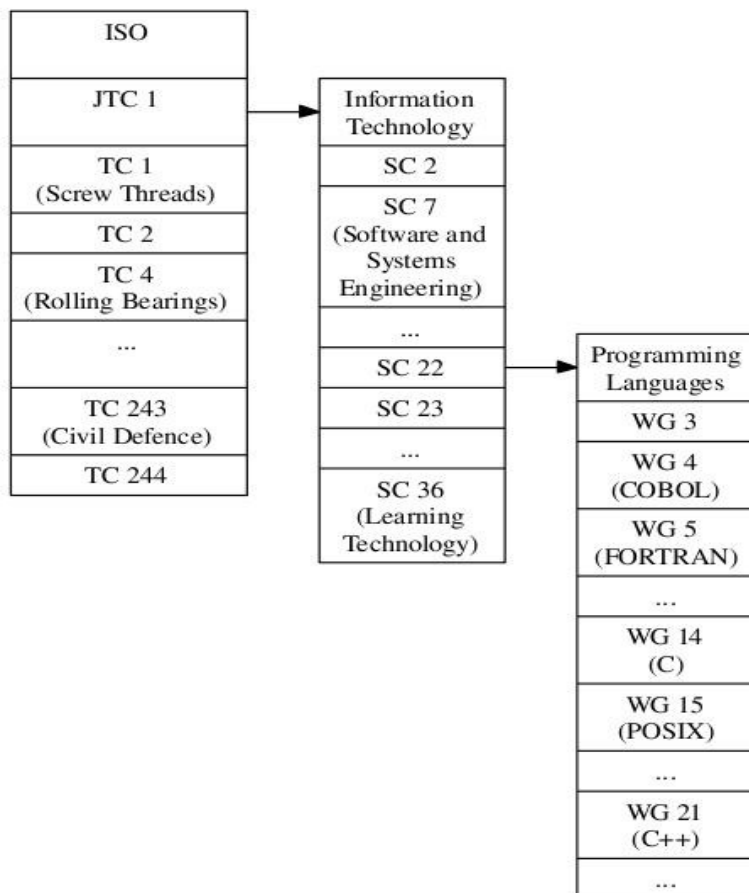
### 0x311. 什么标准？

C 语言标准[1]是由 ANSI（美国国家标准委员会）制定，目前主要的 C 语言规范有 C89(C90)，C95(C94)和 C99。C89 是最早的 C 语言规范，于 1989 年提出，1990 年先由美国国家标准委员会推出 ANSI 版本，后来被接纳为 ISO 国际标准 (ISO/IEC 9899:1990)，因而有时也称为 C90。而后在 94 和 96 年分别对 C90 进行了两次错误修正，在 95 年提出过对 90 版规范的修订案，称为 C95 或者 AMD1。在这次修订中，加入了更多的用来处理多字节和宽字节字符 (`wide and multibyte characters`) 的库函数。gcc 支持的是修正后的 C89(90)版本

的C语言规范，当然也支持C95规范。

## C 标准委员会

C语言标准是由INCITS J11和SC22 WG14共同制订的。J11代表美国的跨部门的C社区，它由大约二三十名成员组成，这些成员代表了硬件生产商，编译器提供商，软件开发商，软件设计师，学术界科学家，作家等。而WG14隶属于ISO，它和ISO的关系可以在下图中反映出来：



最新的一次C语言标准修订是在1999年完成（ISO/IEC 9899:1999），即常称的C99规范。在2001年（ISO/IEC 9899:1999/Cor.1:2001 (E)）和2004年（ISO/IEC 9899:1999/Cor.2:2004 (E)）对C99的错误进行了修正。

C语言标准制定过程中很大的一处败笔就是没有把非常有用的Rationale纳入，而是把Rationale作为单独的一部分和标准一起发布。其实Rationale里包含了很多对标准有用的解释，我们在编写本章时参考了很多Rationale中的内容，建议读者也应该把它作为

标准的一部分去读。

### 从哪里得到 C99?

pdf 格式的标准可以从 [open-std.org](http://open-std.org) 上下载:

<http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf>

这个文档可供个人使用，如果你需要真正的标准，那就得向 ANSI 支付 20\$ 购买，或者买一本《The C Standard》。

另一个在线的版本可以在下面看到:

<http://www.vmunix.com/~gabor/c/draft.html>

gcc 是较早比较全面支持 C99 的 C 语言编译器之一，它的性能丝毫不亚于任何商业编译器。它具有惊人的可移植性，而其最优化功能也非常强大。这些也是我们选择 gcc 的重要理由。gcc 支持的修正后的 C99 规范，但是到目前为止，gcc 还没有完成对 C99 规范的完全支持。附录第二节是 GNU 官方列出的最新的 gcc4.1 对 C99 的支持情况。

### 与标准 C 相关的 gcc 选项 (一)

`-std=iso9899:1990`, `-ansi` 或 `-std=c89` (三者完全等同)

指定完全按照 C89 规范，而禁止 gcc 对 C 语言的扩展。

`-std=iso9899:199409`

使用 C95 规范

`-std=c99` 或 `-std=iso9899:1999`

使用 C99 规范

`-std=gnu89`

使用 C89 规范加上 gcc 自己的扩展 (目前默认)

`-std=gnu99`

使用 C99 规范加上 gcc 自己的扩展

## 0x 312. 标准概览

标准也不是随便就可以制订的，有一些原则也是要遵循的。从 C89 往后，C 标准委员会在制订标准时就一直遵循下面的几条原则[2]:

**a. 兼容性比实现更重要 (Existing code is important, existing implementations are not.)**

C语言自诞生以来积累了很多的代码，一些代码有很大的商业价值，新的标准必须考虑对这些代码的兼容。而现存的编译器却不那么重要，因为没有哪一个编译器可以拥有标准，编译器的实现要考虑标准的要求。

**b. C代码是可移植的，也是不可移植的 (C code can be portable. C code can be non-portable.)**

C虽然诞生在运行UNIX的PDP-11上，但它已经在其它各种各样的构架和系统上实现，标准委员会让语言和库变得更加可移植。虽然这给程序员更多的机会编写完全可移植的代码，但标准并不会强制程序员必须这么做，因为C语言本身就是“高级汇编”，编写与机器密切相关的代码也是C的优点。

**c. 避免“安静的改变” (Avoid “quiet changes.”)**

任何改变了现存代码的意义的修改都会引发问题，标准委员会会避免那些安静的改变，更不会在不给出任何通知的情况下让以前可以运行的程序运行得有所不同。

**d. 标准是“编译器作者和程序员之间的协定” (A standard is a treaty between implementor and programmer.)**

一些数字上的限制被加入了标准中，这能够告诉编译器作者和程序员什么是实现必须提供的，哪些东西是可以期待和依赖的。

**e. 发扬C的精神 (Keep the spirit of C.)**

C89委员会的主要目标就是保护传统的C语言精神，C语言精神有多个方面，但本质的内容就是一些潜在的规则，C语言就是建立在这些规则之上。C语言的精神可以总结为以下几个方面：

- (1) 信任程序员
- (2) 不要阻止程序员去做需要做的工作
- (3) 保持语言小巧而简单
- (4) 给每一种操作只提供一种方式
- (5) 快一些，即使不能保证它是可移植的

**f. 支持国际化编程 (Support international programming.)**

在初期的标准化过程中，国际化在某种程度上只是一种向后的考虑。而现在，国际化已经是一个重要的话题，国际化支持成为C标准委员会不得不考虑的一个重要问题。

**g. 规范化已有的行为来弥补明显的缺陷。(Codify existing practice to address evident deficiencies.)**

这是 C 标准编写的一个重要原则，除非某些提议的新特性可以弥补在很多 C 程序员看来很明显的不足，否则不会引入新的特性。

#### **h. 最小化和 C90 的不兼容性 ( Minimize incompatibilities with C90 (ISO/IEC 9899:1990).)**

标准委员会必须保持向前的兼容性，这样既能够让现有的实现能够转移到新的标准上，又能让遵循以前标准的程序能够无所改变的运行。

#### **i. 最小化与 C++ 的不兼容性 ( Minimize incompatibilities with C++.)**

C++ 作为 C 的超集，它的发展无疑也影响着 C 的发展。委员会乐意让 C++ 变成一个大的语言，C++ 的一些特性值得借鉴，但委员会不会让 C 成为 C++。

#### **j. 保持概念上的简洁 ( Maintain conceptual simplicity.)**

委员会了解用简洁明了的语言描述技术术语的重要性，和其它标准一样，C 语言标准在叙述用词上力求准确而简洁。

### 与标准 C 相关的 gcc 选项 (二)

#### **-Wall**

打开一些很有用的警告，比如：

#### **-Wmain**

对 main 函数的原型进行检查

#### **-Wimplicit**

对使用默认返回 int 这一特性进行检查

等等。

根据我们的经验，加上此选项时 gcc 给的警告信息往往非常有用，我们建议你在使用 gcc 是总是把这个选项加上。

#### **-W**

实际上是 -Wextra，其作用是除了 -Wall 给出的警告，再打开一些额外的警告，这里面给出的警告有时也会很有用，编译时可以考虑加上它。

#### **-pedantic**

打开所有关于标准 C 或标准 C++ 的警告，这让 gcc 看起来非常“卖弄学问”。它会禁止任何标准 C 以外的扩展。

## 0x313. 从 C90 到 C99

### 1. 内联函数

C99 从 C++ 中借鉴了 `inline` 关键字，这也就意味着如果可能，函数会以内联的方式“嵌入”到调用函数中，这保证它能“像宏一样快”，因为它消除了普通函数调用时必须的压栈出栈操作。不过切记，使用内联函数不宜过大。

内联函数的正确使用没有你想象中那么容易，现在我们仔细讨论一下。函数内联无非就是让编译器做一些额外的工作，让该函数能够更快地执行，而编译器采用的手段一般就是把函数的代码直接“替换”到调用函数的代码中，这样可以省去普通函数调用所需的压栈和出栈操作，而且编译器还有可能在两个函数的代码中间做一些必要的优化。

但是，有些时候编译器并不总是按我们的要求把函数内联掉，它可能还要为这个函数生成一段独立的代码，比如，我们要取这个函数的地址，或者函数处于其它不能被内联的上下文中，或者优化选项根本就没打开。如果我们不正确使用内联的话，可能会产生一些重复的代码，如果函数既被内联，而有独立代码生成，更严重时还会导致一些错误，比如函数被内联了却又要计算它的地址。

### 2. 使用“//”进行注释

C 现在也支持 C++ 风格的注释了。虽然一些编译器早已经这么做了，但是只有被标准认定了才是官方的。与传统的“`/**/`”注释相比，这种注释的明显优势是可以嵌套，而且对于少于一行短注释使用这种风格更方便。附录中有我用 Python 编写的一个脚本，它能帮助你把“`//`”风格的注释转换成“`/**/`”风格。这最初是为 Linux 内核准备的，因为内核源代码中积累大量的“`//`”风格的注释，而这种注释是内核编码风格所不推荐的。

### 3. 声明和代码混合

早期的 C 只允许在块的开头，在任何语句开始之前，进行声明，现在标准放松了这一限制，你可以在任何地方声明你需要的变量了。这一点也是受 C++ 的影响。

### 4. for 循环名字约束

for 循环中的变量被约束在 for 循环之中，也就是说，下面的程序是安全的：

```

#include <stdio.h>

int main(void)
{
    int n =1;      /*This 'n' is NOT that 'n' below.*/
    for (int n=0; n<10;n++);
    if (1 == n)
        printf("all right.\n");
    return 0;
}

```

## 5. 可变长数组

C99 的一大亮点就是支持可变长数组，这里的可变长数组并不是像 Perl/Python 中的 `list` 类型那样可以真正的自由伸缩，只是在定义数组时可以使用变量了，而不像过去那样，要求数组定义时的长度必须是一个编译前就确定好的常数。这确实是向前迈进了一步，不过，C99 中的可变长数组还有着不少的限制，具体内容我们在稍后的标准讲解中分析。先看一下下面这个使用变长数组的例子：

```

#include <stdio.h>

int main(void)
{
    int n =10;
    int a[++n];
    printf("The size of the variable-length array a is: %u.\n",
        sizeof(a));
    return 0;
}

```

为了支持变长数组，C99 引入了符号：[\*]，它用在函数的参数列表中，说明这个参数是一个可变长的数组（到目前为止，gcc 还未完全支持[\*]这种数组声明）。而且，C99 为变长数组扩展了 `sizeof` 的功能，当 `sizeof` 作用于变长数组时，它会在运行期间决定这个数组的大小。这真是一个大胆的改进，因为它可以算是一个“安静的改变”。

## 6. \_\_func\_\_ 标识符

或许你对 `__FILE__` 和 `__LINE__` 这两个宏不陌生，现在新增的这个 `__func__` 和它们类似，通过它可以获取当前函数的名字，需要注意的是，在函数作用域之外，这个宏是未定义的。下面的代码可以说明这个问题：

```

#include <stdio.h>

#ifndef __func__
#define I_AM_RIGHT 1
#else
#define I_AM_RIGHT 0
#endif

int main(void)
{
    if (I_AM_RIGHT)
        puts(__func__);
    return 0;
}

```

注意，不要自己定义\_\_func\_\_（事实上，你应该避免自己定义任何以双下划线包围的宏，因为这类宏往往是编译器预定义好的），否则它就会失去原有的意义了。下面是使用\_\_func\_\_的一个好的例子：

```

# if __STDC_VERSION__ < 199901L
# if __GNUC__ >= 2
#   define __func__ __FUNCTION__
# else
#   define __func__ "<unknown>"
# endif
# endif

#ifdef DEBUG
#define debug_printf(fmt, ...) \
    printf("DEBUG: %s:%d@%s: " fmt,\
        __FILE__, __LINE__, __func__, ##__VA_ARGS__)
#else /* !DEBUG */
#define debug_printf(fmt, ...) do{} while(0)
#endif

```

## 7. 特定初始化

标准 C89 需要初始化语句的元素以固定的顺序出现，和被初始化的数组或结构体中的元素顺序一样。

在 ISO C99 中，你可以按任何顺序给出这些元素，指明它们对应的数组的下标或结构体的成员名。为了指定一个数组下标，在元素值的前面写上 “[index] =”，比如：

```
int a[6] = { [4] = 29, [2] = 15 };
```

相当于:

```
int a[6] = { 0, 0, 15, 0, 29, 0 };
```

下标值必须是常量表达式，即使被初始化的数组是自动的。

在结构体的初始化语句中，在元素值的前面用 “.fieldname = ” 指定要初始化的成员名。例如，给定下面的结构体:

```
struct point { int x, y; };
```

和下面的初始化,

```
struct point p = { .y = yvalue, .x = xvalue };
```

等价于:

```
struct point p = { xvalue, yvalue };
```

“[index]” 或 “.fieldname” 就是指示符。在初始化共同体时，你也可以使用一个指示符（或不再使用的冒号语法），来指定共同体的哪个元素应该使用。比如:

```
union foo { int i; double d; };
```

```
union foo f = { .d = 4 };
```

将会使用第二个元素把 4 转换成一个 double 类型来在共同体存放。相反，把 4 转换成 union foo 类型将会把它作为整数 i 存入共同体，既然它是一个整数。

你可以把这种命名元素的技术和连续元素的普通 C 初始化结合起来。每个没有指示符的初始化元素应用于数组或结构体中的下一个连续的元素。比如，

```
int a[6] = { [1] = v1, v2, [4] = v4 };
```

等价于

```
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

当下标是字符或者属于 enum 类型时，标识数组初始化语句的元素特别有用。例如:

```
int whitespace[256]
= { [' '] = 1, ['\t'] = 1, ['\h'] = 1,
  ['\f'] = 1, ['\n'] = 1, ['\r'] = 1 };
```

你也可以在 “=” 前面写上一系列的 “.fieldname” 和 “[index]” 指示符来指定一个要初始化的嵌套的子对象；这个列表是相对于和最近的花括号对一致的子对象。比如，用上面的 struct point 声明:

```
struct point parray[10] = { [2].y = yv2, [2].x = xv2, [0].x = xv0 };
```

如果同一个成员被初始化多次，它将从最后一次初始化中取值。

## 8. 可变参数宏

可变参数的函数在 C 中并不少见，其原理我们也已经在上一章中分析过了。我们最常用的函数 `printf` 就是带可变参数：

```
int printf(const char * fmt,...);
```

而 C99 现在支持可变参数的宏了，这种宏看起来这样：

```
#define display(...) printf( __VA_ARGS__)
```

预留的 `__VA_ARGS__` 就是专门用来处理多个参数的。这样，你就可以这样用上面的宏了：

```
display( "x=%d." ,x);  
display( "Hello, world" );
```

## 与标准 C 相关的 gcc 选项 (三)

### *-Wformat-nonliteral*

如果 `-Wformat` 被指定, 则当格式化字符串不是常量时给出警告。这可以防止格式化字符串攻击 (我们在后面的章节会详细讲这个问题)。

### *-Wcast-align*

对指针转化可能带来的对齐的问题给出警告, 比如, 把 `char*` 转换为 `int*` 时, 该指针指向的 `int` 类型未必就是对齐的。

### *-Wpointer-arith*

对依赖于函数指针或 `void` 类型的指针大小的代码予以警告。在 GNU C 的扩展中, 这两种类型的指针大小都为 1。

### *-Wbad-function-cast*

(仅对 C 适用) 当把函数转化为不匹配的类型时给出警告。

### *-Wmissing-prototypes*

(仅对 C 适用) 当一个全局函数在定义之前没有原型声明时给出警告。

### *-Wstrict-prototypes*

(仅对 C 适用) 对函数原型进行严格检查, 如果函数声明或定义没有指定参数类型就给出警告。

### *-Wmissing-declarations*

(仅对 C 适用) 当一个全局函数在定义之前没有被声明时给出警告。

### *-Winline*

当一个不能被内联的函数声明为内联时给出警告。

## 0x320. 解读 C99

从现在开始, 让我们一起打开标准, 仔细解读其中的奥秘吧。别担心, 这是一次愉快的经历, 读完这一部分会让你成为一名合格的 C 语言律师。

## 语言律师 (Language Lawyer)

*The Free Online Dictionary of Computing* 上对“语言律师”的解释是：语言律师是有经验的软件工程师，熟悉计算机语言的限制和特性，他们能够从长达 200 页的语言说明中找出 5 句话来回答你关于该计算机语言的问题。《人月神话》第三章中对此的描述是：“语言律师能够找到一种简洁而有效的方法来使用这种语言去解决困难，模糊和棘手的问题。”

整个标准大体分为八个部分，各部分内容如下：

| 部分 | 标题                              | 内容                      |
|----|---------------------------------|-------------------------|
| 1  | Scope                           | 描述哪些内容是标准要规定的，哪些是标准之外的。 |
| 2  | Normative references            | 一些参考。                   |
| 3  | Terms, definitions, and symbols | 标准中使用的术语，定义，和符号。        |
| 4  | Conformance                     | 关于一致性的描述。               |
| 5  | Environment                     | 定义和描述 C 程序的环境。          |
| 6  | Language                        | 最重要的一部分，定义 C 语言的语法。     |
| 7  | Library                         | 描述标准头文件，库函数的一些行为。       |
| 8  | Annex                           | 介绍一些警告信息和可移植性话题。        |

标准中的语言风格生硬但又严谨，每一小节的主题都有统一的格式，大致如下：

| 标准一般格式       | 举例  |
|--------------|---|
| 段落号 标题       | <b>6.6 Constant expressions</b>   |
| 语法：<br>语法描述图 | <b>Syntax</b><br>constant-expression:<br>conditional-expression   |
| 描述：<br>一般性描述 | <b>Description</b><br>A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be. |
| 约束：          | <b>Constraints</b>  |

| 标准一般 格式         | 举例   |
|-----------------|--|
| 应该怎样, 不应该怎样     | Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.<br>...  |
| 语义:<br>解释该特性的意义 | <b>Semantics</b><br>An expression that evaluates to a constant is required in several contexts. If a floating expression is evaluated in the translation environment, the arithmetic precision and range shall be at least as great as if the expression were being evaluated in the execution environment.<br>... |

在下面的几个小节中, 我们将一起来分析一下最重要的第 1, 3, 5, 6 部分, 其余部分的分析留给读者作为练习。

## 0x311. 约束 (Scope)

C 语言标准是为了促进 C 的可移植性, 为了供 C 语言编译器编写者和 C 程序员使用。它指定了:

1. C 程序的表示;
2. C 语言的语法和约束;
3. 解释 C 程序的句法规则;
4. C 程序要处理的输入数据的表示;
5. C 程序产生的输出数据的表示;
6. 为统一实现强加的约束和限制。

除此之外, 其它东西都不在标准的规定范围之中的。

## 0x312. 术语，定义和符号 ( Terms, definitions, and symbols)

为制订此标准，如下定义适用。（根据 ISO 31-11，一些数学符号并没有定义。）

“ (对象) *object*: region of data storage in the execution environment, the contents of which can represent values.

*NOTE* When referenced, an object may be interpreted as having a particular type; see 6.3.2.1.”

C 中的“对象”在其它语言中一般被称为“变量”，随着面向对象技术的引入，为了避免歧义，这个术语逐渐被 C 程序员摒弃。C99 Rationale[2]中又进一步指出：

“All objects in C must be representable as a contiguous sequence of bytes, each of which is at least 8 bits wide.”

想想 C 语言中有哪个类型不是如此呢？指针是，结构体是，共同体是，更复杂的类型也皆是如此。这个定义是标准最基础的定义之一，和“位”，“字节”，“访问”等术语的定义有着密切的关系。“字节”这个名词的定义如下：

“ (字节) *byte*: addressable unit of data storage large enough to hold any member of the basic character set of the execution environment.

*NOTE 1* It is possible to express the address of each individual byte of an object uniquely.

*NOTE 2* A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit.”

通常都是一个字节 8 位，但也不排除其它“异类”。但一个 *char* 类型，无论是有符号的还是无符号的，必须刚好占据一个字节。头文件 `limits.h` 中的宏 `CHAR_BIT` 定义了具体平台上一个字节中的位数。位的定义如下：

“ (位) *bit*: unit of data storage in the execution environment large enough to hold an object that may have one of two values.

*NOTE* It need not be possible to express the address of each individual bit of an object.”

这个定义受目前硬件条件的约束，目前的电子计算机都是使用二进制来表示数据，用高电平表示1，用低电平表示0，自从世界上第一台计算机诞生以来就是如此。未来出现量子计算机后这个定义恐怕会受到影响。再向上的一个定义是“字符”：

“ (字符) *character*: *<C>* bit representation that fits in a byte.”

这个定义非常狡猾，很明显，这个定义基于字节和位的定义，它也决定了 char 类型的数据必须是刚好一个字节的大小。抽象意义上的字符定义如下：

“ (字符) *character*: *<abstract>* member of a set of elements used for the organization, control, or representation of data.”

这个定义和“字符集 (character set)”中的“字符”是一个意思，表示抽象意义上我们所看到的一个个字符。与此密切相关的两个概念是“多字节字符”和“宽字符”，把它们加入C是为了支持更大的字符集。它们的定义如下：

“ (多字节字符) *multibyte character*: sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment.

*NOTE* The extended character set is a superset of the basic character set.

(宽字符) *wide character*: bit representation that fits in an object of type *wchar\_t*, capable of representing any character in the current locale.

我们可以认为宽字符是多字节字符在执行环境中的表示。上面的定义也告诉我们，除非直接说明，我们通常所说的“字符”并不包含“多字节字符”，这一点在别处也类似。下面一个重要的定义是：

“ (访问) *access*: *<execution-time action>* to read or modify the value of an object.

*NOTE 1 Where only one of these two actions is meant, ‘ ‘read’ ’ or “modify” is used.*

*NOTE 2 “Modify” includes the case where the new value being stored is the same as the previous value.*

*NOTE 3 Expressions that are not evaluated do not access objects.”*

这可能看起来很简单，但稍微进行深入的探讨我们很快就会发现“访问”这个词定义得并不完整，有一些细节的地方还有探讨，比如：给一个对象乘以一也算是访问这个对象了吗？是的，注释二已经说明了。好的，那么访问一个位字段时是否也访问了和它共享一个存储单元的其它位字段了呢？标准并没有给出答案，WG14已经着手开始探索这种标准的“阴暗角落”，这种工作需要花一些时间。

## 与标准 C 相关的 gcc 选项 (四)

### *-Wundef*

如果一个不是宏的标识符出现在 `#if` 中 (`defined` 之外), 加上此选项会给出警告。

### *-Wnested-externs*

(仅对 C 适用) 当一个 `extern` 的声明出现在函数中时给出警告。

### *-Wcast-qual*

在对指针进行强制转化时, 如果目标类型仅仅是比原类型少一些类型修饰符, 这时会给出警告。比如, 把一个 `const char *` 类型的指针转化成 `char *`。

### *-Wshadow*

当某个局部变量遮盖住其它局部变量或者全局变量, 或者内建函数被遮盖住时, 让编译器给出警告。这个选项可能会非常有用。

### *-Wconversion*

对一些转换给出警告, 比如, 把 `-1` 直接赋给一个 `int` 类型的变量。

### *-Wwrite-strings*

当把 `const` 类型的字符指针直接赋给非 `const` 类型的指针时给出警告。

### *-ffloat-store*

禁止编译器把浮点数放到寄存器中。在某些构架上, 这可以避免不想要的浮点精度超出。对于大多数程序来说, 浮点精度超出只会是一件好事, 但对于那些过分依赖 IEEE 浮点术精度的程序来说并不是如此。

和“访问”的语义最接近的一个术语是“引用 (reference)”, 这个术语同样也适用于某个对象, 它和“访问”的最大区别是一个没有计算的表达式是不会访问对象的, 但却会引用某些对象。

“对齐”的定义如下:

“ (**对齐**) *alignment*: requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address.”

我们在前一章中讨论了对齐的意义, 在这里只不过是把它标准化了。对齐会引发一些其它

问题，比如：既然指向类型 T 的指针应该和指向 T 类型数组元素的指针行为一致，所以 `sizeof(T)` 应该是 T 对齐的倍数，毕竟标准要求不能在数组元素之间插入填充字节。填充字节在结构体类型中很常见，这也是为了对齐，不过，填充的字节不会参与对象的值的表示。`malloc()` 可能会返回对齐的对象，但不要过分假设这一点。`malloc()` 不知道它分配的空间会用来做什么，所以它才会做最坏的打算。

C++ 标准中对“对齐”的解释明显要相对弱一些 (3.9 p5) :

*“Object types have alignment requirements (3.9.1, 3.9.2). The alignment of a complete object type is an implementation-defined integer value representing a number of bytes; an object is allocated at an address that meets the alignment requirements of its object type.”*

而在其它语言中一般都把这个细节隐藏起来。`gcc` 提供了两种扩展方式来让我们指定对齐：

1. `__alignof__` 操作符。它返回该类型或对象应该满足的对齐字节数，这个值会因机器不同而异。下面的程序可以展示它的作用：

```
#include <stdio.h>
typedef struct {
    char cval;
    int ival;
}mytype;
int main(void)
{
    mytype test;
    printf("__alignof__(char)=%d\n", __alignof__(char));
    printf("__alignof__(short)=%d\n", __alignof__(short));
    printf("__alignof__(int)=%d\n", __alignof__(int));
    printf("__alignof__(long)=%d\n", __alignof__(long));
    printf("__alignof__(long long)=%d\n", __alignof__(long long));
    printf("__alignof__(float)=%d\n", __alignof__(float));
    printf("__alignof__(double)=%d\n", __alignof__(double));
    printf("__alignof__(mytype)=%d\n", __alignof__(mytype));
    printf("__alignof__(test)=%d\n", __alignof__(test));
    return 0;
}
```

2. `__attribute__` 关键字。`__attribute__` 属性有很多，而且功能非常强大，和对齐密切相关的是 `aligned` 属性，它的作用是在定义对象时指定它的对齐要求。下面两种用法都可以：

```
int intvalue __attribute__((aligned(32)));
short shlist[122] __attribute__((align));
```

我们将会在第五章中看到它的一个实际应用。

“值”的定义如下：

“**(值) value**: *precise meaning of the contents of an object when interpreted as having a specific type.*

**(未指定的值) unspecified value**: *valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance.*

*NOTE An unspecified value cannot be a trap representation.*

**(实现定义的值) implementation-defined value**: *unspecified value where each implementation documents how the choice is made.*

**(未确定的值) indeterminate value**: *either an unspecified value or a trap representation.”*

关于实现定义的值，最好的例子莫过于 `limits.h` 中的宏，比如： `INT_MAX`， `UINT_MAX` 和 `LONG_MAX`。未确定的值就是那些没有初始化的值，一般来说，直接使用未初始化的值编译器都会给出警告。

这一部分中有两个意义非常接近的术语，虽然它们翻译成汉语都是“参数”，但“实参”和“形参”这两个常用术语中的“参数”并不是一个“参数”，实际上，“实参”中是“argument (actual argument)”，而“形参”中是“parameter (formal parameter)”，它们的定义如下：

“**(参数) argument**:

*actual argument (实参)*

*actual parameter (deprecated) (不赞成使用)*

*expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation.*

**(参数) parameter**

*formal parameter (形参)*

*formal argument (deprecated) (不赞成使用)*

*object declared as part of a function declaration or definition that acquires a*

*value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition.”*

从上面的定义中，我们可以清晰地看出文字上的差异。需要额外指出的是，函数似的宏和宏调用中也采用形参和实参这两个概念。下面的程序可以帮你理解这两个概念：

```
#define test(a) (a)+2 /*One parameter.*/
int foo(int a, int b); /*Two parameters.*/
int foo(int a, int b) /*a and b are parameters.*/
{
    return a-b;
}
int main(void)
{
    int i;
    i = foo(3, 2); /*3 and 2 are arguments.*/
    i = test(4); /*So does 4.*/
    return 0;
}
```

好了，剩下的一些术语主要就是供标准叙述时使用，它们的作用仅仅是帮助你更好地理解标准，在后面我们将会频繁地看到这些术语的使用。

“ (行为) **behavior**: *external appearance or action.*

(未定义行为) **undefined behavior**: *behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements*

*NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).*

(未指定行为) **unspecified behavior**: *use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance.*

(实现定义的行为) **implementation-defined behavior**: *unspecified behavior where each implementation documents how the choice is made.*

*EXAMPLE An example of implementation-defined behavior is the propagation of the*

*high-order bit when a signed integer is shifted right.*

**(本地指定行为)** : *locale-specific behavior* behavior that depends on local conventions of nationality, culture, and language that each implementation documents.”

上面这几个术语是标准对程序一些行为的分类和定义，它们能很好地划分这些行为的属性。未定义行为的一个例子是整数溢出时的行为，未指定行为的一个例子是函数实参计算的顺序，实现定义的行为的一个例子是有符号整数进行右移时最高位的传播。

**(约束)** *constraint*: restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted.”

约束出现在第6部分的条款中，通常该国际标准会在这些条款中使用“应该 (should)”一词。

**(正确舍入的结果)** *correctly rounded result*: representation in the result format that is nearest in value, subject to the effective rounding mode, to what the result would be given unlimited range and precision.”

**(诊断信息)** *diagnostic message*: message belonging to an implementation-defined subset of the implementation’s message output”

标准并没有强加该输出什么样的消息，编译器可以根据自己的“喜好”来定。

**(向前引用)** *forward reference*: reference to a later subclause of this International Standard that contains additional information relevant to this subclause.”

C++标准中没有向前引用，但在其它条款中可能包含更多的引用。

**(实现)** *implementation*: particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment.”

说白了，这其实就是指“编译器”。标准委员会之所以尽量把这个名词定义得更加“抽象化”，是因为他们不想纠缠于C程序在抽象机器上是如何解释的。

“（实现限制） *implementation limit: restriction imposed upon programs by the implementation.*”

现实中总是有各种各样的限制，标准也难以逃避。定义它是为了告诉程序员哪里存在限制，不要在限制之外进行操作。

“（操作规程建议） *recommended practice: specification that is strongly recommended as being in keeping with the intent of the standard, but that may be impractical for some implementations.*”

## 0x313. 一致性（Conformance）

这一部分是标准正式内容的序幕，主要解释了程序，实现和标准的一致性这方面的问题。其中值得一提的内容是：

1. “*In this International Standard, “shall” is to be interpreted as a requirement on an implementation or on a program; conversely, “shall not” is to be interpreted as a prohibition.*”

[3]中统计数字表明，shall在C99标准中共出现537次（不计shall not），而shall not共出现51次，包括两次在脚注中。

2. “*The implementation shall not successfully translate a preprocessing translation unit containing a #error preprocessing directive unless it is part of a group skipped by conditional inclusion.*”

这足够可以保证#error指示的错误可以让翻译立即停止。

3. *“A strictly conforming program shall use only those features of the language and library specified in this International Standard. It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit.”*

这一部分的第4条注释中这样解释“严格一致的程序”这个概念。值得一提的是，它和一致程序还不一样，标准定义严格一致的程序是为了最大化程序的可移植性，而一致程序可能还依赖于一致实现中的不可移植的特性，它的可移植程度明显要低于严格一致的程序。

4. *“An implementation shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.”*

## 0x314. 环境 (Environment)

好了，我们的序曲终于开始奏响了。从这一节起，我们将一起进入C99标准最关键的两个部分：环境和语言。

C语言程序涉及到的环境分为两种：翻译环境和执行环境。这部分的一开始，标准就抛出对“翻译环境”和“执行环境”两个名词的定义：

*“An implementation translates C source files and executes C programs in two data-processing-system environments, which will be called the translation environment and the execution environment in this International Standard.”*

这定义了C语言编译和执行的环境，一般来说，这两种环境是相同的，但是对于交叉编译（cross-compiled）的代码来说并非如此。**[2]**中这样解释到：

*“Because C has seen widespread use as a cross-compiled cross-compilation language, a clear distinction must be made between translation and execution environments. The C89 preprocessor, for instance, is permitted to evaluate the expression in a #if directive using the long integer or unsigned long integer arithmetic native to the translation environment: these integers must comprise at least 32 bits, but need not match the number of bits in the execution environment. In C99, this arithmetic must be done in `intmax_t` or `uintmax_t`, which must comprise at least 64 bits and must match the execution environment. Other translation time arithmetic, however, such as type casting and floating point arithmetic, must more closely model the execution*

*environment regardless of translation environment.”*

上面用**粗体**标出的部分应该特别注意，C99 中用 `intmax_t` 或 `uintmax_t` 类型对 `#if` 预处理命令中的算术运算做处理，而且还能保证这至少是 64 位，而在此之前是用 `long` 或者 `unsigned long`。

试试下面的几个预处理命令：

```
#define UINT_MAX      (~0U)
#if (0xffffffffffffff == UINT_MAX)
    #error argh
#endif
```

（在 `gcc` 下用这个命令编译：`gcc -E -P pre.c`）你总会得到一个错误。预处理中的算术运算到底是使用有符号的还是无符号的类型，C99 标准规定：不管其前缀或者后缀，总是先尝试 `intmax_t`，如果不能表示该整数，再尝试 `uintmax_t`。

## 安静的改变

标准 C 做的改动并非都是那么引人注意，一些改变可能看起来非常安静，可能会不声不响地就做了大的变化。虽然 C 标准委员会力图避免这样的改变，但我们仍然可以看到一些安静的改变。这里就是一处。

因为引入了新的类型（`intmax_t` 和 `uintmax_t`，在 `<stdint.h>` 中定义），预处理中的算术运算必须使用这两种类型，而 C89 中并没有对此作任何要求。这个改变对交叉编译的实现非常重要。这个改变很安静，不是吗？

整个第 5 部分又分为两节：第一节“概念模型”中主要描述了 C99 标准定义的 C 程序环境，第二节“环境因素”中描述了影响 C 程序翻译环境和执行环境的一些基本因素。

## A. 翻译环境 ( Translation environment )

### 1. 程序结构 ( Program structure )

这一小节中有几个重要的定义，如下：

源文件 ( source file ) : *“The text of the program is kept in units called source files, (or preprocessing files) in this International Standard.”*

预处理翻译单元 (preprocessing translation unit) : “A source file together with all the headers and source files included via the preprocessing directive #include is known as a preprocessing translation unit.”

翻译单元 (translation unit) : “After preprocessing, a preprocessing translation unit is called a translation unit.”

这几个概念能很好地解释为什么我们最好不要#include 一个.c 文件。因为，预处理时，预处理器会原封不动地把.c 文件中的代码拷贝进包含它的源文件中，而翻译时，这两个文件就是同一翻译单元了，里面的 static 和 extern 的意义就可能被破坏了。

## 2. 翻译阶段 ( Translation phases)

这一小节主要阐述了编译的步骤，标准把这个过程分为了 8 个阶段，实际上，主要的阶段还是我们熟悉的三个：编译，链接，执行，这三阶段我们会在第 5 章中详细讨论。

需要额外指出的一点是：相邻字符串常量的连接并不是在预处理阶段完成的，而是在预处理之后。

用 gcc 的 -v 选项我们可以查看编译的大体过程，下面是一个示例：

```
...
/usr/libexec/gcc/i386-redhat-linux/4.1.1/ccl -quiet -v attack.c -quiet -dumpbase attack.c
-mtune=generic -auxbase attack -g -O2 -version -o /tmp/ccgBzac9.s
ignoring nonexistent directory "/usr/lib/gcc/i386-redhat-linux/4.1.1/../../../../i386-redhat-
linux/include"
#include ".." search starts here:
#include <...> search starts here:
  /usr/local/include
  /usr/lib/gcc/i386-redhat-linux/4.1.1/include
  /usr/include
End of search list.
...
GGC heuristics: --param ggc-min-expand=62 --param ggc-min-heapsize=60400
Compiler executable checksum: 0f3190ce853c36e93c35149414a5b09c
  as -V -Qy -o /tmp/ccJfjb4Q.o /tmp/ccgBzac9.s
...
/usr/libexec/gcc/i386-redhat-linux/4.1.1/collect2 --eh-frame-hdr -m elf_i386 -dynamic-linker
/lib/ld-linux.so.2 -o attack /usr/lib/gcc/i386-redhat-linux/4.1.1/../../../../crt1.o /usr/lib/gcc/i386-
redhat-linux/4.1.1/../../../../crti.o /usr/lib/gcc/i386-redhat-linux/4.1.1/crtbegin.o -L
/usr/lib/gcc/i386-redhat-linux/4.1.1 -L/usr/lib/gcc/i386-redhat-linux/4.1.1 -L/usr/lib/gcc/i386-
redhat-linux/4.1.1/../../../../tmp/ccJfjb4Q.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --
as-needed -lgcc_s --no-as-needed /usr/lib/gcc/i386-redhat-linux/4.1.1/crtend.o /usr/lib/gcc/i386-
redhat-linux/4.1.1/../../../../crtn.o
```

更详细的内容我们放到第五章详细介绍。

在这里，我们第一次碰到了三联符（trigraphs）这个概念，我们暂时忽略它，后面我们会详细解释。

### 3. 诊断信息（Diagnostics）

预处理单元或翻译单元中包含违反语法规则的代码时，即使这种行为已经被指明是未定义的或由实现定义的，实现至少应该给出一条诊断信息。其它环境下没必要给出诊断信息。

标准并没说明具体的诊断信息应该是什么，它可以仅仅是简单的“syntax error”，也可以是数十行的说明。一般的编译器都会给出出错的位置，错误描述等相关信息，以便程序员纠正错误。

C++标准中对此的相关描述如下：

*“If a program contains a violation of any diagnosable rule, a conforming implementation shall issue at least one diagnostic message, except that...”*

*“If a program contains a violation of a rule for which no diagnostic is required, this International Standard places no requirement on implementations with respect to that program.”*

## B. 执行环境（Execution environments）

标准定义了两种执行环境：独立式（freestanding）和托管式（hosted）。独立式执行环境一般是嵌入式系统，不在 C99 的范围之中，也不在我们的讨论范围中。标准对它只做了很少的规定，我们直接来看 Hosted environment 部分。（注：此章后面如无特殊说明，均指在 hosted environment 中。）

### 1. 程序启动（Program startup）

在这一小节中标准明确指出 main 函数的标准原型只有以下两个（Rationale 中指出，main 是唯一一个带零个或者两个参数的函数）：

```
int main(void) { /* ... */ }
```

或：

```
int main(int argc, char *argv[]) { /* ... */ }
```

（或者与其等价的：int main(int argc, char \*\*\*argv) { /\* ... \*/ }，或者用把 int 用

typedef 重新定义后的类型) 而其它都是现实来定义的。当然, 有很多人也喜欢这样的 main 原型:

```
main() { /* ... */ }
```

我们知道由于历史的原因, 如果函数定义时不指明返回类型, 默认的是返回 int。但是, 现在, 这已经不再被 C99 支持了, 函数返回值的类型是必须要指明的。其它常见但不可移植的 main 函数原型有:

```
int main (int argc, char *argv[], char *env[])  
int main (int argc, wchar_t *argv[])
```

## 其它语言的对比

从 C 派生出来的语言——C++, Java, C#等都有 main 函数 (方法), 它们的执行也都是从 main 开始的。标准 C++ 中 main 函数的原型:

```
int main() { /* ... */ }
```

或

```
int main(int argc, char* argv[]) { /* ... */ }
```

但它明确指出了其它实现中的 main 的返回值都必须是 int。

Java 中的 main 方法的原型是:

```
public static void main (String args[])
```

C#中的 main 方法原型是:

```
public static void Main (string [] args)
```

而 Python 和 Perl 这种脚本语言中是没有 main 函数的, 它们的程序的执行是自上而下的。

如果使用 main 的第二种标准原型, 标准又做出如下规定:

- argc 应该是非负数。这一点很显然, 因为它表示通过命令行传递进来的参数的个数。
- argv[argc]应该是空指针。因为在 argv 指针数组中, 空指针用来表示结束。
- 如果 argc 大于 0, 则 argv[0]~argv[argc-1]存放的是指向字符串的指针, 它们在系统启动此程序时就已经建立好, 是由实现来确定的值。其中, argv[0]指向的字符串用来表示程序的名字 (如果在此系统中程序名字是不可得到的, argv[0][0]应该是'\0'), argv[1]~argv[argc-1]指向的字符串表示程序的参数。

- `argc`, `argv` 以及 `argv` 数组中指向的字符串的值都应该是可以修改的，这些值在程序启动到程序结束之间的这段时间中应该保持最后修改的结果。注意：`argv` 数组的内容并不一定 是可以修改的。标准之所以这样规定可能是为递归调用 `main` 函数方便，但递归调用 `main` 在 C99 中是不允许的（在 C90 中是可以的）。

## 2. 程序执行 ( Program execution)

- 一个程序可以使用到标准库中所有函数，宏，类型定义，对象。C 标准库没有子集，编译器的库应该包含全部标准库函数的实现。但是，在某些系统上，一些库函数的实现比较困难，这时就需要有最小的功能需求，比如信号处理函数。
- 本国际标准中的语义描述是针对抽象机器的行为，优化是与其不相关的。好的编译器都可以对代码做适当的优化，经过优化的程序和未优化的可能在行为上有很大的差异，优化代码的行为不在此标准规定的范围之内。
- 访问 `volatile` 类型的对象，修改对象，修改文件，或者调用做这些操作的函数都有副作用，它会改变执行环境的状态。计算表达式的值可能会产生副作用。下面的例子可以很好地说明问题：

```
extern int glob_var;
void foo(void)
{
    glob_var+4;      /* No side effect. */
    if (glob_var+4 == 0)
        glob_var++; /* A side effect. */
}
```

- 有副作用的计算中，子表达式的计算顺序是重要的。例如： $(++x)*(x+1)$ ，当  $x=0$  时，如果先算  $++x$ ，上式计算结果为 2，如果先算  $x+1$ ，上式计算结果为 1。再如，对函数 `g(int, int)` 的调用 `g(x, ++x)`，当  $x=1$ ，这个调用是 `g(1, 2)` 还是 `g(2, 2)`？这就需要下面的一个概念。
- 接下来是一个非常重要的概念：顺序点 (Sequence point)。一个顺序点，被定义为程序执行过程中的这样一个点：该点前的表达式的所有副作用，在程序执行到达该点之前发生完毕；该点后的表达式的所有副作用，在程序执行到该点时尚未发生。再看这个例子： $(++i) + (++j)$ ，这个表达式的计算，有两个副作用：`i` 自增 1；`j` 自增 1。但是到底哪一个先发生？答案是：任何答案都不对。为什么？因为标准并不定义副作用发生的顺序。标准只保证：一个表达式的全部副作用，不在达到该表达式紧邻的前一顺序点前发生，并且一定在达到该表达式紧邻的下一个顺序点之前发生完毕。
- 标准在 6.5#2 中还规定：两个相邻顺序点之间，对某一表达式求值，最多只能造成任一特定对象的值被更改一次。如果表达式求值过程会更改某对象的值，那么要求更改前的值被读取的唯一目的，只能是用来确定要存入的新值。任何对相邻顺序点间表达式求值的多个副作用发生的顺序进行假设，或者违反上述“一次更改、读

取仅用于确定新值”规定的代码，其执行结果都是未定义的。

- 标准在 Annex C 中列出的顺序点有：(1) 函数调用时，实参表内全部参数求值结束，函数的第一条指令执行之前（注意参数分隔符“,”不是顺序点）；(2) &&操作符的左操作数结尾处；(3) ||操作符的左操作数结尾处；(4) ?:操作符的第一个操作数的结尾处；(5) 逗号运算符；(6) 完整声明语句的结束；(7) 在库函数马上返回之前；(8) 表达式求值的结束点，具体包括下列几类：自动对象的初值计算结束处；表达式语句末尾的分号处；do/while/if/switch/for 语句的控制条件的右括号处；for 语句控制条件中的两个分号处；return 语句返回值计算结束（末尾的分号）处。
- 通常我们认为，标准对“顺序点”及其语义的定义，一是为了严谨地定义 C 的表达式和求值过程，并不是为了让程序员通过对顺序点的掌握，（过分地）利用表达式求值的副作用；二是为了尽量消除编译器解释表达式时的歧义，如果顺序点还是不能解决某些歧义，那么标准允许编译器的实现自由选择解释方式。实际工作中，我们完全可以通过引入中间变量，避开“顺序点”这样既容易出错又极大地降低代码可读性的“边缘概念”。
- 任何顺序点上，volatile 类型的对象都是稳定的，前一次访问完成时后序访问还未发生。很多编译器在对待 volatile 类型的对象时都非常小心，它们一般不会在上面做优化。

接着，标准给出了几个例子，让我们一起来看看。例 2 中，

```
char c1, c2;  
/* ... */  
c1 = c1 + c2;
```

标准说明，在上面的计算过程中 char 类型会先被转换成 int 类型，然后进行相加，再把相加的结果从 int 类型转回 char 类型。这是出于处理器的考虑，一般来说，安静地丢弃高位值对处理器来说很常见；而且现代的处理器的计算整型甚至可能比计算单字节类型还要快。

同理，例 3 中，

```
float f1, f2;  
double d;  
/* ... */  
f1 = f2 * d;
```

标准也说明，如果实现可以保证用单精度浮点计算的结果和用双精度浮点一样，那就可以用。但实际上，这个保证是很难实现的。

例 5 中关于浮点数的计算，因为我们在第 2 章中已经总结过了，所以这里就不再重复了。

例 6 给出的是整数溢出的一个例子，实际上，整数溢出是很容易犯的一个错误，我们在后面的章节中将会看到，被用了多年的 Binary Search 程序居然也存在溢出的 bug！避免溢出的发生必须慎之又慎。

### 3. 程序结束 ( Program termination)

如果 main 函数的返回类型是和 int 兼容的，从 main 中返回就等价于调用 exit，exit 的参数就是返回值。到达 main 结束的 “}” 时意味着返回 0。如果 main 的返回值和 int 不兼容，返回到系统的结束状态是未指定的。从这来我们可以看出，如果是 void main 的话，标准就不能保证 exit 有正确的参数，退出到系统的状态是未知的。

在 Windows 上，程序的退出状态值可能不那么重要，但在 Linux/Unix 上，退出值直接决定着程序运行的好坏，很多脚本都是根据这个值来判断该程序执行成功与否。一般来说，0 代表成功，非 0 代表失败。请重视这个值！

## C. 环境因素 ( Environmental considerations)

### 1. 字符集

编写源文件使用的字符集被称为源字符集 (source character set)；而在执行环境中解释用的字符集叫做执行字符集 (execution character set)。上面两种字符集每个又可以分为两种：基本字符集 (basic character set)，仅仅包含该国际标准指定的内容；扩展字符集 (extended characters)，包含一些本地化的字符。标准暗示，所有的实现都应该支持基本字符集。

基本字符集总共包含 96 个字符，它们是：空格，水平和垂直制表字符，换页符 (form feed)，和换行符 (new-line)，其余 91 个如下 (我们在前面已经提到过)：

```
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
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
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " ' "
```

标准对它们又进一步规定：

- “这些字符必须能刚好放到一个字节中表示。”
- “在源代码文件中遇到的其它字符 (除了标识符，字符常量，字符串常量，头文件名，注释，和未被转化成标识符的预处理符号中的)，行为是为定义的。”
- “在字符常量或者字符串中，执行字符集的成员应该可以被源字符集中对应的成员，或者可以用包含反斜线\跟着一个或多个字符的转义序列来表示。”
- “一个字节中各位全为 0 的字符是空字符，它应该在基本执行字符集中，用来结束一个字符串。”

## 2. 三联符序列 ( Trigraph sequences)

三联符是什么呢？它们其实就是就一连串的三个字符（前两个是??），用来对某个其它字符进行指定的替换。替换规则如下表：

| 三联符 | 替换 |
|-----|----|
| ??= | #  |
| ??/ | \  |
| ??' | ^  |
| ??( | [  |
| ??) | ]  |
| ??! |    |
| ??< | {  |
| ??> | }  |
| ??- | ~  |

（注意：???并不是三联符序列。）可既然 C 语言的关键词和操作符都是 ASCII 字符，为什么还存在替换呢？引入三联符的原因是字符编码问题。ISO 646 规定说各国可以用自己国家的一些符号替换掉 ASCII 里的“{” “}” “[” “]” 等，这样问题就来了。那些字符明明不是 ASCII，但仍然可以用到程序中，甚至可能没错，因为编译程序只认识字符对应的 ASCII 码值。但这样程序读起来明显就费劲了。为了照顾这种情况，C/C++ 语言标准才考虑用三联符来把它们给替换掉。虽然非 ASCII 的 ISO646 字符现在已经很少使用了，但这个替换规则还是在标准中保留了下来。这还没完，后来，1994 年的时候，C 语言标准的一个修订（现在已经被包含到 C99 中）提出了二联符 (digraphs)，它们比三联符更具有可读性，它们的替换规则如下：

| 二联符 | 替换 |
|-----|----|
| <:  | [  |
| :>  | ]  |
| <%  | {  |
| %>  | }  |
| #:  | #  |
| %%: | ## |

和三联符不同的是，引用字符串，字符常量，和注释中的二联符是不会被替换掉的。

下面就是一个使用三联符的 C 程序：

```
??=include <stdio.h> /* # */

int main(void)
??< /* { */
    char n??(5??); /* [ and ] */

    n??(4??) = '0' - (??-0 ??' 1 ??! 2); /* ~, ^ and | */
    printf("%c??/n", n??(4??)); /* ??/ = \ */
    return 0;
??> /* } */
```

注意，在 gcc 上编译这个程序时需要打开 `-ansi`，或者 `-trigraphs` 选项。C99 Rationale 指出：**这是一个安静的改变**，这可能会导致 C89 的标准程序出现异常结果。

因为三联符的原因，[4] 中建议不要在代码中使用连续的问号，在注释中也不要使用。另一个问题又来了：我要是非得使用连续的问号呢？注释中的没太大关系，直接用空格拆开就是了。如果出现在字符串常量中而又不好直接避免，我们可以这样拆开它们：“...?\?...”或者“...?\"?...”。

### 3. 字符显示

活动位置（active position）是 `fputc` 函数输出的下一个字符出现在显示设备上的位置。而打印字符是由 `isprint` 函数来定义的。

执行环境中的表示非图形字符的字母表的转义序列应该在显示设备上做如下动作：

`\a`（响铃）：发出声音的或可视的警告，而不改变当前的活动位置。

`\b`（退格）：把活动位置移动到当前行的前一个位置。如果活动位置在一行的开始，行为是未指定的。

`\f`（换页符）：把活动位置移动到下一个逻辑页的开始。

`\n`（换行符）：把活动位置移动到下一行的开始。

`\r`（回车符）：把活动位置移动到当前这行的开始。

`\t`（水平制表符）：把活动位置移动到当前行的下一个水平表格位置。如果活动位置在或者超过最后一个水平表格位置，行为是未定义的。

`\v`（垂直制表符）：把活动位置移动到当前行的下一个垂直表格位置。如果活动位置在或者超过最后一个垂直表格位置，行为是未定义的。

这些转义字符应该对应一个唯一的值，使其能放到一个 `char` 对象中。

### 4. 信号与中断

信号处理对 C99 委员会来说是一个棘手的问题，因为它很难独立于平台而描述。信号处理在 Unix/Linux 上是一个复杂的话题，这涉及进程间的异步通信。所以，标准只对此作了如下规定：

“函数应该考虑到它有可能被信号中断，或者被信号处理函数调用，没有提前警告，但之后仍然可以存活。”

这对实现做了明确的要求，必须能够处理信号的影响，保证函数的可靠性。Rationale 中还提到了 `longjmp` 这个库函数，在后面的章节中，我们会对它进行详细介绍。

## 5. 环境限制

这一部分主要就是对编译器中的一些限制做最小化约束，也就是要求编译器至少要能解释具备这些限制的程序。这些最小限制包括：

- 127 层嵌套的语句块；
- 63 层嵌套的条件包含；
- 声明语句和表达式中的 63 层圆括号；
- 一个函数定义中的 127 个参数；
- 函数调用中的 127 个参数；
- 65535 字节大小的对象；

等等。其它就是一些头文件，像 `<limits.h>`、`<float.h>`，中关于数值限制的宏定义（比如，`INT_MAX`，`UINT_MAX`）。看过 C89 的读者一定会发现，C99 在这部分数值上进行了大幅度的增加。我们在这里仅列出一部分常见的宏（摘自 `glibc` 中的 `limits.h`）：

```
/* Number of bits in a `char'. */
# define CHAR_BIT      8

/* Minimum and maximum values a `signed char' can hold. */
# define SCHAR_MIN     (-128)
# define SCHAR_MAX     127

/* Maximum value an `unsigned char' can hold. (Minimum is 0.) */
# define UCHAR_MAX     255

/* Minimum and maximum values a `char' can hold. */
# ifdef __CHAR_UNSIGNED__
#   define CHAR_MIN     0
#   define CHAR_MAX     UCHAR_MAX
# else
#   define CHAR_MIN     SCHAR_MIN
#   define CHAR_MAX     SCHAR_MAX
# endif
```

```

/* Minimum and maximum values a `signed short int' can hold. */
# define SHRT_MIN      (-32768)
# define SHRT_MAX      32767

/* Maximum value an `unsigned short int' can hold. (Minimum is 0.) */
# define USHRT_MAX     65535

/* Minimum and maximum values a `signed int' can hold. */
# define INT_MIN       (-INT_MAX - 1)
# define INT_MAX       2147483647

/* Maximum value an `unsigned int' can hold. (Minimum is 0.) */
# define UINT_MAX     4294967295U

/* Minimum and maximum values a `signed long int' can hold. */
# if __WORDSIZE == 64
#   define LONG_MAX    9223372036854775807L
# else
#   define LONG_MAX    2147483647L
# endif
# define LONG_MIN     (-LONG_MAX - 1L)

/* Maximum value an `unsigned long int' can hold. (Minimum is 0.) */
# if __WORDSIZE == 64
#   define ULONG_MAX   18446744073709551615UL
# else
#   define ULONG_MAX   4294967295UL
# endif

```

更多的限制读者可以直接参考 C99 标准，我们在这里就不详细讨论了，毕竟除了少数的 C 编译器作者外，其他人很少关心这些限制的具体数字。

## 参考资料

- [1] *ISO 1999 Programming Languages—C*. Available at <http://www.vmunix.com/~gabor/c/draft.html>.
- [2] *Rationale for International Standard—Programming Languages—C*.
- [3] Derek M. Jones, *The New C Standard—An Economic and Cultural Commentary*, 2005.

[4] Avoid Using Repeated Question Marks,  
[https://www.securecoding.cert.org/confluence/x/nAE\\_](https://www.securecoding.cert.org/confluence/x/nAE_)