



Notes for Professionals

专业人员笔记C 专业人员笔记

Chapter 10: Arrays

Arrays are derived data types, representing an ordered collection of values ("elements") of another type in C have a fixed number of elements of any one type, and its representation stored contiguously in memory without gaps or padding. C allows multidimensional arrays whose size, and also arrays of pointers.

C supports dynamically allocated array whose size is determined at run time. C99 and later support arrays or VLA.

Section 10.1: Declaring and initializing an array

The general syntax for declaring a one-dimensional array is:

```
type arrName[size];
```

where type could be any built-in type or user-defined types such as structures, arrays and etc. x is an integer constant.

Declaring an array (an array of 10 int variables) in this case is done like this:

```
int array[10];
```

It now holds indeterminate values. To ensure it holds zero values while declaring, you can use initialization. This example declares an array of 10 int's, where values 1, 2, 3, all other values will be zero:

```
int array[10] = {1, 2, 3};
```

In the above method of initialization, the first value in the list will be assigned to the second value will be assigned to the second member of the array and so on. If size, then as in the above example, the remaining members of the array will be zero. List initialization (ISO C99), explicit initialization of the array members is possible.

Arrays can also have initializers, this example declares an array of 10 int's, where values 1, 2, 3, all other values will be zero:

int array[10] = {1, 2, 3};

In most cases, the compiler can deduce the length of the array for you, thus brackets are optional.

int array[] = {1, 2, 3}; // an array of 3 int's //

Declaring an array of zero length is not allowed.

Variable Length Arrays (VLA for shorts) were added in C99, and made optional in C11. They are equal to normal arrays, with one, important difference. The length doesn't have to be known at compile time. VLAs have automatic storage duration. Only pointers to VLAs can have static storage duration.

VLAs are derived data types, representing an ordered collection of values ("elements") of another type in C have a fixed number of elements of any one type, and its representation stored contiguously in memory without gaps or padding. C allows multidimensional arrays whose size, and also arrays of pointers.

C supports dynamically allocated array whose size is determined at run time. C99 and later support arrays or VLA.

Chapter 12: Enumerations

Section 12.1: Simple Enumeration

An enumeration is a user-defined data type consists of integral constants and each integral constant is given a name. Keyword enum is used to define enumerated data type.

If you use enum instead of int or xing' char, you increase compile-time checking and avoid errors from passing in invalid constants, and you document which values are legal to use.

Example 1

```
enum color { RED, GREEN, BLUE };  
  
void printColor(enum color chosenColor)  
{  
    const char *color_name = "Invalid color";  
    switch(chosenColor)  
    {  
        case RED:  
            color_name = "RED";  
            break;  
  
        case GREEN:  
            color_name = "GREEN";  
            break;  
  
        case BLUE:  
            color_name = "BLUE";  
            break;  
    }  
    printf("%s\n", color_name);  
}
```

With a main function defined as follows (for example):

```
int main()  
{  
    enum color chosenColor;  
    printf("Enter a color between 0 and 2:");  
    scanf("%d", &chosenColor);  
    printColor(chosenColor);  
    return 0;  
}
```

Example 2

(This example uses designated initializers which are standardized since C99.)

```
enum week { MON, TUE, MED, THU, FRI, SAT, SUN };  
  
static const char* const day[] = {  
    [MON] = "Mon", [TUE] = "Tue", [MED] = "Wed",  
    [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };  
  
void printDay(enum week day)  
{  
    printf("%s\n", day);  
}
```

Example 3

(This program opens the file with name given in the argument to main, defaulting to output.txt if no argument is given. If the file with the same name already exists, its contents are discarded and the file is treated as a new empty file. If the fopen() call fails for some reason, it returns a NULL value and sets the global errno variable value. This means that the program can test the returned value after the fopen() call and use perror() if fopen() fails. If the fopen() call succeeds, it returns a valid FILE pointer. This pointer can then be used to reference this file until fclose() is called on it.)

The fopen() function when the given file to the opened file, replacing any previous contents of the file. Similarly to fopen(), the fputs() function also sets the errno value if it fails, though in this case the function returns EOF to the caller.

Variable Length Arrays (VLA for shorts) were added in C99, and made optional in C11. They are equal to normal arrays, with one, important difference. The length doesn't have to be known at compile time. VLAs have automatic storage duration. Only pointers to VLAs can have static storage duration.

VLAs are derived data types, representing an ordered collection of values ("elements") of another type in C have a fixed number of elements of any one type, and its representation stored contiguously in memory without gaps or padding. C allows multidimensional arrays whose size, and also arrays of pointers.

C supports dynamically allocated array whose size is determined at run time. C99 and later support arrays or VLA.

Chapter 20: Files and I/O streams

Parameter

const char *mode A string describing the opening mode of the file. SEEK_SET to set from the beginning of the file. SEEK_END to set from its end, or SEEK_CUR to set relative to the current cursor value. Note: SEEK_CUR is non-portable.

Section 20.1: Open and write to file

```
int main(int argc, char **argv)  
{  
    FILE *fp;  
    char *path = argv[1]; // for person1, fopen(), fputs() and ylabel() //  
    char *path2 = argv[2]; // for the EXIT_* macros //  
    int maxInt = 10;  
    fp = fopen(path, "w");  
    if (fp == NULL)  
    {  
        perror("Error opening file");  
        exit(EXIT_FAILURE);  
    }  
    for (int i = 0; i < maxInt; i++)  
    {  
        fprintf(fp, "%d\n", i);  
    }  
    if (fseek(fp, -1, SEEK_SET) != 0)  
    {  
        perror("Error seek");  
        exit(EXIT_FAILURE);  
    }  
    if (fputs(path2, fp) != EOF)  
    {  
        perror("Error fputs");  
        exit(EXIT_FAILURE);  
    }  
    if (fclose(fp) != 0)  
    {  
        perror("Error fclose");  
        exit(EXIT_FAILURE);  
    }  
}
```

With a main function defined as follows (for example):

```
int main()  
{  
    enum color chosenColor;  
    printf("Enter a color between 0 and 2:");  
    scanf("%d", &chosenColor);  
    printColor(chosenColor);  
    return 0;  
}
```

Example 2

(This example uses designated initializers which are standardized since C99.)

```
enum week { MON, TUE, MED, THU, FRI, SAT, SUN };  
  
static const char* const day[] = {  
    [MON] = "Mon", [TUE] = "Tue", [MED] = "Wed",  
    [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };  
  
void printDay(enum week day)  
{  
    printf("%s\n", day);  
}
```

Example 3

(This program opens the file with name given in the argument to main, defaulting to output.txt if no argument is given. If the file with the same name already exists, its contents are discarded and the file is treated as a new empty file. If the fopen() call fails for some reason, it returns a NULL value and sets the global errno variable value. This means that the program can test the returned value after the fopen() call and use perror() if fopen() fails. If the fopen() call succeeds, it returns a valid FILE pointer. This pointer can then be used to reference this file until fclose() is called on it.)

The fopen() function when the given file to the opened file, replacing any previous contents of the file. Similarly to fopen(), the fputs() function also sets the errno value if it fails, though in this case the function returns EOF to the caller.

Variable Length Arrays (VLA for shorts) were added in C99, and made optional in C11. They are equal to normal arrays, with one, important difference. The length doesn't have to be known at compile time. VLAs have automatic storage duration. Only pointers to VLAs can have static storage duration.

VLAs are derived data types, representing an ordered collection of values ("elements") of another type in C have a fixed number of elements of any one type, and its representation stored contiguously in memory without gaps or padding. C allows multidimensional arrays whose size, and also arrays of pointers.

C supports dynamically allocated array whose size is determined at run time. C99 and later support arrays or VLA.

Chapter 12: Enumerations

Section 12.1: Simple Enumeration

An enumeration is a user-defined data type consists of integral constants and each integral constant is given a name. Keyword enum is used to define enumerated data type.

If you use enum instead of int or xing' char, you increase compile-time checking and avoid errors from passing in invalid constants, and you document which values are legal to use.

Example 1

```
enum color { RED, GREEN, BLUE };  
  
void printColor(enum color chosenColor)  
{  
    const char *color_name = "Invalid color";  
    switch(chosenColor)  
    {  
        case RED:  
            color_name = "RED";  
            break;  
  
        case GREEN:  
            color_name = "GREEN";  
            break;  
  
        case BLUE:  
            color_name = "BLUE";  
            break;  
    }  
    printf("%s\n", color_name);  
}
```

With a main function defined as follows (for example):

```
int main()  
{  
    enum color chosenColor;  
    printf("Enter a color between 0 and 2:");  
    scanf("%d", &chosenColor);  
    printColor(chosenColor);  
    return 0;  
}
```

Example 2

(This example uses designated initializers which are standardized since C99.)

```
enum week { MON, TUE, MED, THU, FRI, SAT, SUN };  
  
static const char* const day[] = {  
    [MON] = "Mon", [TUE] = "Tue", [MED] = "Wed",  
    [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };  
  
void printDay(enum week day)  
{  
    printf("%s\n", day);  
}
```

Example 3

(This program opens the file with name given in the argument to main, defaulting to output.txt if no argument is given. If the file with the same name already exists, its contents are discarded and the file is treated as a new empty file. If the fopen() call fails for some reason, it returns a NULL value and sets the global errno variable value. This means that the program can test the returned value after the fopen() call and use perror() if fopen() fails. If the fopen() call succeeds, it returns a valid FILE pointer. This pointer can then be used to reference this file until fclose() is called on it.)

The fopen() function when the given file to the opened file, replacing any previous contents of the file. Similarly to fopen(), the fputs() function also sets the errno value if it fails, though in this case the function returns EOF to the caller.

Variable Length Arrays (VLA for shorts) were added in C99, and made optional in C11. They are equal to normal arrays, with one, important difference. The length doesn't have to be known at compile time. VLAs have automatic storage duration. Only pointers to VLAs can have static storage duration.

VLAs are derived data types, representing an ordered collection of values ("elements") of another type in C have a fixed number of elements of any one type, and its representation stored contiguously in memory without gaps or padding. C allows multidimensional arrays whose size, and also arrays of pointers.

C supports dynamically allocated array whose size is determined at run time. C99 and later support arrays or VLA.

Chapter 20: Files and I/O streams

Parameter
const char *mode A string describing the opening mode of the file. SEEK_SET to set from the beginning of the file. SEEK_END to set from its end, or SEEK_CUR to set relative to the current cursor value. Note: SEEK_CUR is non-portable.

Section 20.1: Open and write to file

```
int main(int argc, char **argv)  
{  
    FILE *fp;  
    char *path = argv[1]; // for person1, fopen(), fputs() and ylabel() //  
    char *path2 = argv[2]; // for the EXIT_* macros //  
    int maxInt = 10;  
    fp = fopen(path, "w");  
    if (fp == NULL)  
    {  
        perror("Error opening file");  
        exit(EXIT_FAILURE);  
    }  
    for (int i = 0; i < maxInt; i++)  
    {  
        fprintf(fp, "%d\n", i);  
    }  
    if (fseek(fp, -1, SEEK_SET) != 0)  
    {  
        perror("Error seek");  
        exit(EXIT_FAILURE);  
    }  
    if (fputs(path2, fp) != EOF)  
    {  
        perror("Error fputs");  
        exit(EXIT_FAILURE);  
    }  
    if (fclose(fp) != 0)  
    {  
        perror("Error fclose");  
        exit(EXIT_FAILURE);  
    }  
}
```

With a main function defined as follows (for example):

```
int main()  
{  
    enum color chosenColor;  
    printf("Enter a color between 0 and 2:");  
    scanf("%d", &chosenColor);  
    printColor(chosenColor);  
    return 0;  
}
```

Example 2

(This example uses designated initializers which are standardized since C99.)

```
enum week { MON, TUE, MED, THU, FRI, SAT, SUN };  
  
static const char* const day[] = {  
    [MON] = "Mon", [TUE] = "Tue", [MED] = "Wed",  
    [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };  
  
void printDay(enum week day)  
{  
    printf("%s\n", day);  
}
```

Example 3

(This program opens the file with name given in the argument to main, defaulting to output.txt if no argument is given. If the file with the same name already exists, its contents are discarded and the file is treated as a new empty file. If the fopen() call fails for some reason, it returns a NULL value and sets the global errno variable value. This means that the program can test the returned value after the fopen() call and use perror() if fopen() fails. If the fopen() call succeeds, it returns a valid FILE pointer. This pointer can then be used to reference this file until fclose() is called on it.)

The fopen() function when the given file to the opened file, replacing any previous contents of the file. Similarly to fopen(), the fputs() function also sets the errno value if it fails, though in this case the function returns EOF to the caller.

Variable Length Arrays (VLA for shorts) were added in C99, and made optional in C11. They are equal to normal arrays, with one, important difference. The length doesn't have to be known at compile time. VLAs have automatic storage duration. Only pointers to VLAs can have static storage duration.

VLAs are derived data types, representing an ordered collection of values ("elements") of another type in C have a fixed number of elements of any one type, and its representation stored contiguously in memory without gaps or padding. C allows multidimensional arrays whose size, and also arrays of pointers.

C supports dynamically allocated array whose size is determined at run time. C99 and later support arrays or VLA.

300多页
专业提示和技巧

300+ pages
of professional hints and tricks

目录

关于	1
第1章：C语言入门	2
第1.1节：你好，世界	2
第1.2节：K&R C中的原始“你好，世界！”	4
第2章：注释	6
第2.1节：使用预处理器进行注释	6
第2.2节：/* */ 定界注释	6
第2.3节：// 定界注释	7
第2.4节：三字符组合可能导致的陷阱	7
第3章：数据类型	9
第3.1节：声明的解释	9
第3.2节：固定宽度整数类型（自C99起）	11
第3.3节：整数类型和常量	11
第3.4节：浮点常量	12
第3.5节：字符串字面量	13
第4章：运算符	14
第4.1节：关系运算符	14
第4.2节：条件运算符/三元运算符	15
第4.3节：按位运算符	16
第4.4节：逻辑运算符的短路行为	18
第4.5节：逗号运算符	19
第4.6节：算术运算符	19
第4.7节：访问运算符	22
第4.8节：sizeof运算符	24
第4.9节：类型转换运算符	24
第4.10节：函数调用运算符	24
第4.11节：自增/自减	25
第4.12节：赋值运算符	25
第4.13节：逻辑运算符	26
第4.14节：指针运算	27
第4.15节：Alignof	28
第5章：布尔	30
第5.1节：使用 stdbool.h	30
第5.2节：使用 #define	30
第5.3节：使用内置类型 Bool	31
第5.4节：布尔表达式中的整数和指针	31
第5.5节：使用typedef定义bool类型	32
第6章：字符串	33
第6.1节：分词：strtok()、strtok_r()和strtok_s()	33
第6.2节：字符串字面量	35
第6.3节：计算长度：strlen()	36
第6.4节：字符串基础介绍	37
第6.5节：字符串复制	37
第6.6节：遍历字符串中的字符	40
第6.7节：创建字符串数组	41
第6.8节：字符串转换为数字：atoi(), atof()（危险，不要使用）	41
第6.9节：字符串格式化数据的读写	42

Contents

About	1
Chapter 1: Getting started with C Language	2
Section 1.1: Hello World	2
Section 1.2: Original "Hello, World!" in K&R C	4
Chapter 2: Comments	6
Section 2.1: Commenting using the preprocessor	6
Section 2.2: /* */ delimited comments	6
Section 2.3: // delimited comments	7
Section 2.4: Possible pitfall due to trigraphs	7
Chapter 3: Data Types	9
Section 3.1: Interpreting Declarations	9
Section 3.2: Fixed Width Integer Types (since C99)	11
Section 3.3: Integer types and constants	11
Section 3.4: Floating Point Constants	12
Section 3.5: String Literals	13
Chapter 4: Operators	14
Section 4.1: Relational Operators	14
Section 4.2: Conditional Operator/Ternary Operator	15
Section 4.3: Bitwise Operators	16
Section 4.4: Short circuit behavior of logical operators	18
Section 4.5: Comma Operator	19
Section 4.6: Arithmetic Operators	19
Section 4.7: Access Operators	22
Section 4.8: sizeof Operator	24
Section 4.9: Cast Operator	24
Section 4.10: Function Call Operator	24
Section 4.11: Increment / Decrement	25
Section 4.12: Assignment Operators	25
Section 4.13: Logical Operators	26
Section 4.14: Pointer Arithmetic	27
Section 4.15: Alignof	28
Chapter 5: Boolean	30
Section 5.1: Using stdbool.h	30
Section 5.2: Using #define	30
Section 5.3: Using the Intrinsic (built-in) Type Bool	31
Section 5.4: Integers and pointers in Boolean expressions	31
Section 5.5: Defining a bool type using typedef	32
Chapter 6: Strings	33
Section 6.1: Tokenisation: strtok(), strtok_r() and strtok_s()	33
Section 6.2: String literals	35
Section 6.3: Calculate the Length: strlen()	36
Section 6.4: Basic introduction to strings	37
Section 6.5: Copying strings	37
Section 6.6: Iterating Over the Characters in a String	40
Section 6.7: Creating Arrays of Strings	41
Section 6.8: Convert Strings to Number: atoi(), atof() (dangerous, don't use them)	41
Section 6.9: string formatted data read/write	42

第6.10节：查找特定字符的首次/末次出现：strchr(), strrchr()	43
第6.11节：复制与连接：strcpy(), strcat()	44
第6.12节：比较函数：strcmp(), strncmp(), strcasecmp(), strncasecmp()	45
第6.13节：安全地将字符串转换为数字：strtod()函数	47
第6.14节：strspn和strcspn	48
第7章：数字、字符和字符串的字面量	50
第7.1节：浮点字面量	50
第7.2节：字符串字面量	50
第7.3节：字符字面量	50
第7.4节：整数字面量	51
第8章：复合字面量	53
第8.1节：复合字面量的定义/初始化	53
第9章：位域	55
第9.1节：位域	55
第9.2节：将位域用作小整数	56
第9.3节：位域对齐	56
第9.4节：位域的禁忌事项	57
第9.5节：位域何时有用？	58
第10章：数组	60
第10.1节：声明和初始化数组	60
第10.2节：高效遍历数组及行优先顺序	61
第10.3节：数组长度	62
第10.4节：将多维数组传递给函数	63
第10.5节：多维数组	64
第10.6节：定义数组并访问数组元素	67
第10.7节：清除数组内容（置零）	67
第10.8节：设置数组中的值	68
第10.9节：分配并零初始化用户定义大小的数组	68
第10.10节：使用指针遍历数组	69
第11章：链表	71
第11.1节：双向链表	71
第11.2节：链表反转	73
第11.3节：在第n个位置插入节点	75
第11.4节：在单向链表开头插入节点	76
第12章：枚举	79
第12.1节：简单枚举	79
第12.2节：无类型的枚举常量	80
第12.3节：具有重复值的枚举	80
第12.4节：typedef 枚举	81
第13章：结构体	83
第13.1节：灵活数组成员	83
第13.2节：typedef结构体	85
第13.3节：指向结构体的指针	86
第13.4节：将结构体传递给函数	88
第13.5节：使用结构体的面向对象编程	89
第13.6节：简单数据结构	91
第14章：标准数学	93
第14.1节：幂函数 - pow(), powf(), powl()	93
第14.2节：双精度浮点余数：fmod()	94

Section 6.10: Find first/last occurrence of a specific character: strchr(), strrchr()	43
Section 6.11: Copy and Concatenation: strcpy(), strcat()	44
Section 6.12: Comparison: strcmp(), strncmp(), strcasecmp(), strncasecmp()	45
Section 6.13: Safely convert Strings to Number: strtod() functions	47
Section 6.14: strspn and strcspn	48
Chapter 7: Literals for numbers, characters and strings	50
Section 7.1: Floating point literals	50
Section 7.2: String literals	50
Section 7.3: Character literals	50
Section 7.4: Integer literals	51
Chapter 8: Compound Literals	53
Section 8.1: Definition/Initialisation of Compound Literals	53
Chapter 9: Bit-fields	55
Section 9.1: Bit-fields	55
Section 9.2: Using bit-fields as small integers	56
Section 9.3: Bit-field alignment	56
Section 9.4: Don'ts for bit-fields	57
Section 9.5: When are bit-fields useful?	58
Chapter 10: Arrays	60
Section 10.1: Declaring and initializing an array	60
Section 10.2: Iterating through an array efficiently and row-major order	61
Section 10.3: Array length	62
Section 10.4: Passing multidimensional arrays to a function	63
Section 10.5: Multi-dimensional arrays	64
Section 10.6: Define array and access array element	67
Section 10.7: Clearing array contents (zeroing)	67
Section 10.8: Setting values in arrays	68
Section 10.9: Allocate and zero-initialize an array with user defined size	68
Section 10.10: Iterating through an array using pointers	69
Chapter 11: Linked lists	71
Section 11.1: A doubly linked list	71
Section 11.2: Reversing a linked list	73
Section 11.3: Inserting a node at the nth position	75
Section 11.4: Inserting a node at the beginning of a singly linked list	76
Chapter 12: Enumerations	79
Section 12.1: Simple Enumeration	79
Section 12.2: enumeration constant without typename	80
Section 12.3: Enumeration with duplicate value	80
Section 12.4: Typedef enum	81
Chapter 13: Structs	83
Section 13.1: Flexible Array Members	83
Section 13.2: Typedef Structs	85
Section 13.3: Pointers to structs	86
Section 13.4: Passing structs to functions	88
Section 13.5: Object-based programming using structs	89
Section 13.6: Simple data structures	91
Chapter 14: Standard Math	93
Section 14.1: Power functions - pow(), powf(), powl()	93
Section 14.2: Double precision floating-point remainder: fmod()	94

第14.3节：单精度和长双精度浮点余数 : fmodf(), fmodl()	94
第15章：迭代语句/循环：for, while, do-while	96
第15.1节：for循环	96
第15.2节：循环展开与Duff's装置	96
第15.3节：while循环	97
第15.4节：Do-While循环	97
第15.5节：for循环的结构与控制流程	98
第15.6节：无限循环	99
第16章：选择语句	100
第16.1节：if () 语句	100
第16.2节：嵌套if()...else与if()..else阶梯结构	100
第16.3节：switch () 语句	102
第16.4节：if () ... else语句及语法	104
第16.5节：if()...else阶梯式链式两个或多个if () ... else语句	104
第17章：初始化	105
第17.1节：C语言中的变量初始化	105
第17.2节：使用指定初始化器	106
第17.3节：初始化结构体和结构体数组	108
第18章：声明与定义	110
第18.1节：理解声明和定义	110
第19章：命令行参数	111
第19.1节：打印程序参数并转换为整数值	111
第19.2节：打印命令行参数	111
第19.3节：使用GNU getopt工具	112
第20章：文件和输入输出流	115
第20.1节：打开并写入文件	115
第20.2节：运行进程	116
第20.3节：fprintf	116
第20.4节：使用getline()从文件获取行	116
第20.5节：fscanf()	120
第20.6节：从文件读取行	121
第20.7节：打开并写入二进制文件	122
第21章：格式化输入/输出	124
第21.1节：打印的转换说明符	124
第21.2节：printf()函数	125
第21.3节：打印格式标志	125
第21.4节：打印指向对象的指针的值	126
第21.5节：打印两个指向对象的指针值的差	127
第21.6节：长度修饰符	128
第22章：指针	129
第22.1节：介绍	129
第22.2节：常见错误	131
第22.3节：指针解引用	134
第22.4节：结构体指针解引用	134
第22.5节：常量指针	135
第22.6节：函数指针	138
第22.7节：使用void指针的多态行为	139
第22.8节：取地址运算符（&）	140
第22.9节：指针初始化	140

Section 14.3: Single precision and long double precision floating-point remainder: fmodf(), fmodl()	94
Chapter 15: Iteration Statements/Loops: for, while, do-while	96
Section 15.1: For loop	96
Section 15.2: Loop Unrolling and Duff's Device	96
Section 15.3: While loop	97
Section 15.4: Do-While loop	97
Section 15.5: Structure and flow of control in a for loop	98
Section 15.6: Infinite Loops	99
Chapter 16: Selection Statements	100
Section 16.1: if () Statements	100
Section 16.2: Nested if()...else VS if()..else Ladder	100
Section 16.3: switch () Statements	102
Section 16.4: if () ... else statements and syntax	104
Section 16.5: if()...else Ladder Chaining two or more if () ... else statements	104
Chapter 17: Initialization	105
Section 17.1: Initialization of Variables in C	105
Section 17.2: Using designated initializers	106
Section 17.3: Initializing structures and arrays of structures	108
Chapter 18: Declaration vs Definition	110
Section 18.1: Understanding Declaration and Definition	110
Chapter 19: Command-line arguments	111
Section 19.1: Print the arguments to a program and convert to integer values	111
Section 19.2: Printing the command line arguments	111
Section 19.3: Using GNU getopt tools	112
Chapter 20: Files and I/O streams	115
Section 20.1: Open and write to file	115
Section 20.2: Run process	116
Section 20.3: fprintf	116
Section 20.4: Get lines from a file using getline()	116
Section 20.5: fscanf()	120
Section 20.6: Read lines from a file	121
Section 20.7: Open and write to a binary file	122
Chapter 21: Formatted Input/Output	124
Section 21.1: Conversion Specifiers for printing	124
Section 21.2: The printf() Function	125
Section 21.3: Printing format flags	125
Section 21.4: Printing the Value of a Pointer to an Object	126
Section 21.5: Printing the Difference of the Values of two Pointers to an Object	127
Section 21.6: Length modifiers	128
Chapter 22: Pointers	129
Section 22.1: Introduction	129
Section 22.2: Common errors	131
Section 22.3: Dereferencing a Pointer	134
Section 22.4: Dereferencing a Pointer to a struct	134
Section 22.5: Const Pointers	135
Section 22.6: Function pointers	138
Section 22.7: Polymorphic behaviour with void pointers	139
Section 22.8: Address-of Operator (&)	140
Section 22.9: Initializing Pointers	140

第22.10节：指向指针的指针	141
第22.11节：作为参数和返回值的void*指针与标准函数	141
第22.12节：相同的星号，不同的含义	142
第23章：序列点	144
第23.1节：无序表达式	144
第23.2节：有序表达式	144
第23.3节：不确定顺序的表达式	145
第24章：函数指针	146
第24.1节：介绍	146
第24.2节：从函数返回函数指针	146
第24.3节：最佳实践	147
第24.4节：赋值函数指针	149
第24.5节：编写函数指针的助记符	149
第24.6节：基础知识	150
第25章：函数参数	152
第25.1节：参数按值传递	152
第25.2节：向函数传递数组	152
第25.3节：函数参数执行顺序	153
第25.4节：使用指针参数返回多个值	153
第25.5节：函数返回包含错误代码值的结构体示例	154
第26章：将二维数组传递给函数	156
第26.1节：将二维数组传递给函数	156
第26.2节：将一维数组用作二维数组	162
第27章：错误处理	163
第27.1节：errno	163
第27.2节：strerror	163
第27.3节： perror	163
第28章：未定义行为	165
第28.1节：解引用超出其生命周期的变量指针	165
第28.2节：复制重叠的内存	165
第28.3节：有符号整数溢出	166
第28.4节：使用未初始化的变量	167
第28.5节：数据竞争	168
第28.6节：读取已释放指针的值	169
第28.7节：在printf中使用错误的格式说明符	170
第28.8节：修改字符串字面量	170
第28.9节：向printf的%s转换传递空指针	170
第28.10节：在两个序列点之间多次修改同一对象	171
第28.11节：重复释放内存	172
第28.12节：使用负数或超出类型宽度的位移操作	172
第28.13节：从声明为`_Noreturn`或`noreturn`函数说明符的函数返回	173
第28.14节：访问超出分配块的内存	174
第28.15节：通过指针修改const变量	174
第28.16节：读取未初始化且未由内存支持的对象	175
第28.17节：指针的加减未正确限定范围	175
第28.18节：解引用空指针	175
第28.19节：在输入流上使用flush	176
第28.20节：标识符链接不一致	176
第28.21节：返回值函数缺少返回语句	177

Section 22.10: Pointer to Pointer	141
Section 22.11: void* pointers as arguments and return values to standard functions	141
Section 22.12: Same Asterisk, Different Meanings	142
Chapter 23: Sequence points	144
Section 23.1: Unsequenced expressions	144
Section 23.2: Sequenced expressions	144
Section 23.3: Indeterminately sequenced expressions	145
Chapter 24: Function Pointers	146
Section 24.1: Introduction	146
Section 24.2: Returning Function Pointers from a Function	146
Section 24.3: Best Practices	147
Section 24.4: Assigning a Function Pointer	149
Section 24.5: Mnemonic for writing function pointers	149
Section 24.6: Basics	150
Chapter 25: Function Parameters	152
Section 25.1: Parameters are passed by value	152
Section 25.2: Passing in Arrays to Functions	152
Section 25.3: Order of function parameter execution	153
Section 25.4: Using pointer parameters to return multiple values	153
Section 25.5: Example of function returning struct containing values with error codes	154
Chapter 26: Pass 2D-arrays to functions	156
Section 26.1: Pass a 2D-array to a function	156
Section 26.2: Using flat arrays as 2D arrays	162
Chapter 27: Error handling	163
Section 27.1: errno	163
Section 27.2: strerror	163
Section 27.3: perror	163
Chapter 28: Undefined behavior	165
Section 28.1: Dereferencing a pointer to variable beyond its lifetime	165
Section 28.2: Copying overlapping memory	165
Section 28.3: Signed integer overflow	166
Section 28.4: Use of an uninitialized variable	167
Section 28.5: Data race	168
Section 28.6: Read value of pointer that was freed	169
Section 28.7: Using incorrect format specifier in printf	170
Section 28.8: Modify string literal	170
Section 28.9: Passing a null pointer to printf %s conversion	170
Section 28.10: Modifying any object more than once between two sequence points	171
Section 28.11: Freeing memory twice	172
Section 28.12: Bit shifting using negative counts or beyond the width of the type	172
Section 28.13: Returning from a function that's declared with `_Noreturn` or `noreturn` function specifier	173
Section 28.14: Accessing memory beyond allocated chunk	174
Section 28.15: Modifying a const variable using a pointer	174
Section 28.16: Reading an uninitialized object that is not backed by memory	175
Section 28.17: Addition or subtraction of pointer not properly bounded	175
Section 28.18: Dereferencing a null pointer	175
Section 28.19: Using fflush on an input stream	176
Section 28.20: Inconsistent linkage of identifiers	176
Section 28.21: Missing return statement in value returning function	177

第28.22节：除以零	177
第28.23节：指针类型转换导致结果未正确对齐	178
第28.24节：修改由getenv、strerror和setlocale函数返回的字符串	179
第29章：随机数生成	180
第29.1节：基本随机数生成	180
第29.2节：置换同余生成器	180
第29.3节：异或移位生成	181
第29.4节：限制生成到给定范围内	182
第30章：预处理器和宏	183
第30.1节：头文件包含保护	183
第30.2节：使用#if 0屏蔽代码段	186
第30.3节：类函数宏	187
第30.4节：源文件包含	188
第30.5节：条件包含和条件函数签名修改	188
第30.6节：cplusplus 用于在用C++编译的C++代码中使用C外部函数——名称修饰	190
第30.7节：标记粘贴	191
第30.8节：预定义宏	192
第30.9节：可变参数宏	193
第30.10节：宏替换	194
第30.11节：错误指令	195
第30.12节：FOREACH实现	196
第31章：信号处理	199
第31.1节：使用“signal()”进行信号处理	199
第32章：可变参数	201
第32.1节：使用显式计数参数确定va_list的长度	201
第32.2节：使用终止值确定va_list的结束	202
第32.3节：实现类似`printf()`接口的函数	202
第32.4节：使用格式字符串	205
第33章：断言	207
第33.1节：简单断言	207
第33.2节：静态断言	207
第33.3节：断言错误信息	208
第33.4节：不可达代码的断言	209
第33.5节：前置条件和后置条件	209
第34章：泛型选择	211
第34.1节：检查变量是否为某种限定类型	211
第34.2节：基于多个参数的泛型选择	211
第34.3节：类型泛型打印宏	213
第35章：X宏	214
第35.1节：X宏在printf中的简单使用	214
第35.2节：扩展：将X宏作为参数传递	214
第35.3节：枚举值和标识符	215
第35.4节：代码生成	215
第36章：别名和有效类型	217
第36.1节：有效类型	217
第36.2节：restrict限定符	217
第36.3节：修改字节	218
第36.4节：字符类型不能通过非字符类型访问	219
第36.5节：违反严格别名规则	220

Section 28.22: Division by zero	177
Section 28.23: Conversion between pointer types produces incorrectly aligned result	178
Section 28.24: Modifying the string returned by getenv, strerror, and setlocale functions	179
Chapter 29: Random Number Generation	180
Section 29.1: Basic Random Number Generation	180
Section 29.2: Permuted Congruential Generator	180
Section 29.3: Xorshift Generation	181
Section 29.4: Restrict generation to a given range	182
Chapter 30: Preprocessor and Macros	183
Section 30.1: Header Include Guards	183
Section 30.2: #if 0 to block out code sections	186
Section 30.3: Function-like macros	187
Section 30.4: Source file inclusion	188
Section 30.5: Conditional inclusion and conditional function signature modification	188
Section 30.6: cplusplus for using C externals in C++ code compiled with C++ - name mangling	190
Section 30.7: Token pasting	191
Section 30.8: Predefined Macros	192
Section 30.9: Variadic arguments macro	193
Section 30.10: Macro Replacement	194
Section 30.11: Error directive	195
Section 30.12: FOREACH implementation	196
Chapter 31: Signal handling	199
Section 31.1: Signal Handling with “signal()”	199
Chapter 32: Variable arguments	201
Section 32.1: Using an explicit count argument to determine the length of the va_list	201
Section 32.2: Using terminator values to determine the end of va_list	202
Section 32.3: Implementing functions with a `printf()`-like interface	202
Section 32.4: Using a format string	205
Chapter 33: Assertion	207
Section 33.1: Simple Assertion	207
Section 33.2: Static Assertion	207
Section 33.3: Assert Error Messages	208
Section 33.4: Assertion of Unreachable Code	209
Section 33.5: Precondition and Postcondition	209
Chapter 34: Generic selection	211
Section 34.1: Check whether a variable is of a certain qualified type	211
Section 34.2: Generic selection based on multiple arguments	211
Section 34.3: Type-generic printing macro	213
Chapter 35: X-macros	214
Section 35.1: Trivial use of X-macros for printf	214
Section 35.2: Extension: Give the X macro as an argument	214
Section 35.3: Enum Value and Identifier	215
Section 35.4: Code generation	215
Chapter 36: Aliasing and effective type	217
Section 36.1: Effective type	217
Section 36.2: restrict qualification	217
Section 36.3: Changing bytes	218
Section 36.4: Character types cannot be accessed through non-character types	219
Section 36.5: Violating the strict aliasing rules	220

第37章：编译	221
第37.1节：编译器	221
第37.2节：文件类型	222
第37.3节：链接器	222
第37.4节：预处理器	224
第37.5节：翻译阶段	225
第38章：内联汇编	227
第38.1节：宏中的gcc内联汇编	227
第38.2节：gcc基本汇编支持	227
第38.3节：gcc扩展汇编支持	228
第39章：标识符作用域	229
第39.1节：函数原型作用域	229
第39.2节：块作用域	230
第39.3节：文件作用域	230
第39.4节：函数作用域	231
第40章：隐式和显式转换	232
第40.1节：函数调用中的整数转换	232
第40.2节：函数调用中的指针转换	233
第41章：类型限定符	235
第41.1节：易变变量	235
第41.2节：不可修改 (const) 变量	236
第42章：类型定义 (typedef)	237
第42.1节：结构体和联合体的typedef	237
第42.2节：函数指针的typedef	238
第42.3节：typedef的简单用法	239
第43章：存储类	241
第43.1节：auto	241
第43.2节：register	241
第43.3节：static	242
第43.4节：typedef	243
第43.5节：extern	243
第43.6节：Thread_local	244
第44章：声明	246
第44.1节：从另一个C文件调用函数	246
第44.2节：使用全局变量	247
第44.3节：简介	247
第44.4节：typedef	250
第44.5节：使用全局常量	250
第44.6节：使用左右规则或螺旋规则解析C语言声明	252
第45章：结构体填充与打包	256
第45.1节：结构体打包	256
第45.2节：结构体填充	257
第46章：内存管理	258
第46.1节：分配内存	258
第46.2节：释放内存	259
第46.3节：重新分配内存	261
第46.4节：realloc(ptr, 0) 不等同于 free(ptr)	262
第46.5节：可变大小的多维数组	262
第46.6节：alloca：在栈上分配内存	263

Chapter 37: Compilation	221
Section 37.1: The Compiler	221
Section 37.2: File Types	222
Section 37.3: The Linker	222
Section 37.4: The Preprocessor	224
Section 37.5: The Translation Phases	225
Chapter 38: Inline assembly	227
Section 38.1: gcc Inline assembly in macros	227
Section 38.2: gcc Basic asm support	227
Section 38.3: gcc Extended asm support	228
Chapter 39: Identifier Scope	229
Section 39.1: Function Prototype Scope	229
Section 39.2: Block Scope	230
Section 39.3: File Scope	230
Section 39.4: Function scope	231
Chapter 40: Implicit and Explicit Conversions	232
Section 40.1: Integer Conversions in Function Calls	232
Section 40.2: Pointer Conversions in Function Calls	233
Chapter 41: Type Qualifiers	235
Section 41.1: Volatile variables	235
Section 41.2: Unmodifiable (const) variables	236
Chapter 42: Typedef	237
Section 42.1: Typedef for Structures and Unions	237
Section 42.2: Typedef for Function Pointers	238
Section 42.3: Simple Uses of Typedef	239
Chapter 43: Storage Classes	241
Section 43.1: auto	241
Section 43.2: register	241
Section 43.3: static	242
Section 43.4: typedef	243
Section 43.5: extern	243
Section 43.6: Thread_local	244
Chapter 44: Declarations	246
Section 44.1: Calling a function from another C file	246
Section 44.2: Using a Global Variable	247
Section 44.3: Introduction	247
Section 44.4: Typedef	250
Section 44.5: Using Global Constants	250
Section 44.6: Using the right-left or spiral rule to decipher C declaration	252
Chapter 45: Structure Padding and Packing	256
Section 45.1: Packing structures	256
Section 45.2: Structure padding	257
Chapter 46: Memory management	258
Section 46.1: Allocating Memory	258
Section 46.2: Freeing Memory	259
Section 46.3: Reallocating Memory	261
Section 46.4: realloc(ptr, 0) is not equivalent to free(ptr)	262
Section 46.5: Multidimensional arrays of variable size	262
Section 46.6: alloca: allocate memory on stack	263

第46.7节：用户定义的内存管理	264
第47章：实现定义的行为	266
第47.1节：负整数的右移	266
第47.2节：给整数赋值超出范围的值	266
第47.3节：分配零字节	266
第47.4节：有符号整数的表示	266
第48章：原子操作	267
第48.1节：原子操作和运算符	267
第49章：跳转语句	268
第49.1节：使用return	268
第49.2节：使用goto跳出嵌套循环	268
第49.3节：使用break和continue	269
第50章：创建和包含头文件	271
第50.1节：介绍	271
第50.2节：自包含性	271
第50.3节：最小性	273
第50.4节：符号和杂项	273
第50.5节：幂等性	275
第50.6节：包含你所使用的 (IWYU)	275
第51章：<ctype.h> — 字符分类与转换	277
第51.1节：介绍	277
第51.2节：从流中分类读取的字符	278
第51.3节：从字符串中分类字符	279
第52章：副作用	280
第52.1节：前置/后置自增/自减运算符	280
第53章：多字符字符序列	282
第53.1节：三字符组合	282
第53.2节：双字符组合	282
第54章：约束	284
第54.1节：同一作用域内重复的变量名	284
第54.2节：一元算术运算符	284
第55章：内联	285
第55.1节：在多个源文件中使用的内联函数	285
第56章：联合体	287
第56.1节：使用联合体重新解释值	287
第56.2节：向一个联合体成员写入并从另一个成员读取	287
第56.3节：结构体与联合体的区别	288
第57章：线程（本地）	289
第57.1节：由一个线程初始化	289
第57.2节：启动多个线程	289
第58章：多线程	291
第58.1节：C11线程简单示例	291
第59章：进程间通信（IPC）	292
第59.1节：信号量	292
第60章：测试框架	297
第60.1节：Unity测试框架	297
第60.2节：CMocka	297
第60.3节：CppUTest	298

Section 46.7: User-defined memory management	264
Chapter 47: Implementation-defined behaviour	266
Section 47.1: Right shift of a negative integer	266
Section 47.2: Assigning an out-of-range value to an integer	266
Section 47.3: Allocating zero bytes	266
Section 47.4: Representation of signed integers	266
Chapter 48: Atomics	267
Section 48.1: atomics and operators	267
Chapter 49: Jump Statements	268
Section 49.1: Using return	268
Section 49.2: Using goto to jump out of nested loops	268
Section 49.3: Using break and continue	269
Chapter 50: Create and include header files	271
Section 50.1: Introduction	271
Section 50.2: Self-containment	271
Section 50.3: Minimality	273
Section 50.4: Notation and Miscellany	273
Section 50.5: Idempotence	275
Section 50.6: Include What You Use (IWYU)	275
Chapter 51: <ctype.h> — character classification & conversion	277
Section 51.1: Introduction	277
Section 51.2: Classifying characters read from a stream	278
Section 51.3: Classifying characters from a string	279
Chapter 52: Side Effects	280
Section 52.1: Pre/Post Increment/Decrement operators	280
Chapter 53: Multi-Character Character Sequence	282
Section 53.1: Trigraphs	282
Section 53.2: Digraphs	282
Chapter 54: Constraints	284
Section 54.1: Duplicate variable names in the same scope	284
Section 54.2: Unary arithmetic operators	284
Chapter 55: Inlining	285
Section 55.1: Inlining functions used in more than one source file	285
Chapter 56: Unions	287
Section 56.1: Using unions to reinterpret values	287
Section 56.2: Writing to one union member and reading from another	287
Section 56.3: Difference between struct and union	288
Chapter 57: Threads (native)	289
Section 57.1: Initialization by one thread	289
Section 57.2: Start several threads	289
Chapter 58: Multithreading	291
Section 58.1: C11 Threads simple example	291
Chapter 59: Interprocess Communication (IPC)	292
Section 59.1: Semaphores	292
Chapter 60: Testing frameworks	297
Section 60.1: Unity Test Framework	297
Section 60.2: CMocka	297
Section 60.3: CppUTest	298

第61章：Valgrind	300
第61.1节：字节丢失——忘记释放	300
第61.2节：使用Valgrind时遇到的最常见错误	300
第61.3节：运行Valgrind	301
第61.4节：添加标志	301
第62章：常见的C语言编程习惯和开发者实践	302
第62.1节：比较字面量和变量	302
第62.2节：函数的参数列表不要留空——使用void	302
第63章：常见陷阱	305
第63.1节：算术运算中混合有符号和无符号整数	305
第63.2节：宏是简单的字符串替换	305
第63.3节：忘记将realloc的返回值复制到临时变量中	307
第63.4节：忘记为\0分配额外的一个字节	308
第63.5节：误解数组衰减	308
第63.6节：忘记释放内存（内存泄漏）	310
第63.7节：复制过多内容	311
第63.8节：比较时错误地写成-=而非==	312
第63.9节：典型的scanf()调用中换行符未被消耗	313
第63.10节：在#define中添加分号	314
第63.11节：分号的粗心使用	314
第63.12节：链接时的未定义引用错误	315
第63.13节：将逻辑表达式与“true”比较	317
第63.14节：在指针运算中进行额外的缩放	318
第63.15节：多行注释不能嵌套	319
第63.16节：忽略库函数的返回值	321
第63.17节：比较浮点数	321
第63.18节：浮点字面量默认类型为double	323
第63.19节：使用字符常量代替字符串字面量，反之亦然	323
第63.20节：递归函数——缺少基本条件	324
第63.21节：越界访问数组	325
第63.22节：向期望“真实”多维数组的函数传递非连续数组	326
致谢	328
你可能也喜欢	333

Chapter 61: Valgrind	300
Section 61.1: Bytes lost -- Forgetting to free	300
Section 61.2: Most common errors encountered while using Valgrind	300
Section 61.3: Running Valgrind	301
Section 61.4: Adding flags	301
Chapter 62: Common C programming idioms and developer practices	302
Section 62.1: Comparing literal and variable	302
Section 62.2: Do not leave the parameter list of a function blank – use void	302
Chapter 63: Common pitfalls	305
Section 63.1: Mixing signed and unsigned integers in arithmetic operations	305
Section 63.2: Macros are simple string replacements	305
Section 63.3: Forgetting to copy the return value of realloc into a temporary	307
Section 63.4: Forgetting to allocate one extra byte for \0	308
Section 63.5: Misunderstanding array decay	308
Section 63.6: Forgetting to free memory (memory leaks)	310
Section 63.7: Copying too much	311
Section 63.8: Mistakenly writing = instead of == when comparing	312
Section 63.9: Newline character is not consumed in typical scanf() call	313
Section 63.10: Adding a semicolon to a #define	314
Section 63.11: Incautious use of semicolons	314
Section 63.12: Undefined reference errors when linking	315
Section 63.13: Checking logical expression against 'true'	317
Section 63.14: Doing extra scaling in pointer arithmetic	318
Section 63.15: Multi-line comments cannot be nested	319
Section 63.16: Ignoring return values of library functions	321
Section 63.17: Comparing floating point numbers	321
Section 63.18: Floating point literals are of type double by default	323
Section 63.19: Using character constants instead of string literals, and vice versa	323
Section 63.20: Recursive function – missing out the base condition	324
Section 63.21: Overstepping array boundaries	325
Section 63.22: Passing unadjacent arrays to functions expecting "real" multidimensional arrays	326
Credits	328
You may also like	333

欢迎免费与任何人分享此PDF，
本书的最新版本可从以下网址下载：

<https://goalkicker.com/CBook>

本专业人士的C语言笔记一书汇编自[Stack Overflow Documentation](#)，内容由Stack Overflow的优秀人士撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

这是一本非官方的免费书籍，旨在教育用途，与官方C语言组织或公司及Stack Overflow无关。所有商标和注册商标均为其各自公司所有者所有。

本书所提供的信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至web@petercv.com

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:

<https://goalkicker.com/CBook>

This *C Notes for Professionals* book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official C group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：C语言入门

版本	标准	发布日期
K&R	不适用	1978-02-22
C89	ANSI X3.159-1989	1989-12-14
C90	ISO/IEC 9899:1990	1990-12-20
C95	ISO/IEC 9899/AMD1:1995	1995-03-30
C99	ISO/IEC 9899:1999	1999-12-16
C11	ISO/IEC 9899:2011	2011-12-15

第1.1节：Hello World

要创建一个简单的C程序，在屏幕上打印"Hello, World"，使用文本编辑器创建一个新文件（例如 hello.c — 文件扩展名必须是.c），包含以下源代码：

hello.c

```
#include <stdio.h>

int main(void)
{
    puts("Hello, World");
    return 0;
}
```

[Coliru 在线演示](#)

让我们逐行查看这个简单的程序

```
#include <stdio.h>
```

这一行告诉编译器在程序中包含标准库头文件 stdio.h 的内容。

头文件通常包含函数声明、宏和数据类型，使用它们之前必须包含相应的头文件。

这一行包含了 stdio.h，以便调用函数 puts()。

查看更多关于头文件的信息。

```
int main(void)
```

这一行开始定义一个函数。它说明了函数的名称（main）、期望的参数类型和数量（void，表示无参数），以及该函数返回值的类型（int）。程序执行从main()函数开始。

```
{
...
}
```

花括号成对使用，用于表示代码块的开始和结束。它们可以有多种用法，但在这里表示函数的开始和结束。

```
    puts("Hello, World");
```

这一行调用了puts()函数，将文本输出到标准输出（默认是屏幕），并在末尾添加换行符。

Chapter 1: Getting started with C Language

Version	Standard	Publication Date
K&R	n/a	1978-02-22
C89	ANSI X3.159-1989	1989-12-14
C90	ISO/IEC 9899:1990	1990-12-20
C95	ISO/IEC 9899/AMD1:1995	1995-03-30
C99	ISO/IEC 9899:1999	1999-12-16
C11	ISO/IEC 9899:2011	2011-12-15

Section 1.1: Hello World

To create a simple C program which prints "Hello, World" on the screen, use a [text editor](#) to create a new file (e.g. hello.c — the file extension must be .c) containing the following source code:

hello.c

```
#include <stdio.h>

int main(void)
{
    puts("Hello, World");
    return 0;
}
```

[Live demo on Coliru](#)

Let's look at this simple program line by line

```
#include <stdio.h>
```

This line tells the compiler to include the contents of the standard library header file stdio.h in the program. Headers are usually files containing function declarations, macros and data types, and you must include the header file before you use them. This line includes stdio.h so it can call the function puts().

See more about headers.

```
int main(void)
```

This line starts the definition of a function. It states the name of the function (main), the type and number of arguments it expects (void, meaning none), and the type of value that this function returns (int). Program execution starts in the main() function.

```
{
...
}
```

The curly braces are used in pairs to indicate where a block of code begins and ends. They can be used in a lot of ways, but in this case they indicate where the function begins and ends.

```
    puts("Hello, World");
```

This line calls the puts() function to output text to standard output (the screen, by default), followed by a newline.

要输出的字符串包含在括号内。

"Hello, World"是将写入屏幕的字符串。在C语言中，每个字符串字面值必须放在双引号"..."内。

查看更多关于字符串的信息。

在C程序中，每条语句都需要以分号（即;）结尾。

```
return 0;
```

当我们定义main()时，我们将其声明为返回int类型的函数，意味着它需要返回一个整数。在这个例子中，我们返回整数值0，用于表示程序成功退出。

在return 0;语句之后，执行过程将终止。

编辑程序

简单的文本编辑器包括Linux上的vim或gedit，或Windows上的Notepad。跨平台编辑器还包括Visual Studio Code或Sublime Text。

编辑器必须创建纯文本文件，而不是RTF或其他任何格式。

编译并运行程序

要运行程序，首先需要将此源文件（hello.c）编译成可执行文件（例如Unix/Linux系统上的hello或Windows上的hello.exe）。这需要使用C语言的编译器来完成。

查看更多关于编译的信息

使用GCC编译

GCC（GNU 编译器集合）是一款广泛使用的 C 语言编译器。使用时，打开终端，使用命令行导航到源文件所在位置，然后运行：

```
gcc hello.c -o hello
```

如果在源代码（hello.c）中未发现错误，编译器将创建一个二进制文件，其名称由-o命令行选项的参数指定（hello）。这就是最终的可执行文件。

我们还可以使用警告选项-Wall -Wextra -Werror，这些选项有助于识别可能导致程序失败或产生意外结果的问题。对于这个简单程序来说它们不是必需的，但添加它们的方法如下：

```
gcc -Wall -Wextra -Werror -o hello hello.c
```

使用 clang 编译器

要使用clang编译程序，可以使用：

```
clang -Wall -Wextra -Werror -o hello hello.c
```

设计上，clang命令行选项与GCC的类似。

使用命令行中的 Microsoft C 编译器

The string to be output is included within the parentheses.

"Hello, World" is the string that will be written to the screen. In C, every string literal value must be inside the double quotes "...".

See more about strings.

In C programs, every statement needs to be terminated by a semi-colon (i.e. ;).

```
return 0;
```

When we defined main(), we declared it as a function returning an int, meaning it needs to return an integer. In this example, we are returning the integer value 0, which is used to indicate that the program exited successfully. After the return 0; statement, the execution process will terminate.

Editing the program

Simple text editors include vim or gedit on Linux, or Notepad on Windows. Cross-platform editors also include Visual Studio Code or Sublime Text.

The editor must create plain text files, not RTF or other any other format.

Compiling and running the program

To run the program, this source file (hello.c) first needs to be compiled into an executable file (e.g. hello on Unix/Linux system or hello.exe on Windows). This is done using a compiler for the C language.

See more about compiling

Compile using GCC

GCC (GNU Compiler Collection) is a widely used C compiler. To use it, open a terminal, use the command line to navigate to the source file's location and then run:

```
gcc hello.c -o hello
```

If no errors are found in the the source code (hello.c), the compiler will create a binary file, the name of which is given by the argument to the -o command line option (hello). This is the final executable file.

We can also use the warning options -Wall -Wextra -Werror, that help to identify problems that can cause the program to fail or produce unexpected results. They are not necessary for this simple program but this is way of adding them:

```
gcc -Wall -Wextra -Werror -o hello hello.c
```

Using the clang compiler

To compile the program using clang you can use:

```
clang -Wall -Wextra -Werror -o hello hello.c
```

By design, the clang command line options are similar to those of GCC.

Using the Microsoft C compiler from the command line

如果在支持Visual Studio的Windows系统上使用Microsoft的cl.exe编译器，并且所有环境变量都已设置，则可以使用以下命令编译此C示例，该命令将在执行命令的目录中生成一个可执行文件hello.exe（对于cl有类似/W3的警告选项，大致相当于GCC或clang的-Wall等选项）。

```
cl hello.c
```

执行程序

编译完成后，可以通过在终端输入`./hello`来执行二进制文件。执行时，编译后的程序将向命令提示符打印Hello, World，随后换行。

第1.2节：K&R C语言中的原始“Hello, World!”程序

下面是Brian

Kernighan和Dennis Ritchie (Ritchie是贝尔实验室C语言的最初开发者) 所著书籍《C程序设计语言》中的原始“Hello, World!”程序，简称“K&R”：

```
版本 = K&R
#include <stdio.h>

main()
{
    printf("hello, world");}
```

请注意，在本书第一版（1978年）撰写时，C语言尚未标准化，该程序在大多数现代编译器上可能无法编译，除非它们被指示接受C90代码。

K&R书中的这个最初示例现在被认为质量较差，部分原因是它缺少对main()的显式返回类型，部分原因是缺少return语句。该书第二版是针对旧的C89标准编写的。在C89中，main的类型默认是int，但K&R示例没有向环境返回定义的值。在C99及以后的标准中，返回类型是必须的，但可以安全地省略main（仅限main）的return语句，因为C99 5.1.2.2.3引入了一个特殊情况——这等同于返回0，表示成功。

对于托管系统，推荐且最通用的main形式是int main (void)，当程序不使用任何命令行参数时；当程序使用命令行参数时，则是int main(int argc, char **argv)。

C90 §5.1.2.2.3 程序终止

从对main函数的初始调用返回，相当于调用exit函数，并以main函数返回的值作为参数。如果main函数执行了不指定返回值的return，则返回给主机环境的终止状态是未定义的。

C90 §6.6.6.4 return 语句

如果执行了一个没有表达式的return语句，并且函数调用的值被使用，

If using the Microsoft cl.exe compiler on a Windows system which supports Visual Studio and if all environment variables are set, this C example may be compiled using the following command which will produce an executable hello.exe within the directory the command is executed in (There are warning options such as /W3 for cl, roughly analogous to -Wall etc for GCC or clang).

```
cl hello.c
```

Executing the program

Once compiled, the binary file may then be executed by typing `./hello` in the terminal. Upon execution, the compiled program will print Hello, World, followed by a newline, to the command prompt.

Section 1.2: Original "Hello, World!" in K&R C

The following is the original "Hello, World!" program from the book [The C Programming Language](#) by Brian Kernighan and Dennis Ritchie (Ritchie was the original developer of the C programming language at Bell Labs), referred to as "K&R":

```
Version = K&R
#include <stdio.h>

main()
{
    printf("hello, world\n");}
```

Notice that the C programming language was not standardized at the time of writing the first edition of this book (1978), and that this program will probably not compile on most modern compilers unless they are instructed to accept C90 code.

This very first example in the K&R book is now considered poor quality, in part because it lacks an explicit return type for `main()` and in part because it lacks a `return` statement. The 2nd edition of the book was written for the old C89 standard. In C89, the type of `main` would default to `int`, but the K&R example does not return a defined value to the environment. In C99 and later standards, the return type is required, but it is safe to leave out the `return` statement of `main` (and only `main`), because of a special case introduced with C99 5.1.2.2.3 — it is equivalent to returning 0, which indicates success.

The recommended and most portable form of `main` for hosted systems is `int main (void)` when the program does not use any command line arguments, or `int main(int argc, char **argv)` when the program does use the command line arguments.

C90 §5.1.2.2.3 Program termination

A return from the initial call to the `main` function is equivalent to calling the `exit` function with the value returned by the `main` function as its argument. If the `main` function executes a return that specifies no value, the termination status returned to the host environment is undefined.

C90 §6.6.6.4 The `return` statement

If a `return` statement without an expression is executed, and the value of the function call is used by the

调用者，行为未定义。到达终止函数的}等同于执行一个无表达式的return语句。

C99 §5.1.2.2.3 程序终止

如果main函数的返回类型是与int兼容的类型，则从对main函数的初始调用返回相当于调用exit函数，参数为main函数返回的值；达到终止main函数的}时返回值为0。如果返回类型与int不兼容，则返回给主机环境的终止状态未指定。

caller, the behavior is undefined. Reaching the } that terminates a function is equivalent to executing a `return` statement without an expression.

C99 §5.1.2.2.3 Program termination

If the return type of the `main` function is a type compatible with `int`, a return from the initial call to the `main` function is equivalent to calling the `exit` function with the value returned by the `main` function as its argument; reaching the } that terminates the `main` function returns a value of 0. If the return type is not compatible with `int`, the termination status returned to the host environment is unspecified.

第二章：评论

注释用于向阅读代码的人指示某些内容。注释在编译器看来就像空白，不会改变代码的实际含义。C语言中有两种注释语法，原始的`/* */`和稍新的`//`。一些文档系统使用特殊格式的注释来帮助生成代码文档。

第2.1节：使用预处理器进行注释

也可以使用预处理器指令`#if 0`和`#endif`“注释掉”大块代码。当代码中包含多行注释且无法嵌套时，这非常有用。

```
#if 0 /* 开始“注释”，从这里开始的内容将被预处理器移除 */

/* 大量包含多行注释的代码 */
int foo()
{
    /* 大量代码 */

    ...

    /* ... 一些描述if语句的注释 ... */
    if (someTest) {
        /* 一些更多的注释 */
        return 1;
    }

    return 0;
}

#endif /* 0 */

/* 从这里开始的代码是“取消注释”的（包含在编译的可执行文件中） */
...
```

第2.2节：/* */ 分隔的注释

注释以正斜杠紧跟星号`(/*)`开始，遇到星号紧跟正斜杠`(*)`时结束。这两个字符组合之间的所有内容都是注释，编译器将其视为空白（基本忽略）。

```
/* 这是一个注释 */
```

上面的注释是单行注释。这种`/*`类型的注释可以跨多行，如下所示：

```
/* 这是一个
多行
注释 */
```

虽然不是严格必要，但多行注释的常见风格约定是在第一行之后的各行前加空格和星号，并将`/*`和`*/`放在新行，使它们对齐：

```
/*
* 这是一个
* 多行
* 注释
```

Chapter 2: Comments

Comments are used to indicate something to the person reading the code. Comments are treated like a blank by the compiler and do not change anything in the code's actual meaning. There are two syntaxes used for comments in C, the original `/* */` and the slightly newer `//`. Some documentation systems use specially formatted comments to help produce the documentation for code.

Section 2.1: Commenting using the preprocessor

Large chunks of code can also be "commented out" using the preprocessor directives `#if 0` and `#endif`. This is useful when the code contains multi-line comments that otherwise would not nest.

```
#if 0 /* Starts the "comment", anything from here on is removed by preprocessor */

/* A large amount of code with multi-line comments */
int foo()
{
    /* lots of code */

    ...

    /* ... some comment describing the if statement ... */
    if (someTest) {
        /* some more comments */
        return 1;
    }

    return 0;
}

#endif /* 0 */

/* code from here on is "uncommented" (included in compiled executable)
...
```

Section 2.2: /* */ delimited comments

A comment starts with a forward slash followed immediately by an asterisk`(/*)`, and ends as soon as an asterisk immediately followed by a forward slash`(*)` is encountered. Everything in between these character combinations is a comment and is treated as a blank (basically ignored) by the compiler.

```
/* this is a comment */
```

The comment above is a single line comment. Comments of this`/*` type can span multiple lines, like so:

```
/* this is a
multi-line
comment */
```

Though it is not strictly necessary, a common style convention with multi-line comments is to put leading spaces and asterisks on the lines subsequent to the first, and the`/*` and`*/` on new lines, such that they all line up:

```
/*
* this is a
* multi-line
* comment
```

*/

额外的星号对注释没有任何功能性影响，因为它们都没有相关的正斜杠。

这些 /* 类型的注释可以单独占一行，放在代码行末，甚至嵌入代码行中：

```
/* 这个注释单独占一行 */
if (x && y) { /*这个注释在行尾 */
    if ((complexCondition1) /* 这个注释在代码行内 */
        && (complexCondition2)) {
        /* 这个注释在 if 语句内，单独占一行 */
    }
}
```

注释不能嵌套。这是因为任何后续的 /* 都会被忽略（作为注释内容），而第一个遇到的 */ 会被视为注释的结束。以下示例中的注释 将无法正常工作：

```
/* 外层注释，意味着这里被忽略 => /* 尝试的内层注释 */ <= 结束了注释，不是
这个 => */
```

要注释包含此类注释的代码块（否则会嵌套），请参见下面的使用预处理器注释示例

第2.3节：// 分隔的注释

版本 ≥ C99

C99引入了使用C++风格的单行注释。这种注释以两个斜杠开头，并持续到行尾：

```
// 这是一个注释
```

这种注释不允许多行注释，尽管可以通过连续添加多条单行注释来形成注释块：

```
// 这些行中的每一行都是单行注释
// 注意每一行都必须以
// 两个斜杠开头
```

这种注释可以单独占一行，也可以放在代码行的末尾。然而，因为它们会持续到行尾，所以不能在代码行中间使用

```
// 这条注释单独占一行
if (x && y) { // 这条注释在行尾
    // 这条注释在if语句内，单独占一行
}
```

第2.4节：三字符组合可能导致的陷阱

版本 ≥ C99

在编写//分隔的注释时，可能会出现影响其预期操作的打字错误。如果输入：

*/

The extra asterisks do not have any functional effect on the comment as none of them have a related forward slash.

These /* type of comments can be used on their own line, at the end of a code line, or even within lines of code:

```
/* this comment is on its own line */
if (x && y) { /*this comment is at the end of a line */
    if ((complexCondition1) /* this comment is within a line of code */
        && (complexCondition2)) {
        /* this comment is within an if, on its own line */
    }
}
```

Comments cannot be nested. This is because any subsequent /* will be ignored (as part of the comment) and the first */ reached will be treated as ending the comment. The comment in the following example will not work:

```
/* outer comment, means this is ignored => /* attempted inner comment */ <= ends the comment, not
this one => */
```

To comment blocks of code that contain comments of this type, that would otherwise be nested, see the Commenting using the preprocessor example below

Section 2.3: // delimited comments

Version ≥ C99

C99 introduced the use of C++-style single-line comments. This type of comment starts with two forward slashes and runs to the end of a line:

```
// this is a comment
```

This type of comment does not allow multi-line comments, though it is possible to make a comment block by adding several single line comments one after the other:

```
// each of these lines are a single-line comment
// note how each must start with
// the double forward-slash
```

This type of comment may be used on its own line or at the end of a code line. However, because they run to the end of the line, they may not be used within a code line

```
// this comment is on its own line
if (x && y) { // this comment is at the end of a line
    // this comment is within an if, on its own line
}
```

Section 2.4: Possible pitfall due to trigraphs

Version ≥ C99

While writing // delimited comments, it is possible to make a typographical error that affects their expected operation. If one types:

```
int x = 20; // 我为什么要这么做??/
```

结尾的/是一个打字错误，但现在会被解释为\。这是因为??/形成了一个三字符组合。

三字符组合??/实际上是\的长写形式，\是行连接符号。这意味着编译器认为下一行是当前行的延续，也就是注释的延续，这可能不是预期的。

```
int foo = 20; // 从20开始 ??/
```

```
int bar = 0;
```

```
// 以下代码会导致编译错误 (未声明变量 'bar')  
// 因为 'int bar = 0;' 被视为前一行注释的一部分  
bar += foo;
```

```
int x = 20; // Why did I do this??/
```

The / at the end was a typo but now will get interpreted into \. This is because the ??/ forms a trigraph.

The ??/ trigraph is actually a longhand notation for \, which is the line continuation symbol. This means that the compiler thinks the next line is a continuation of the current line, that is, a continuation of the comment, which may not be what is intended.

```
int foo = 20; // Start at 20 ??/  
int bar = 0;
```

```
// The following will cause a compilation error (undeclared variable 'bar')  
// because 'int bar = 0;' is part of the comment on the preceding line  
bar += foo;
```

第3章：数据类型

第3.1节：解释声明

C语言的一个独特语法特点是，声明的形式反映了被声明对象在普通表达式中的使用方式。

以下一组具有相同优先级和结合性的运算符在声明符中被重复使用，具体包括：

- 一元*“解引用”运算符，表示指针；二元[]“数组下标”运算符，表示数组；(1+n)元()“函数调用”运算符，表示函数；()分组括号，用于覆盖上述运算符的优先级和结合性。

上述三种运算符具有以下优先级和结合性：

运算符	相对优先级	结合性
[] (数组下标)	1	从左到右
() (函数调用)	1	从左到右
* (解引用)	2	从右到左

在解释声明时，必须从标识符向外开始，并按照上述表格中的正确顺序应用相邻的运算符。每次应用运算符都可以替换为以下英文词语：

表达式	解释
thing[X]	一个大小为X的数组，包含...
thing(t1, t2, t3)	一个接受 t1、t2、t3并返回...的函数
*thing	一个指向...的指针

因此，英文解释的开头总是从标识符开始，结尾则是声明左侧的类型。

示例

```
char * names[20];
```

[] 的优先级高于 *，因此解释为：names 是一个大小为20的指向 char 的指针数组。

```
char (* place)[10];
```

使用括号覆盖优先级时，先应用 *：place 是一个指向大小为10的 char 数组的指针。

```
int fn(long, short);
```

这里无需担心优先级：fn 是一个接受 long 和 short 参数并返回 int 的函数。

```
int * fn(void);
```

() 先应用：fn 是一个接受 void 参数并返回指向 int 的指针的函数。

Chapter 3: Data Types

Section 3.1: Interpreting Declarations

A distinctive syntactic peculiarity of C is that declarations mirror the use of the declared object as it would be in a normal expression.

The following set of operators with identical precedence and associativity are reused in declarators, namely:

- the unary * "dereference" operator which denotes a pointer;
- the binary [] "array subscription" operator which denotes an array;
- the (1+n)-ary () "function call" operator which denotes a function;
- the () grouping parentheses which override the precedence and associativity of the rest of the listed operators.

The above three operators have the following precedence and associativity:

Operator	Relative Precedence	Associativity
[] (array subscription)	1	Left-to-right
() (function call)	1	Left-to-right
* (dereference)	2	Right-to-left

When interpreting declarations, one has to start from the identifier outwards and apply the adjacent operators in the correct order as per the above table. Each application of an operator can be substituted with the following English words:

Expression	Interpretation
thing[X]	an array of size X of...
thing(t1, t2, t3)	a function taking t1, t2, t3 and returning...
*thing	a pointer to...

It follows that the beginning of the English interpretation will always start with the identifier and will end with the type that stands on the left-hand side of the declaration.

Examples

```
char *names[20];
```

[] takes precedence over *，so the interpretation is: names is an array of size 20 of a pointer to char.

```
char (*place)[10];
```

In case of using parentheses to override the precedence, the * is applied first: place is a pointer to an array of size 10 of char.

```
int fn(long, short);
```

There is no precedence to worry about here: fn is a function taking long, short and returning int.

```
int *fn(void);
```

The () is applied first: fn is a function taking void and returning a pointer to int.

```
int (* fp)(void);
```

覆盖() 的优先级：fp 是一个指向函数的指针，该函数无参数并返回 int 类型。

```
int arr[5][8];
```

多维数组也不例外；[] 运算符按照表中结合性的从左到右顺序应用：arr 是一个大小为 5 的数组，每个元素是一个大小为 8 的 int 数组。

```
int **ptr;
```

两个解引用运算符优先级相同，因此结合性生效。运算符按从右到左顺序应用：ptr 是一个指向指向 int 的指针。

多重声明

逗号可以用作分隔符（*不是*逗号运算符），用于在单个语句中分隔多个声明。以下语句包含五个声明：

```
int fn(void), *ptr, (*fp)(int), arr[10][20], num;
```

上述示例中声明的对象有：

- fn：一个无参数并返回 int 的函数；
- ptr：一个指向 int 的指针；
- fp：指向一个接受 int 并返回 int 的函数的指针；
- arr：一个大小为 10 的数组，数组元素是大小为 20 的 int 数组；
- num：int。

替代解释

因为声明反映了使用情况，声明也可以从可以应用于对象的操作符以及该表达式最终产生的类型来解释。位于左侧的类型是应用所有操作符后产生的最终结果。

```
/*
 * 对 "arr" 进行下标操作并解引用，结果类型为 "char"。
 * 具体来说：*arr[5] 的类型是 "char"。
 */
char *arr[20];

/*
 * 调用 "fn" 得到的结果类型是 "int"。
 * 具体来说：fn('b') 的类型是 "int"。
 */
int fn(char);

/*
 * 解引用 "fp" 后调用，得到的结果类型是 "int"。
 * 具体来说：(*fp)() 的类型是 "int"。
 */
int (* fp)(void);

/*
 * 对 "strings" 进行两次下标操作并解引用，结果是 "char" 类型。
 * 特别地：*strings[5][15] 的类型是 "char"
 */
char *strings[10][20];
```

```
int (*fp)(void);
```

Overriding the precedence of ()：fp 是一个指向函数的指针，该函数无参数并返回 int。

```
int arr[5][8];
```

Multidimensional arrays are not an exception to the rule; the [] operators are applied in left-to-right order according to the associativity in the table: arr is an array of size 5 of an array of size 8 of int.

```
int **ptr;
```

The two dereference operators have equal precedence, so the associativity takes effect. The operators are applied in right-to-left order: ptr 是一个指向指向 int 的指针。

Multiple Declarations

逗号可以作为分隔符（*not* 逗号运算符）来分隔多个声明。以下语句包含五个声明：

```
int fn(void), *ptr, (*fp)(int), arr[10][20], num;
```

The declared objects in the above example are:

- fn：一个无参数并返回 int 的函数；
- ptr：一个指向 int 的指针；
- fp：指向一个接受 int 并返回 int 的函数的指针；
- arr：一个大小为 10 的数组，数组元素是大小为 20 的 int 数组；
- num：int。

Alternative Interpretation

因为声明反映了使用情况，声明也可以从可以应用于对象的操作符以及该表达式最终产生的类型来解释。位于左侧的类型是应用所有操作符后产生的最终结果。

```
/*
 * Subscripting "arr" and dereferencing it yields a "char" result.
 * Particularly: *arr[5] is of type "char".
 */
char *arr[20];

/*
 * Calling "fn" yields an "int" result.
 * Particularly: fn('b') is of type "int".
 */
int fn(char);

/*
 * Dereferencing "fp" and then calling it yields an "int" result.
 * Particularly: (*fp)() is of type "int".
 */
int (*fp)(void);

/*
 * Subscripting "strings" twice and dereferencing it yields a "char" result.
 * Particularly: *strings[5][15] is of type "char"
 */
char *strings[10][20];
```

第3.2节：固定宽度整数类型（自C99起）

版本 ≥ C99

头文件 `<stdint.h>` 提供了若干固定宽度整数类型定义。这些类型是可选的，且仅在平台具有对应宽度的整数类型，且对应的有符号类型采用二进制补码表示负值时才提供。

有关固定宽度类型的使用提示，请参见备注部分。

```
/* 常用类型包括 */
uint32_t u32 = 32; /* 精确为32位宽 */

uint8_t u8 = 255; /* 精确为8位宽 */

int64_t i64 = -65 /* 精确为64位的二进制补码表示 */
```

第3.3节：整数类型和常量

有符号整数可以是以下类型（`short` 或 `long` 后的 `int` 是可选的）：

```
有符号字符 c = 127; /* 需要为1字节，详见备注。 */
有符号短整型 si = 32767; /* 需要至少16位。 */
signed int i = 32767; /* 要求至少为16位 */
signed long int li = 2147483647; /* 要求至少为32位。 */

版本 ≥ C99
signed long long int li = 2147483647; /* 要求至少为64位 */
```

这些有符号整数类型都有对应的无符号版本。

```
unsigned int i = 65535;
unsigned short = 2767;
unsigned char = 255;
```

对于除`char`以外的所有类型，如果省略了`signed`或`unsigned`部分，则默认是`signed`版本。类型`char`构成第三种字符类型，区别于`signed char`和`unsigned char`，其符号性（是否有符号）取决于平台。

不同类型的整数常量（在C语言术语中称为*literals*）可以用不同的进制和不同的宽度表示，具体取决于它们的前缀或后缀。

```
/* 以下变量初始化为相同的值：*/
int d = 42; /* 十进制常量（基数10） */
int o = 052; /* 八进制常量（基数8） */
int x = 0xaf; /* 十六进制常量（基数16） */
int X = 0XAf; /* （字母'a'到'f'（不区分大小写）表示10到15） */
```

十进制常量总是有符号的。十六进制常量以`0x`或`0X`开头，八进制常量以`0`开头。后两者是否为有符号或无符号，取决于值是否能适应有符号类型。

```
/* 用于描述宽度和有无符号的后缀：*/
长整型 i = 0x32; /* 无后缀表示int或long int */
无符号整型 ui = 65535u; /* u或U表示无符号int或long int */
长整型 li = 65536l; /* l或L表示long int */
```

没有后缀时，常量的类型为第一个能容纳其值的类型，即十进制常量大于

Section 3.2: Fixed Width Integer Types (since C99)

Version ≥ C99

The header `<stdint.h>` provides several fixed-width integer type definitions. These types are *optional* and only provided if the platform has an integer type of the corresponding width, and if the corresponding signed type has a two's complement representation of negative values.

See the remarks section for usage hints of fixed width types.

```
/* commonly used types include */
uint32_t u32 = 32; /* exactly 32-bits wide */

uint8_t u8 = 255; /* exactly 8-bits wide */

int64_t i64 = -65 /* exactly 64 bit in two's complement representation */
```

Section 3.3: Integer types and constants

Signed integers can be of these types (the `int` after `short`, or `long` is optional):

```
signed char c = 127; /* required to be 1 byte, see remarks for further information. */
signed short int si = 32767; /* required to be at least 16 bits. */
signed int i = 32767; /* required to be at least 16 bits */
signed long int li = 2147483647; /* required to be at least 32 bits. */

版本 ≥ C99
signed long long int li = 2147483647; /* required to be at least 64 bits */
```

Each of these signed integer types has an unsigned version.

```
unsigned int i = 65535;
unsigned short = 2767;
unsigned char = 255;
```

For all types but `char` the `signed` version is assumed if the `signed` or `unsigned` part is omitted. The type `char` constitutes a third character type, different from `signed char` and `unsigned char` and the signedness (or not) depends on the platform.

Different types of integer constants (called *literals* in C jargon) can be written in different bases, and different width, based on their prefix or suffix.

```
/* the following variables are initialized to the same value: */
int d = 42; /* decimal constant (base10) */
int o = 052; /* octal constant (base8) */
int x = 0xaf; /* hexadecimal constants (base16) */
int X = 0XAf; /* (letters 'a' through 'f' (case insensitive) represent 10 through 15) */
```

Decimal constants are always `signed`. Hexadecimal constants start with `0x` or `0X` and octal constants start just with a `0`. The latter two are `signed` or `unsigned` depending on whether the value fits into the signed type or not.

```
/* suffixes to describe width and signedness : */
long int i = 0x32; /* no suffix represent int, or long int */
unsigned int ui = 65535u; /* u or U represent unsigned int, or long int */
long int li = 65536l; /* l or L represent long int */
```

Without a suffix the constant has the first type that fits its value, that is a decimal constant that is larger than

INT_MAX 的类型如果可能为long，否则为long long。

头文件`<limits.h>`描述了整数的限制。其实现定义的值应当在大小（绝对值）上等于或大于下列所示值，且符号相同。

宏	类型	值
CHAR_BIT	非位域的最小对象（字节）	8
SCHAR_MIN	有符号字符	-127 / -(27 - 1)
SCHAR_MAX	有符号字符	+127 / 27 - 1
UCHAR_MAX	无符号字符	255 / 28 - 1
CHAR_MIN	字符	见下文
CHAR_MAX	字符	见下文
SHRT_MIN	短整型	-32767 / -(215 - 1)
SHRT_MAX	短整型	+32767 / 215 - 1
USHRT_MAX	无符号短整型	65535 / 216 - 1
INT_MIN	int	-32767 / -(215 - 1)
INT_MAX	int	+32767 / 215 - 1
UINT_MAX	无符号整型	65535 / 216 - 1
LONG_MIN	长整型	-2147483647 / -(231 - 1)
LONG_MAX	长整型	+2147483647 / 231 - 1
ULONG_MAX	无符号长整型	4294967295 / 232 - 1

版本 ≥ C99

宏	类型	值
LLONG_MIN	长长整型	-9223372036854775807 / -(263 - 1)
LLONG_MAX	长长整型	+9223372036854775807 / 263 - 1
ULLONG_MAX	无符号长长整型	18446744073709551615 / 2^{64} - 1

如果char类型对象的值在表达式中进行符号扩展，则CHAR_MIN的值应与SCHAR_MIN相同，CHAR_MAX的值应与SCHAR_MAX相同。如果char类型对象的值在表达式中不进行符号扩展，则CHAR_MIN的值应为0，CHAR_MAX的值应与UCHAR_MAX相同。

版本 ≥ C99

C99标准新增了一个头文件，`<stdint.h>`，其中包含固定宽度整数的定义。有关更深入的解释，请参见固定宽度整数示例。

第3.4节：浮点常量

C语言有三种必备的实数浮点类型，分别是float、double和long double。

```
float f = 0.314f; /* 后缀 f 或 F 表示 float 类型 */
double d = 0.314; /* 无后缀表示 double 类型 */
long double ld = 0.314l; /* 后缀 l 或 L 表示 long double 类型 */

/* 浮点数定义的各部分都是可选的 */
double x = 1.; /* 有效，分数部分可选 */
double y = .1; /* 有效，整数部分可选 */

/* 也可以用科学计数法定义 */
double sd = 1.2e3; /* 小数 1.2 乘以 10^3, 即 1200.0 */
```

INT_MAX is of type long if possible, or long long otherwise.

The header file `<limits.h>` describes the limits of integers as follows. Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown below, with the same sign.

Macro	Type	Value
CHAR_BIT	smallest object that is not a bit-field (byte)	8
SCHAR_MIN	signed char	-127 / -(27 - 1)
SCHAR_MAX	signed char	+127 / 27 - 1
UCHAR_MAX	unsigned char	255 / 28 - 1
CHAR_MIN	char	see below
CHAR_MAX	char	see below
SHRT_MIN	short int	-32767 / -(215 - 1)
SHRT_MAX	short int	+32767 / 215 - 1
USHRT_MAX	unsigned short int	65535 / 216 - 1
INT_MIN	int	-32767 / -(215 - 1)
INT_MAX	int	+32767 / 215 - 1
UINT_MAX	unsigned int	65535 / 216 - 1
LONG_MIN	long int	-2147483647 / -(231 - 1)
LONG_MAX	long int	+2147483647 / 231 - 1
ULONG_MAX	unsigned long int	4294967295 / 232 - 1

Version ≥ C99

Macro	Type	Value
LLONG_MIN	long long int	-9223372036854775807 / -(263 - 1)
LLONG_MAX	long long int	+9223372036854775807 / 263 - 1
ULLONG_MAX	unsigned long long int	18446744073709551615 / 264 - 1

If the value of an object of type `char` sign-extends when used in an expression, the value of CHAR_MIN shall be the same as that of SCHAR_MIN and the value of CHAR_MAX shall be the same as that of SCHAR_MAX. If the value of an object of type `char` does not sign-extend when used in an expression, the value of CHAR_MIN shall be 0 and the value of CHAR_MAX shall be the same as that of UCHAR_MAX.

Version ≥ C99

The C99 standard added a new header, `<stdint.h>`, which contains definitions for fixed width integers. See the fixed width integer example for a more in-depth explanation.

Section 3.4: Floating Point Constants

The C language has three mandatory real floating point types, `float`, `double`, and `long double`.

```
float f = 0.314f; /* suffix f or F denotes type float */
double d = 0.314; /* no suffix denotes double */
long double ld = 0.314l; /* suffix l or L denotes long double */

/* the different parts of a floating point definition are optional */
double x = 1.; /* valid, fractional part is optional */
double y = .1; /* valid, whole-number part is optional */

/* they can also be defined in scientific notation */
double sd = 1.2e3; /* decimal fraction 1.2 is scaled by 10^3, that is 1200.0 */
```

头文件 `<float.h>` 定义了浮点运算的各种限制。

浮点运算的实现是由具体平台决定的。然而，大多数现代平台（ARM、x86、x86_64、MIPS）使用 [IEEE 754](#) 浮点运算。

C 语言还有三种可选的复数浮点类型，它们是从上述类型派生的。

第 3.5 节：字符串字面量

C 语言中的字符串字面量是一串字符，以字面量零结尾。

```
char* str = "hello, world"; /* 字符串字面量 */  
  
/* 字符串字面量可用于初始化数组 */  
char a1[] = "abc"; /* a1 是 char[4], 包含 {'a','b','c','\0'} */  
char a2[4] = "abc"; /* 与 a1 相同 */  
char a3[3] = "abc"; /* a1 是一个包含 {'a','b','c'} 的 char[3], 缺少 '\0' */
```

字符串字面量是不可修改的（实际上可能被放置在只读内存中，如 `.rodata`）。尝试修改它们的值会导致未定义行为。

```
char* s = "foobar";  
s[0] = 'F'; /* 未定义行为 */  
  
/* 使用 `const` 来标注字符串字面量是个好习惯 */  
char const* s1 = "foobar";  
s1[0] = 'F'; /* 编译错误！ */
```

多个字符串字面量会在编译时连接，这意味着你可以写出如下构造。

版本 < C99

```
/* 只能连接两个窄字符或两个宽字符字符串字面量 */  
char* s = "Hello, " "World";
```

版本 ≥ C99

```
/* 自 C99 起，可以连接超过两个 */  
/* 连接方式由实现定义 */  
char* s1 = "Hello" ", "World";  
  
/* 常见用法是格式字符串的连接 */  
char* fmt = "%" PRIId16; /* 自C99起的PRIId16宏 */
```

字符串字面量，与字符常量相同，支持不同的字符集。

```
/* 普通字符串字面量，类型为char[] */  
char* s1 = "abc";  
  
/* 宽字符串字面量，类型为wchar_t[] */  
wchar_t* s2 = L"abc";  
  
/* UTF-8字符串字面量，类型为char[] */  
char* s3 = u8"abc";  
  
/* 16位宽字符串字面量，类型为 char16_t[] */  
char16_t* s4 = u"abc";  
  
/* 32位宽字符串字面量，类型为 char32_t[] */  
char32_t* s5 = U"abc";
```

The header `<float.h>` defines various limits for floating point operations.

Floating point arithmetic is implementation defined. However, most modern platforms (arm, x86, x86_64, MIPS) use [IEEE 754](#) floating point operations.

C also has three optional complex floating point types that are derived from the above.

Section 3.5: String Literals

A string literal in C is a sequence of chars, terminated by a literal zero.

```
char* str = "hello, world"; /* string literal */  
  
/* string literals can be used to initialize arrays */  
char a1[] = "abc"; /* a1 is char[4] holding {'a','b','c','\0'} */  
char a2[4] = "abc"; /* same as a1 */  
char a3[3] = "abc"; /* a1 is char[3] holding {'a','b','c'}, missing the '\0' */
```

String literals are **not modifiable** (and in fact may be placed in read-only memory such as `.rodata`). Attempting to alter their values results in undefined behaviour.

```
char* s = "foobar";  
s[0] = 'F'; /* undefined behaviour */  
  
/* it's good practice to denote string literals as such, by using `const` */  
char const* s1 = "foobar";  
s1[0] = 'F'; /* compiler error! */
```

Multiple string literals are concatenated at compile time, which means you can write construct like these.

Version < C99

```
/* only two narrow or two wide string literals may be concatenated */  
char* s = "Hello, " "World";
```

Version ≥ C99

```
/* since C99, more than two can be concatenated */  
/* concatenation is implementation defined */  
char* s1 = "Hello" ", " "World";  
  
/* common usages are concatenations of format strings */  
char* fmt = "%" PRIId16; /* PRIId16 macro since C99 */
```

String literals, same as character constants, support different character sets.

```
/* normal string literal, of type char[] */  
char* s1 = "abc";  
  
/* wide character string literal, of type wchar_t[] */  
wchar_t* s2 = L"abc";  
  
/* UTF-8 string literal, of type char[] */  
char* s3 = u8"abc";  
  
/* 16-bit wide string literal, of type char16_t[] */  
char16_t* s4 = u"abc";  
  
/* 32-bit wide string literal, of type char32_t[] */  
char32_t* s5 = U"abc";
```

第4章：运算符

编程语言中的运算符是一个符号，告诉编译器或解释器执行特定的数学、关系或逻辑运算并产生最终结果。

C语言有许多强大的运算符。许多C运算符是二元运算符，意味着它们有两个操作数。例如，在 `a / b` 中，`/` 是一个二元运算符，接受两个操作数 (`a, b`)。也有一些一元运算符，它们只接受一个操作数（例如：`~, ++`），还有唯一的三元运算符 `? :`。

第4.1节：关系运算符

关系运算符检查两个操作数之间的特定关系是否为真。结果被计算为1（表示真）或0（表示假）。这个结果通常用于影响控制流（通过 `if, while, for`），但也可以存储在变量中。

等于 "=="

检查所提供的操作数是否相等。

```
1 == 0;      /* 计算结果为0。 */
1 == 1;      /* 计算结果为1。 */

int x = 5;
int y = 5;
int *xptr = &x, *yprt = &y;
xptr == yptr; /* 计算结果为0，操作数持有不同的地址。 */
*xptr == *yprt; /* 计算结果为1，操作数指向的地址存储相同的值。 */
```

注意：此运算符不应与赋值运算符 (=) 混淆！

不等于 "!="

检查所提供的操作数是否不相等。

```
1 != 0;      /* 计算结果为1。 */
1 != 1;      /* 计算结果为0。 */

int x = 5;
int y = 5;
int *xptr = &x, *yprt = &y;
xptr != yptr; /* 结果为1，操作数指向不同的地址。 */
*xptr != *yprt; /* 结果为0，操作数指向的地址中存储的值相同。 */
```

该运算符的效果与等于 (==) 运算符的结果相反。

非 "!"

检查一个对象是否等于0。

`!` 也可以直接用于变量，示例如下：

```
!someVal
```

这与以下表达式效果相同：

Chapter 4: Operators

An operator in a programming language is a symbol that tells the compiler or interpreter to perform a specific mathematical, relational or logical operation and produce a final result.

C has many powerful operators. Many C operators are binary operators, which means they have two operands. For example, in `a / b`, `/` is a binary operator that accepts two operands (`a, b`). There are some unary operators which take one operand (for example: `~, ++`), and only one ternary operator `? :`.

Section 4.1: Relational Operators

Relational operators check if a specific relation between two operands is true. The result is evaluated to 1 (which means *true*) or 0 (which means *false*). This result is often used to affect control flow (via `if, while, for`), but can also be stored in variables.

Equals "=="

Checks whether the supplied operands are equal.

```
1 == 0;          /* evaluates to 0. */
1 == 1;          /* evaluates to 1. */

int x = 5;
int y = 5;
int *xptr = &x, *yprt = &y;
xptr == yptr;   /* evaluates to 0, the operands hold different location addresses. */
*xptr == *yprt; /* evaluates to 1, the operands point at locations that hold the same value. */
```

Attention: This operator should not be confused with the assignment operator (=)!

Not equals "!="

Checks whether the supplied operands are not equal.

```
1 != 0;          /* evaluates to 1. */
1 != 1;          /* evaluates to 0. */

int x = 5;
int y = 5;
int *xptr = &x, *yprt = &y;
xptr != yptr;   /* evaluates to 1, the operands hold different location addresses. */
*xptr != *yprt; /* evaluates to 0, the operands point at locations that hold the same value. */
```

This operator effectively returns the opposite result to that of the equals (==) operator.

Not "!"

Check whether an object is equal to 0.

The `!` can also be used directly with a variable as follows:

```
!someVal
```

This has the same effect as:

```
someVal == 0
```

大于 ">"

检查左操作数的值是否大于右操作数的值

```
5 > 4      /* 计算结果为 1。 */
4 > 5      /* 计算结果为 0。 */
4 > 4      /* 计算结果为 0. */
```

小于 "<"

检查左操作数是否小于右操作数

```
5 < 4      /* 计算结果为 0. */
4 < 5      /* 计算结果为 1. */
4 < 4      /* 计算结果为 0. */
```

大于或等于 ">="

检查左操作数是否大于或等于右操作数。

```
5 >= 4     /* 计算结果为 1。 */
4 >= 5     /* 计算结果为 0. */
4 >= 4     /* 计算结果为 1. */
```

小于或等于 "<="

检查左操作数是否小于或等于右操作数的值。

```
5 <= 4     /* 计算结果为 0. */
4 <= 5     /* 计算结果为 1. */
4 <= 4     /* 计算结果为 1. */
```

第4.2节：条件运算符/三元运算符

计算其第一个操作数，如果结果值不等于零，则计算第二个操作数。否则，计算第三个操作数，如下例所示：

```
a = b ? c : d;
```

等价于：

```
如果 (b)
    a = c;
否则
    a = d;
```

这个伪代码表示为：条件 ? 真值 : 假值。每个值都可以是一个已计算表达式的结果。

```
int x = 5;
int y = 42;
printf("%i, %i\n", 1 ? x : y, 0 ? x : y); /* 输出 "5, 42" */
```

```
someVal == 0
```

Greater than ">"

Checks whether the left hand operand has a greater value than the right hand operand

```
5 > 4      /* evaluates to 1. */
4 > 5      /* evaluates to 0. */
4 > 4      /* evaluates to 0. */
```

Less than "<"

Checks whether the left hand operand has a smaller value than the right hand operand

```
5 < 4      /* evaluates to 0. */
4 < 5      /* evaluates to 1. */
4 < 4      /* evaluates to 0. */
```

Greater than or equal ">="

Checks whether the left hand operand has a greater or equal value to the right operand.

```
5 >= 4     /* evaluates to 1. */
4 >= 5     /* evaluates to 0. */
4 >= 4     /* evaluates to 1. */
```

Less than or equal "<="

Checks whether the left hand operand has a smaller or equal value to the right operand.

```
5 <= 4     /* evaluates to 0. */
4 <= 5     /* evaluates to 1. */
4 <= 4     /* evaluates to 1. */
```

Section 4.2: Conditional Operator/Ternary Operator

Evaluates its first operand, and, if the resulting value is not equal to zero, evaluates its second operand. Otherwise, it evaluates its third operand, as shown in the following example:

```
a = b ? c : d;
```

is equivalent to:

```
if (b)
    a = c;
else
    a = d;
```

This pseudo-code represents it: condition ? value_if_true : value_if_false. Each value can be the result of an evaluated expression.

```
int x = 5;
int y = 42;
printf("%i, %i\n", 1 ? x : y, 0 ? x : y); /* Outputs "5, 42" */
```

条件运算符可以嵌套。例如，以下代码确定三个数中较大的一个：

```
big= a > b ? (a > c ? a : c)
: (b > c ? b : c);
```

以下示例将偶数写入一个文件，将奇数写入另一个文件：

```
#include<stdio.h>

int main()
{
FILE *偶数, *奇数;
int n = 10;
size_t k = 0;

偶数 = fopen("even.txt", "w");
奇数 = fopen("odds.txt", "w");

for(k = 1; k < n + 1; k++)
{
k%2==0 ? fprintf(偶数, "%5d", k): fprintf(奇数, "%5d",
k);}

fclose(偶数);
fclose(奇数);

return 0;
}
```

条件运算符的结合方向是从右到左。考虑以下表达式：

```
exp1 ? exp2 : exp3 ? exp4 : exp5
```

由于结合性是从右到左，上述表达式被计算为

```
exp1 ? exp2 : (exp3 ? exp4 : exp5 )
```

第4.3节：按位运算符

按位运算符可用于对变量执行位级操作。

下面是C语言支持的所有六种按位运算符列表：

符号	运算符
&	按位与
	按位或（包含）
^	按位异或（XOR）
~	按位取反（补码）
<<	逻辑左移
>>	逻辑右移

以下程序演示了所有按位运算符的使用：

```
#include <stdio.h>

int main(void)
{
```

The conditional operator can be nested. For example, the following code determines the bigger of three numbers:

```
big= a > b ? (a > c ? a : c)
: (b > c ? b : c);
```

The following example writes even integers to one file and odd integers to another file:

```
#include<stdio.h>

int main()
{
FILE *even, *odds;
int n = 10;
size_t k = 0;

even = fopen("even.txt", "w");
odds = fopen("odds.txt", "w");

for(k = 1; k < n + 1; k++)
{
k%2==0 ? fprintf(even, "\t%5d\n", k)
: fprintf(odds, "\t%5d\n", k);

}

fclose(even);
fclose(odds);

return 0;
}
```

The conditional operator associates from right to left. Consider the following:

```
exp1 ? exp2 : exp3 ? exp4 : exp5
```

As the association is from right to left, the above expression is evaluated as

```
exp1 ? exp2 : (exp3 ? exp4 : exp5 )
```

Section 4.3: Bitwise Operators

Bitwise operators can be used to perform bit level operation on variables.

Below is a list of all six bitwise operators supported in C:

Symbol	Operator
&	bitwise AND
	bitwise inclusive OR
^	bitwise exclusive OR (XOR)
~	bitwise not (one's complement)
<<	logical left shift
>>	logical right shift

Following program illustrates the use of all bitwise operators:

```
#include <stdio.h>

int main(void)
{
```

```

unsigned int a = 29; /* 29 = 0001 1101 */
unsigned int b = 48; /* 48 = 0011 0000 */
int c = 0;

c = a & b; /* 32 = 0001 0000 */printf("%d
& %d = %d", a, b, c );

c = a | b; /* 61 = 0011 1101 */printf("%d
| %d = %d", a, b, c );

c = a ^ b; /* 45 = 0010 1101 */printf("%d
^ %d = %d", a, b, c );

c = ~a; /* -30 = 1110 0010 */printf("~%d
= %d", a, c );

c = a << 2; /* 116 = 0111 0100 */printf("%d
<< 2 = %d", a, c );

c = a >> 2; /* 7 = 0000 0111 */printf("%d
>> 2 = %d", a, c );

return 0;
}

```

应避免对有符号类型进行按位操作，因为此类位表示的符号位具有特殊含义。移位运算符有特定的限制：

- 将1位左移到符号位是错误的，会导致未定义行为。
- 对负值（符号位为1）进行右移是实现定义的，因此不可移植。
- 如果移位运算符右操作数的值为负数，或大于等于提升后的左操作数的位宽，则行为未定义。

掩蔽：

掩码指的是通过使用逻辑按位运算，从变量中提取所需的位（或转换所需的位）的过程。用于执行掩码操作的操作数（常量或变量）称为掩码（mask）。

掩码有多种不同的用途：

- 决定整数变量的位模式。
- 将给定位模式的一部分复制到新变量中，而新变量的其余部分用0填充（使用按位与运算）。
- 将给定位模式的一部分复制到新变量中，而新变量的其余部分用1填充（使用按位或运算）。
- 将给定位模式的一部分复制到新变量中，而原始位模式的其余部分在新变量中取反（使用按位异或运算）。

下面的函数使用掩码来显示变量的位模式：

```

#include <limits.h>
void bit_pattern(int u)
{
    int i, x, word;
    unsigned mask = 1;

```

```

unsigned int a = 29; /* 29 = 0001 1101 */
unsigned int b = 48; /* 48 = 0011 0000 */
int c = 0;

c = a & b; /* 32 = 0001 0000 */
printf("%d & %d = %d\n", a, b, c );

c = a | b; /* 61 = 0011 1101 */
printf("%d | %d = %d\n", a, b, c );

c = a ^ b; /* 45 = 0010 1101 */
printf("%d ^ %d = %d\n", a, b, c );

c = ~a; /* -30 = 1110 0010 */
printf("~%d = %d\n", a, c );

c = a << 2; /* 116 = 0111 0100 */
printf("%d << 2 = %d\n", a, c );

c = a >> 2; /* 7 = 0000 0111 */
printf("%d >> 2 = %d\n", a, c );

return 0;
}

```

Bitwise operations with signed types should be avoided because the sign bit of such a bit representation has a particular meaning. Particular restrictions apply to the shift operators:

- Left shifting a 1 bit into the signed bit is erroneous and leads to undefined behavior.
- Right shifting a negative value (with sign bit 1) is implementation defined and therefore not portable.
- If the value of the right operand of a shift operator is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

Masking:

Masking refers to the process of extracting the desired bits from (or transforming the desired bits in) a variable by using logical bitwise operations. The operand (a constant or variable) that is used to perform masking is called a *mask*.

Masking is used in many different ways:

- To decide the bit pattern of an integer variable.
- To copy a portion of a given bit pattern to a new variable, while the remainder of the new variable is filled with 0s (using bitwise AND)
- To copy a portion of a given bit pattern to a new variable, while the remainder of the new variable is filled with 1s (using bitwise OR).
- To copy a portion of a given bit pattern to a new variable, while the remainder of the original bit pattern is inverted within the new variable (using bitwise exclusive OR).

The following function uses a mask to display the bit pattern of a variable:

```

#include <limits.h>
void bit_pattern(int u)
{
    int i, x, word;
    unsigned mask = 1;

```

```

word = CHAR_BIT * sizeof(int);
mask = mask << (word - 1); /* 向左移动1位到最左边的位置 */
for(i = 1; i <= word; i++)
{
    x = (u & mask) ? 1 : 0; /* 识别该位 */
    printf("%d", x); /* 打印位值 */
    mask >>= 1; /* 将mask右移1位 */
}
}

```

第4.4节：逻辑运算符的短路行为

短路是一种功能，当可能时跳过对（if/while/...）条件部分的求值。在对两个操作数进行逻辑运算时，先对第一个操作数求值（为真或假），如果已有结论（即使用`&&`时第一个操作数为假，使用`||`时第一个操作数为真），则不再对第二个操作数求值。

示例：

```

#include <stdio.h>

int main(void) {
    int a = 20;
    int b = -5;

    /* 这里 'b == -5' 不会被求值,
       因为 'a != 20' 为假。 */
    if (a != 20 && b == -5) {printf
        ("我不会被打印！");}

    return 0;
}

```

自己试试看：

```

#include <stdio.h>int

print(int i) {printf("打
印函数 %d", i);return i;
}

int
int a = 20;

/* 这里没有调用 'print(a)',
   因为表达式 'a != 20' 为假。 */
if (a != 20 && print(a)) {printf(
    "我不会被打印！);}

/* 这里调用了 'print(a)',
   因为表达式 'a == 20' 为真。 */
if (a == 20 && print(a)) {printf(
    "我会被打印！);}

return 0;
}

```

```

word = CHAR_BIT * sizeof(int);
mask = mask << (word - 1); /* shift 1 to the leftmost position */
for(i = 1; i <= word; i++)
{
    x = (u & mask) ? 1 : 0; /* identify the bit */
    printf("%d", x); /* print bit value */
    mask >>= 1; /* shift mask to the right by 1 bit */
}
}

```

Section 4.4: Short circuit behavior of logical operators

Short circuiting is a functionality that skips evaluating parts of a (if/while/...) condition when able. In case of a logical operation on two operands, the first operand is evaluated (to true or false) and if there is a verdict (i.e first operand is false when using `&&`, first operand is true when using `||`) the second operand is not evaluated.

Example:

```

#include <stdio.h>

int main(void) {
    int a = 20;
    int b = -5;

    /* here 'b == -5' is not evaluated,
       since a 'a != 20' is false. */
    if (a != 20 && b == -5) {
        printf("I won't be printed!\n");
    }

    return 0;
}

```

Check it out yourself:

```

#include <stdio.h>

int print(int i) {
    printf("print function %d\n", i);
    return i;
}

int main(void) {
    int a = 20;

    /* here 'print(a)' is not called,
       since a 'a != 20' is false. */
    if (a != 20 && print(a)) {
        printf("I won't be printed!\n");
    }

    /* here 'print(a)' is called,
       since a 'a == 20' is true. */
    if (a == 20 && print(a)) {
        printf("I will be printed!\n");
    }

    return 0;
}

```

输出：

```
$ ./a.out
print function 20
我会被打印!
```

短路求值很重要，当你想避免计算代价高的表达式时。此外，它还会像本例中这样严重影响程序的流程：[为什么这个程序打印了4次“forked!”？](#)

第4.5节：逗号运算符

先计算左操作数，丢弃其结果值，然后计算右操作数，结果为最右操作数的值。

```
int x = 42, y = 42;
printf("%i", (x *= 2, y)); /* 输出 "42". */
```

逗号运算符在其操作数之间引入了一个序列点。

注意，函数调用中用于分隔参数的逗号不是逗号运算符，而是称为分隔符，与逗号运算符不同。因此，它不具备逗号运算符的属性。

上述printf()调用同时包含了逗号运算符和分隔符。

```
printf("%i", (x *= 2, y)); /* 输出 "42". */
^          ^          ^
    这是逗号运算符      这是分隔符  */
```

逗号运算符常用于for循环的初始化部分以及更新部分。例如：

```
for(k = 1; k < 10; printf("\%d\n", k), k += 2); /* 输出小于9的奇数 */
/* 输出前9个自然数的和 */
for(sumk = 1, k = 1; k < 10; k++, sumk += k)
    printf("\%5d\%5d\n", k, sumk);
```

第4.6节：算术运算符

基础算术

返回一个值，该值是将左操作数应用于右操作数后，使用相关的数学运算得到的结果。遵循正常的数学交换规则（即加法和乘法是交换律的，减法、除法和取模运算则不是）。

加法运算符

加法运算符（+）用于将两个操作数相加。示例：

```
#include <stdio.h>

int main(void)
{
    int a = 5;
```

Output:

```
$ ./a.out
print function 20
I will be printed!
```

Short circuiting is important, when you want to avoid evaluating terms that are (computationally) costly. Moreover, it can heavily affect the flow of your program like in this case: [Why does this program print "forked!" 4 times?](#)

Section 4.5: Comma Operator

Evaluates its left operand, discards the resulting value, and then evaluates its rightmost operand.

```
int x = 42, y = 42;
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
```

The comma operator introduces a sequence point between its operands.

Note that the *comma* used in functions calls that separate arguments is NOT the *comma operator*, rather it's called a *separator* which is different from the *comma operator*. Hence, it doesn't have the properties of the *comma operator*.

The above printf() call contains both the *comma operator* and the *separator*.

```
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
/*           ^           ^
    this is a comma operator  */
/*           this is a separator  */
```

The comma operator is often used in the initialization section as well as in the updating section of a `for` loop. For example:

```
for(k = 1; k < 10; printf("\%d\n", k), k += 2); /*outputs the odd numbers below 9*/
/* outputs sum to first 9 natural numbers */
for(sumk = 1, k = 1; k < 10; k++, sumk += k)
    printf("\%5d\%5d\n", k, sumk);
```

Section 4.6: Arithmetic Operators

Basic Arithmetic

Return a value that is the result of applying the left hand operand to the right hand operand, using the associated mathematical operation. Normal mathematical rules of commutation apply (i.e. addition and multiplication are commutative, subtraction, division and modulus are not).

Addition Operator

The addition operator (+) is used to add two operands together. Example:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;
```

```

int c = a + b; /* c 现在的值是 12 */

printf("%d + %d = %d", a, b, c); /* 将输出 "5 + 7 = 12" */

return 0;
}

```

减法运算符

减法运算符 (-) 用于从第一个操作数中减去第二个操作数。示例：

```

#include <stdio.h>

int main(void)
{
    int a = 10;
    int b = 7;

    int c = a - b; /* c 现在保存的值是 3 */

    printf("%d - %d = %d", a, b, c); /* 将输出 "10 - 7 = 3" */

    return 0;
}

```

乘法运算符

乘法运算符 (*) 用于对两个操作数进行相乘。示例：

```

#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a * b; /* c 现在保存的值是 35 */

    printf("%d * %d = %d", a, b, c); /* 将输出 "5 * 7 = 35" */

    return 0;
}

```

不要与 * 解引用运算符混淆。

除法运算符

除法运算符 (/) 用于将第一个操作数除以第二个操作数。如果两个操作数都是整数，它将返回一个整数值并舍弃余数（计算和获取余数请使用取模运算符%）。

如果其中一个操作数是浮点数，结果将是分数的近似值。

示例：

```

#include <stdio.h>

int main (void)

```

```

int c = a + b; /* c now holds the value 12 */

printf("%d + %d = %d", a, b, c); /* will output "5 + 7 = 12" */

return 0;
}

```

Subtraction Operator

The subtraction operator (-) is used to subtract the second operand from the first. Example:

```

#include <stdio.h>

int main(void)
{
    int a = 10;
    int b = 7;

    int c = a - b; /* c now holds the value 3 */

    printf("%d - %d = %d", a, b, c); /* will output "10 - 7 = 3" */

    return 0;
}

```

Multiplication Operator

The multiplication operator (*) is used to multiply both operands. Example:

```

#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a * b; /* c now holds the value 35 */

    printf("%d * %d = %d", a, b, c); /* will output "5 * 7 = 35" */

    return 0;
}

```

Not to be confused with the * dereference operator.

Division Operator

The division operator (/) divides the first operand by the second. If both operands of the division are integers, it will return an integer value and discard the remainder (use the modulo operator % for calculating and acquiring the remainder).

If one of the operands is a floating point value, the result is an approximation of the fraction.

Example:

```

#include <stdio.h>

int main (void)

```

```

{
    int a = 19 / 2; /* a 的值为 9 */
    int b = 18 / 2; /* b 的值为 9 */
    int c = 255 / 2; /* c 的值为 127 */
    int d = 44 / 4; /* d 的值为 11 */
    double e = 19 / 2.0; /* e 的值为 9.5 */
    double f = 18.0 / 2; /* f 的值为 9.0 */
    double g = 255 / 2.0; /* g 的值为 127.5 */
    double h = 45.0 / 4; /* h 的值为 11.25 */

    printf("19 / 2 = %d", a); /* 将输出 "19 / 2 = 9" */
    , b); /* 将输出 "18 / 2 = 9" */
    printf("255 / 2 = %d", c); /* 将输出 "255 / 2 = 127" */
    printf("44 / 4 = %d", d); /* 将输出 "44 / 4 = 11" */
    printf("19 / 2.0 = %g", e); /* 将输出 "19 / 2.0 = 9.5" */
    printf("18.0 / 2 = %g", f); /* 将输出 "18.0 / 2 = 9" */
    printf("255 / 2.0 = %g", g); /* 将输出 "255 / 2.0 = 127.5" */
    printf("45.0 / 4 = %g", h); /* 将输出 "45.0 / 4 = 11.25" */
    return 0;
}

```

取模运算符

取模运算符 (%) 仅接受整数操作数，用于计算第一个操作数除以第二个操作数后的余数。示例：

```

#include <stdio.h>

int main (void) {
    int a = 25 % 2; /* a 的值为 1 */
    int b = 24 % 2; /* b 的值为 0 */
    int c = 155 % 5; /* c 的值为 0 */
    int d = 49 % 25; /* d 的值为 24 */

    printf("25 % 2 = %d", a); /* 将输出 "25 % 2 = 1" */
    , b); /* 将输出 "24 % 2 = 0" */
    printf("155 % 5 = %d", c); /* 将输出 "155 % 5 = 0" */
    printf("49 % 25 = %d", d); /* 将输出 "49 % 25 = 24" */
    /* */return 0;
}

```

自增 / 自减运算符

自增 (a++) 和自减 (

a--

) 运算符的不同之处在于它们在没有赋值运算符的情况下改变你所应用变量的值。你可以在变量的前面或后面使用递增和递减运算符。运算符的位置改变了值递增/递减的时机，是在赋值给变量之前还是之后。示例：

```

#include <stdio.h>

int main(void)
{
    int a = 1;
    int b = 4;

```

```

{
    int a = 19 / 2; /* a holds value 9 */
    int b = 18 / 2; /* b holds value 9 */
    int c = 255 / 2; /* c holds value 127 */
    int d = 44 / 4; /* d holds value 11 */
    double e = 19 / 2.0; /* e holds value 9.5 */
    double f = 18.0 / 2; /* f holds value 9.0 */
    double g = 255 / 2.0; /* g holds value 127.5 */
    double h = 45.0 / 4; /* h holds value 11.25 */

    printf("19 / 2 = %d\n", a); /* Will output "19 / 2 = 9" */
    printf("18 / 2 = %d\n", b); /* Will output "18 / 2 = 9" */
    printf("255 / 2 = %d\n", c); /* Will output "255 / 2 = 127" */
    printf("44 / 4 = %d\n", d); /* Will output "44 / 4 = 11" */
    printf("19 / 2.0 = %g\n", e); /* Will output "19 / 2.0 = 9.5" */
    printf("18.0 / 2 = %g\n", f); /* Will output "18.0 / 2 = 9" */
    printf("255 / 2.0 = %g\n", g); /* Will output "255 / 2.0 = 127.5" */
    printf("45.0 / 4 = %g\n", h); /* Will output "45.0 / 4 = 11.25" */

    return 0;
}

```

Modulo Operator

The modulo operator (%) receives integer operands only, and is used to calculate the remainder after the first operand is divided by the second. Example:

```

#include <stdio.h>

int main (void) {
    int a = 25 % 2; /* a holds value 1 */
    int b = 24 % 2; /* b holds value 0 */
    int c = 155 % 5; /* c holds value 0 */
    int d = 49 % 25; /* d holds value 24 */

    printf("25 % 2 = %d\n", a); /* Will output "25 % 2 = 1" */
    printf("24 % 2 = %d\n", b); /* Will output "24 % 2 = 0" */
    printf("155 % 5 = %d\n", c); /* Will output "155 % 5 = 0" */
    printf("49 % 25 = %d\n", d); /* Will output "49 % 25 = 24" */

    return 0;
}

```

Increment / Decrement Operators

The increment (a++) and decrement (

a--

) operators are different in that they change the value of the variable you apply them to without an assignment operator. You can use increment and decrement operators either before or after the variable. The placement of the operator changes the timing of the incrementation/decrementation of the value to before or after assigning it to the variable. Example:

```

#include <stdio.h>

int main(void)
{
    int a = 1;
    int b = 4;

```

```

int c = 1;
int d = 4;

a++;
printf("a = %d",a); /* 将输出 "a = 2" */
b--;

printf("b = %d",b); /* 将输出 "b = 3" */

if (++c > 1) { /* c 在条件比较前先加 1 */printf("这将被打印"); /* 这将被打印 */
} else {
    printf("这永远不会被打印"); /* 这不会被打印 */
}

if (d-- < 4) { /* d 在比较后递减 */printf("这永远不会被打印"); /* 这
    不会被打印 */} else {
    printf("这将打印"); /* 这是打印的内容 */
}

```

正如c和d的例子所示，这两个运算符都有两种形式，分别是前缀表示法和后缀表示法。两者在对变量进行递增（++）或递减（--）时效果相同，但返回的值不同：
前缀操作先执行操作然后返回值，而后缀操作先确定要返回的值，然后再执行操作。

由于这种可能令人困惑的行为，在表达式中使用递增/递减运算符存在争议。

第4.7节：访问运算符

成员访问运算符（点号.和箭头->）用于访问struct的成员。

对象的成员

求值为表示被访问对象成员的左值。

```

struct MyStruct
{
    int x;
    int y;
};

struct MyStruct myObject;
myObject.x = 42;
myObject.y = 123;

printf(".x = %i, .y = %i", myObject.x, myObject.y); /* 输出 ".x = 42, .y = 123". */

```

指向对象的成员

解引用后访问成员的语法糖。实际上，形式为 `x->y` 的表达式是 `(*x).y` 的简写——但箭头操作符更清晰，尤其当结构体指针嵌套时。

```

struct MyStruct
{
    int x;
    int y;
};

```

```

int c = 1;
int d = 4;

a++;
printf("a = %d\n",a); /* Will output "a = 2" */
b--;
printf("b = %d\n",b); /* Will output "b = 3" */

if (++c > 1) { /* c is incremented by 1 before being compared in the condition */
    printf("This will print\n"); /* This is printed */
} else {
    printf("This will never print\n"); /* This is not printed */
}

if (d-- < 4) { /* d is decremented after being compared */
    printf("This will never print\n"); /* This is not printed */
} else {
    printf("This will print\n"); /* This is printed */
}

```

As the example for c and d shows, both operators have two forms, as prefix notation and postfix notation. Both have the same effect in incrementing (++) or decrementing (--) the variable, but differ by the value they return: prefix operations do the operation first and then return the value, whereas postfix operations first determine the value that is to be returned, and then do the operation.

Because of this potentially counter-intuitive behaviour, the use of increment/decrement operators inside expressions is controversial.

Section 4.7: Access Operators

The member access operators (dot . and arrow ->) are used to access a member of a `struct`.

Member of object

Evaluates into the lvalue denoting the object that is a member of the accessed object.

```

struct MyStruct
{
    int x;
    int y;
};

struct MyStruct myObject;
myObject.x = 42;
myObject.y = 123;

printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Outputs ".x = 42, .y = 123". */

```

Member of pointed-to object

Syntactic sugar for dereferencing followed by member access. Effectively, an expression of the form `x->y` is shorthand for `(*x).y` — but the arrow operator is much clearer, especially if the structure pointers are nested.

```

struct MyStruct
{
    int x;
    int y;
};

```

```

};

struct MyStruct myObject;
struct MyStruct *p = &myObject;

p->x = 42;
p->y = 123;

printf(".x = %i, .y = %i", p->x, p->y); /* 输出 ".x = 42, .y = 123". */
printf(".x = %i, .y = %i", myObject.x, myObject.y); /* 也输出 ".x = 42, .y = 123". */

```

地址运算符

一元&运算符是地址运算符。它对给定表达式求值，结果对象必须是左值。然后，它求值为一个对象，该对象的类型是指向结果对象类型的指针，并且包含结果对象的地址。

```

int x = 3;
int *p = &x;
printf("%p = %p", (void *)&x, (void *)p); /* 输出 "A = A", 具体值由实现定义. */
*/

```

解引用

一元*运算符对指针进行解引用。它求值为对给定表达式求值后得到的指针进行解引用所得到的左值。

```

int x = 42;
int *p = &x;
printf("x = %d, *p = %d", x, *p); /* 输出 "x = 42, *p = 42". */
*p = 123;

printf("x = %d, *p = %d", x, *p); /* 输出 "x = 123, *p = 123". */

```

索引

索引是指针加法后解引用的语法糖。实际上，形式为 `a[i]` 等价于 `*(&a + i)` —— 但更推荐使用显式的下标表示法。

```

int arr[] = { 1, 2, 3, 4, 5 };
printf("arr[2] = %i", arr[2]); /* 输出 "arr[2] = 3". */

```

索引的可互换性

指针加整数是交换律运算（即操作数的顺序不影响结果），所以 `指针 + 整数 == 整数 + 指针`。

其结果是 `arr[3]` 和 `3[arr]` 是等价的。

```

printf("3[arr] = %i", 3[arr]); /* 输出 "3[arr] = 4". */

```

使用表达式 `3[arr]` 替代 `arr[3]` 通常不推荐，因为这会影响代码的可读性。它通常在混淆编程竞赛中较为流行。

```

};

struct MyStruct myObject;
struct MyStruct *p = &myObject;

p->x = 42;
p->y = 123;

printf(".x = %i, .y = %i\n", p->x, p->y); /* Outputs ".x = 42, .y = 123". */
printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Also outputs ".x = 42, .y = 123". */

```

Address-of

The unary & operator is the address of operator. It evaluates the given expression, where the resulting object must be an lvalue. Then, it evaluates into an object whose type is a pointer to the resulting object's type, and contains the address of the resulting object.

```

int x = 3;
int *p = &x;
printf("%p = %p\n", (void *)&x, (void *)p); /* Outputs "A = A", for some implementation-defined A. */
*/

```

Dereference

The unary * operator dereferences a pointer. It evaluates into the lvalue resulting from dereferencing the pointer that results from evaluating the given expression.

```

int x = 42;
int *p = &x;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 42, *p = 42". */

*p = 123;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 123, *p = 123". */

```

Indexing

Indexing is syntactic sugar for pointer addition followed by dereferencing. Effectively, an expression of the form `a[i]` is equivalent to `*(&a + i)` — but the explicit subscript notation is preferred.

```

int arr[] = { 1, 2, 3, 4, 5 };
printf("arr[2] = %i\n", arr[2]); /* Outputs "arr[2] = 3". */

```

Interchangeability of indexing

Adding a pointer to an integer is a commutative operation (i.e. the order of the operands does not change the result) so `pointer + integer == integer + pointer`.

A consequence of this is that `arr[3]` and `3[arr]` are equivalent.

```

printf("3[arr] = %i\n", 3[arr]); /* Outputs "3[arr] = 4". */

```

Usage of an expression `3[arr]` instead of `arr[3]` is generally not recommended, as it affects code readability. It tends to be a popular in obfuscated programming contests.

第4.8节：sizeof运算符

作为操作数的类型

计算给定类型对象的字节大小，类型为 `size_t`。要求类型必须用括号括起来。

```
printf("%zu\n", sizeof(int)); /* 有效, 输出int对象的大小, 平台相关 */  
printf("%zu", sizeof int); /* 无效, 作为参数的类型必须用括号括起来! */
```

作为操作数的表达式

计算给定表达式类型对象的字节大小，类型为 `size_t`。表达式本身不被求值。括号不是必须的；但由于给定表达式必须是一元表达式，通常建议始终使用括号。

```
char ch = 'a';  
printf("%zu\n", sizeof(ch)); /* 有效, 输出char对象的大小, 所有平台均为1. */  
printf("%zu", sizeof ch); /* 有效, 输出char对象的大小, 所有平台均为1. */
```

第4.9节：类型转换运算符

执行从给定表达式求值结果到指定类型的显式转换。

```
int x = 3;  
int y = 4;  
printf("%f", (double)x / y); /* 输出 "0.750000". */
```

这里，`x` 的值被转换为 `double`，除法运算也将 `y` 的值提升为 `double`，除法的结果（一个 `double`）传递给 `printf` 进行打印。

第4.10节：函数调用运算符

第一个操作数必须是函数指针（函数标识符也可接受，因为它会被转换为指向函数的指针），用于标识要调用的函数，其他所有操作数（如果有）统称为函数调用的参数。求值结果为调用相应函数并传入对应参数后的返回值。

```
int myFunction(int x, int y)  
{  
    return x * 2 + y;  
}  
  
int (*fn)(int, int) = &myFunction;  
int x = 42;  
int y = 123;  
  
printf("(fn)(%i, %i) = %i\n", x, y, (*fn)(x, y)); /* 输出 "fn(42, 123) = 207". */  
printf("fn(%i, %i) = %i\n", x, y, fn(x, y)); /* 另一种形式：你不需要显式地解引用 */
```

Section 4.8: sizeof Operator

With a type as operand

Evaluates into the size in bytes, of type `size_t`, of objects of the given type. Requires parentheses around the type.

```
printf("%zu\n", sizeof(int)); /* Valid, outputs the size of an int object, which is platform-dependent. */  
printf("%zu\n", sizeof int); /* Invalid, types as arguments need to be surrounded by parentheses! */
```

With an expression as operand

Evaluates into the size in bytes, of type `size_t`, of objects of the type of the given expression. The expression itself is not evaluated. Parentheses are not required; however, because the given expression must be unary, it's considered best practice to always use them.

```
char ch = 'a';  
printf("%zu\n", sizeof(ch)); /* Valid, will output the size of a char object, which is always 1 for all platforms. */  
printf("%zu\n", sizeof ch); /* Valid, will output the size of a char object, which is always 1 for all platforms. */
```

Section 4.9: Cast Operator

Performs an *explicit* conversion into the given type from the value resulting from evaluating the given expression.

```
int x = 3;  
int y = 4;  
printf("%f\n", (double)x / y); /* Outputs "0.750000". */
```

Here the value of `x` is converted to a `double`, the division promotes the value of `y` to `double`, too, and the result of the division, a `double` is passed to `printf` for printing.

Section 4.10: Function Call Operator

The first operand must be a function pointer (a function designator is also acceptable because it will be converted to a pointer to the function), identifying the function to call, and all other operands, if any, are collectively known as the function call's arguments. Evaluates into the return value resulting from calling the appropriate function with the respective arguments.

```
int myFunction(int x, int y)  
{  
    return x * 2 + y;  
}  
  
int (*fn)(int, int) = &myFunction;  
int x = 42;  
int y = 123;  
  
printf("(fn)(%i, %i) = %i\n", x, y, (*fn)(x, y)); /* Outputs "fn(42, 123) = 207". */  
printf("fn(%i, %i) = %i\n", x, y, fn(x, y)); /* Another form: you don't need to dereference explicitly */
```

第4.11节：自增 / 自减

自增和自减运算符存在前缀和后缀两种形式。

```
int a = 1;
int b = 1;
int tmp = 0;

tmp = ++a;      /* 将a加一，并返回新值；a == 2, tmp == 2 */
tmp = a++;      /* 将a加一，但返回旧值；a == 3, tmp == 2 */
tmp = --b;      /* 将b减一，并返回新值；b == 0, tmp == 0 */
tmp = b--;      /* 将b减一，但返回旧值；b == -1, tmp == 0 */
```

注意算术运算不会引入[序列点](#)，因此某些包含++或--运算符的表达式可能会引入未定义行为。

第4.12节：赋值运算符

将右操作数的值赋给左操作数所指的存储位置，并返回该值。

```
int x = 5;      /* 变量x保存值5。返回5。 */
char y = 'c';   /* 变量y保存值99。返回99
                  * (因为字符'c'在ASCII表中对应的值是99)。
                  */
float z = 1.5;  /* 变量z保存值1.5。返回1.5。 */
char const* s = "foo"; /* 变量s保存字符串'foo'第一个字符的地址。 */
/*
```

几种算术运算有一个复合赋值运算符。

```
a += b /* 等同于：a = a + b */
a -= b /* 等同于：a = a - b */
a *= b /* 等同于：a = a * b */
a /= b /* 等同于：a = a / b */
a %= b /* 等同于：a = a % b */
a &= b /* 等同于：a = a & b */
a |= b /* 等同于：a = a | b */
a ^= b /* 等同于：a = a ^ b */
a <<= b /* 等同于：a = a << b */
a >>= b /* 等同于：a = a >> b */
```

这些复合赋值的一个重要特征是左侧表达式(a)只被计算一次。例如，如果p是一个指针

```
*p += 27;
```

只对p解引用一次，而下面的写法则解引用了两次。

```
*p = *p + 27;
```

还应注意，赋值如a = b的结果被称为一个右值(rvalue)。因此，赋值实际上有一个值，这个值可以赋给另一个变量。这允许在一条语句中链式赋值以设置多个变量。

这个右值(rvalue)可以用于控制表达式中，如if语句(或循环或switch语句)中，用于判断条件

Section 4.11: Increment / Decrement

The increment and decrement operators exist in *prefix* and *postfix* form.

```
int a = 1;
int b = 1;
int tmp = 0;

tmp = ++a;      /* increments a by one, and returns new value; a == 2, tmp == 2 */
tmp = a++;      /* increments a by one, but returns old value; a == 3, tmp == 2 */
tmp = --b;      /* decrements b by one, and returns new value; b == 0, tmp == 0 */
tmp = b--;      /* decrements b by one, but returns old value; b == -1, tmp == 0 */
```

Note that arithmetic operations do not introduce [sequence points](#), so certain expressions with ++ or -- operators may introduce undefined behaviour.

Section 4.12: Assignment Operators

Assigns the value of the right-hand operand to the storage location named by the left-hand operand, and returns the value.

```
int x = 5;      /* Variable x holds the value 5. Returns 5. */
char y = 'c';   /* Variable y holds the value 99.
                  * (as the character 'c' is represented in the ASCII table with 99).
                  */
float z = 1.5;  /* variable z holds the value 1.5. Returns 1.5. */
char const* s = "foo"; /* Variable s holds the address of the first character of the string 'foo'.
                  */
/*
```

Several arithmetical operations have a *compound assignment operator*.

```
a += b /* equal to: a = a + b */
a -= b /* equal to: a = a - b */
a *= b /* equal to: a = a * b */
a /= b /* equal to: a = a / b */
a %= b /* equal to: a = a % b */
a &= b /* equal to: a = a & b */
a |= b /* equal to: a = a | b */
a ^= b /* equal to: a = a ^ b */
a <<= b /* equal to: a = a << b */
a >>= b /* equal to: a = a >> b */
```

One important feature of these compound assignments is that the expression on the left hand side (a) is only evaluated once. E.g if p is a pointer

```
*p += 27;
```

dereferences p only once, whereas the following does so twice.

```
*p = *p + 27;
```

It should also be noted that the result of an assignment such as a = b is what is known as an *rvalue*. Thus, the assignment actually has a value which can then be assigned to another variable. This allows the chaining of assignments to set multiple variables in a single statement.

This *rvalue* can be used in the controlling expressions of if statements (or loops or switch statements) that guard

另一个表达式或函数调用结果上的一些代码。例如：

```
char *buffer;
if ((buffer = malloc(1024)) != NULL)
{
    /* 使用 buffer 做一些操作 */
    free(buffer);
}
else
{
    /* 报告分配失败 */
}
```

因此，必须小心避免一个常见的拼写错误，这可能导致神秘的错误。

```
int a = 2;
/* ... */
if (a = 1)
    /* 删掉我硬盘上的所有文件 */
```

这将导致灾难性的后果，因为 `a = 1` 总是会被计算为 1，因此 `if` 语句的控制表达式总是为真（在这里阅读更多关于这个常见陷阱的内容）。作者几乎可以确定是想使用相等运算符（`==`），如下所示：

```
int a = 2;
/* ... */
if (a == 1)
    /* 删掉我硬盘上的所有文件 */
```

运算符结合性

```
int a, b = 1, c = 2;
a = b = c;
```

这将把 `c` 赋值给 `b`, `b` 返回后，再赋值给 `a`。之所以如此，是因为所有赋值运算符都是右结合的，这意味着表达式中最右边的操作先被计算，然后从右向左进行。

第4.13节：逻辑运算符

逻辑与

对两个操作数执行逻辑布尔与运算，如果两个操作数都非零，则返回1。逻辑与运算符的类型是 `int`。

```
0 && 0 /* 返回0。 */
0 && 1 /* 返回0。 */
2 && 0 /* 返回 0。 */
2 && 3 /* 返回 1. */
```

逻辑或

对两个操作数执行逻辑布尔或运算，如果任一操作数非零则返回 1。逻辑或运算符的类型为 `int`。

```
0 || 0 /* 返回 0. */
```

some code on the result of another expression or function call. For example:

```
char *buffer;
if ((buffer = malloc(1024)) != NULL)
{
    /* do something with buffer */
    free(buffer);
}
else
{
    /* report allocation failure */
}
```

Because of this, care must be taken to avoid a common typo which can lead to mysterious bugs.

```
int a = 2;
/* ... */
if (a = 1)
    /* Delete all files on my hard drive */
```

This will have disastrous results, as `a = 1` will always evaluate to 1 and thus the controlling expression of the `if` statement will always be true (read more about this common pitfall here). The author almost certainly meant to use the equality operator (`==`) as shown below:

```
int a = 2;
/* ... */
if (a == 1)
    /* Delete all files on my hard drive */
```

Operator Associativity

```
int a, b = 1, c = 2;
a = b = c;
```

This assigns `c` to `b`, which returns `b`, which is then assigned to `a`. This happens because all assignment-operators have right associativity, that means the rightmost operation in the expression is evaluated first, and proceeds from right to left.

Section 4.13: Logical Operators

Logical AND

Performs a logical boolean AND-ing of the two operands returning 1 if both of the operands are non-zero. The logical AND operator is of type `int`.

```
0 && 0 /* Returns 0. */
0 && 1 /* Returns 0. */
2 && 0 /* Returns 0. */
2 && 3 /* Returns 1. */
```

Logical OR

Performs a logical boolean OR-ing of the two operands returning 1 if any of the operands are non-zero. The logical OR operator is of type `int`.

```
0 || 0 /* Returns 0. */
```

```
0 || 1 /* 返回 1. */
2 || 0 /* 返回 1. */
2 || 3 /* 返回 1. */
```

逻辑非

执行逻辑取反。逻辑非运算符的类型为 int。非运算符检查是否至少有一位等于 1，若是则返回 0，否则返回 1；

```
!1 /* 返回0. */
!5 /* 返回0. */
!0 /* 返回1. */
```

短路求值

以下是&&和||共有的一些关键特性：

- 左操作数 (LHS) 在右操作数 (RHS) 被评估之前会被完全评估，左操作数和右操作数的评估之间存在一个序列点，最重要的是，如果左操作数的结果已经决定了整体结果，则右操作数根本不会被评估。
-

这意味着：

- 如果左操作数的值为“真”（非零），则||的右操作数不会被评估（因为“真或任何值”的结果为“真”），
- 如果左操作数的值为“假”（零），则&&的右操作数不会被评估（因为“假且任何值”的结果为“假”）。

这很重要，因为它允许你编写如下代码：

```
const char *name_for_value(int value)
{
    static const char *names[] = { "zero", "one", "two", "three", };
    enum { NUM_NAMES = sizeof(names) / sizeof(names[0]) };
    return (value >= 0 && value < NUM_NAMES) ? names[value] : "infinity";
}
```

如果传递给函数的是负值，value >= 0 条件将被判定为假，且 value < NUM_NAMES 条件不会被计算。

第4.14节：指针运算

指针加法

给定一个指针和一个标量类型 N，计算结果是指向被指向类型中紧接着被指向对象的第 N 个元素的指针。

```
int arr[] = {1, 2, 3, 4, 5};
printf("%(arr + 3) = %i", *(arr + 3)); /* 输出"4", arr的第四个元素. */
```

无论指针是作为操作数值还是标量值使用都无关紧要。这意味着诸如 3 + arr 是有效的。如果 arr[k] 是数组的第 k+1 个成员，那么 arr+k 是指向 arr[k] 的指针。换句话说，arr 或 arr+0 是指向 arr[0] 的指针，arr+1 是指向 arr[1] 的指针，依此类推。一般来说，*(arr+k) 与 arr[k] 相同。

```
0 || 1 /* Returns 1. */
2 || 0 /* Returns 1. */
2 || 3 /* Returns 1. */
```

Logical NOT

Performs a logical negation. The logical NOT operator is of type int. The NOT operator checks if at least one bit is equal to 1, if so it returns 0. Else it returns 1;

```
!1 /* Returns 0. */
!5 /* Returns 0. */
!0 /* Returns 1. */
```

Short-Circuit Evaluation

There are some crucial properties common to both && and ||:

- the left-hand operand (LHS) is fully evaluated before the right-hand operand (RHS) is evaluated at all,
- there is a sequence point between the evaluation of the left-hand operand and the right-hand operand,
- and, most importantly, the right-hand operand is not evaluated at all if the result of the left-hand operand determines the overall result.

This means that:

- if the LHS evaluates to 'true' (non-zero), the RHS of || will not be evaluated (because the result of 'true OR anything' is 'true'),
- if the LHS evaluates to 'false' (zero), the RHS of && will not be evaluated (because the result of 'false AND anything' is 'false').

This is important as it permits you to write code such as:

```
const char *name_for_value(int value)
{
    static const char *names[] = { "zero", "one", "two", "three", };
    enum { NUM_NAMES = sizeof(names) / sizeof(names[0]) };
    return (value >= 0 && value < NUM_NAMES) ? names[value] : "infinity";
}
```

If a negative value is passed to the function, the value >= 0 term evaluates to false and the value < NUM_NAMES term is not evaluated.

Section 4.14: Pointer Arithmetic

Pointer addition

Given a pointer and a scalar type N, evaluates into a pointer to the Nth element of the pointed-to type that directly succeeds the pointed-to object in memory.

```
int arr[] = {1, 2, 3, 4, 5};
printf("%(arr + 3) = %i\n", *(arr + 3)); /* Outputs "4", arr's fourth element. */
```

It does not matter if the pointer is used as the operand value or the scalar value. This means that things such as 3 + arr are valid. If arr[k] is the k+1 member of an array, then arr+k is a pointer to arr[k]. In other words, arr or arr+0 is a pointer to arr[0], arr+1 is a pointer to arr[1], and so on. In general, *(arr+k) is same as arr[k].

与通常的算术不同，向指向 int 的指针加 1 会使当前地址值增加 4 字节。由于数组名是常量指针，+ 是我们通过数组名使用指针表示法访问数组成员的唯一运算符。然而，通过定义指向数组的指针，我们可以更灵活地处理数组中的数据。例如，我们可以如下打印数组的成员：

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    for(k = 0; k < N; k++)
    {
        printf("%d", *(arr + k));
    }
    return 0;
}
```

通过定义一个指向数组的指针，上述程序等价于以下代码：

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr; /* 或者 int *ptr = &arr[0]; */
    for(k = 0; k < N; k++)
    {
        printf("%d", ptr[k]);/* 或者 p
        rintf("%d", *(ptr + k)); /* 或者 printf("%d",
        *ptr++); */
    }
    return 0;
}
```

可以看到，数组 arr 的成员是通过运算符 + 和 ++ 访问的。指针 ptr 还可以使用的其他运算符有 - 和 --。

指针减法

给定两个指向相同类型的指针，计算结果是一个类型为 ptrdiff_t 的对象，该对象保存一个标量值，该值是加到第二个指针上以获得第一个指针值的偏移量。

```
int arr[] = {1, 2, 3, 4, 5};
int *p = &arr[2];
int *q = &arr[3];
ptrdiff_t diff = q - p;

printf("q - p = %ti", diff); /* 输出 "1". */
printf("(p + (q - p)) = %d", *(p + diff)); /* 输出 "4". */
```

第4.15节：_Alignof

版本 ≥ C11

Unlike the usual arithmetic, addition of 1 to a pointer to an `int` will add 4 bytes to the current address value. As array names are constant pointers, + is the only operator we can use to access the members of an array via pointer notation using the array name. However, by defining a pointer to an array, we can get more flexibility to process the data in an array. For example, we can print the members of an array as follows:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", *(arr + k));
    }
    return 0;
}
```

By defining a pointer to the array, the above program is equivalent to the following:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr; /* or int *ptr = &arr[0]; */
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", ptr[k]);
        /* or printf("\n\t%d", *(ptr + k)); */
        /* or printf("\n\t%d", *ptr++); */
    }
    return 0;
}
```

See that the members of the array arr are accessed using the operators + and ++. The other operators that can be used with the pointer ptr are - and --.

Pointer subtraction

Given two pointers to the same type, evaluates into an object of type ptrdiff_t that holds the scalar value that must be added to the second pointer in order to obtain the value of the first pointer.

```
int arr[] = {1, 2, 3, 4, 5};
int *p = &arr[2];
int *q = &arr[3];
ptrdiff_t diff = q - p;

printf("q - p = %ti\n", diff); /* Outputs "1". */
printf("(p + (q - p)) = %d\n", *(p + diff)); /* Outputs "4". */
```

Section 4.15: _Alignof

Version ≥ C11

查询指定类型的对齐要求。对齐要求是2的正整数次幂，表示两个该类型对象之间可能分配的字节数。在C语言中，对齐要求以 `size_t` 为单位测量。

类型名不得是不完整类型或函数类型。如果使用数组作为类型，则使用数组元素的类型。

该操作符通常通过`<stdalign.h>`中的便利宏`alignof`访问。

```
int main(void)
{
    printf("char的对齐 = %zu", alignof(char));printf("max_align_t的对
齐 = %zu", alignof(max_align_t));printf("alignof(float[10]) = %zu", alignof(float[10
]));
    printf("alignof(struct{char c; int n;}) = %zu", alignof(struct {char c; int n;}));
}
```

可能的输出：

```
char的对齐 = 1
max_align_t的对齐 = 16
alignof(float[10]) = 4
alignof(struct{char c; int n;}) = 4
```

http://en.cppreference.com/w/c/language/_Alignof

Queries the alignment requirement for the specified type. The alignment requirement is a positive integral power of 2 representing the number of bytes between which two objects of the type may be allocated. In C, the alignment requirement is measured in `size_t`.

The type name may not be an incomplete type nor a function type. If an array is used as the type, the type of the array element is used.

This operator is often accessed through the convenience macro `alignof` from `<stdalign.h>`.

```
int main(void)
{
    printf("Alignment of char = %zu\n", alignof(char));
    printf("Alignment of max_align_t = %zu\n", alignof(max_align_t));
    printf("alignof(float[10]) = %zu\n", alignof(float[10]));
    printf("alignof(struct{char c; int n;}) = %zu\n",
        alignof(struct {char c; int n;}));
}
```

Possible Output:

```
Alignment of char = 1
Alignment of max_align_t = 16
alignof(float[10]) = 4
alignof(struct{char c; int n;}) = 4
```

http://en.cppreference.com/w/c/language/_Alignof

第5章：布尔类型

第5.1节：使用 stdbool.h

版本 ≥ C99

使用系统头文件stdbool.h可以让你使用bool作为布尔数据类型。`true`的值为1，`false`的值为0。

```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool x = true; /* 等同于 bool x = 1; */
    bool y = false; /* 等同于 bool y = 0; */
    if (x) /* 功能上等同于 if (x != 0) 或 if (x != false) */
    {
        puts("这将被打印！");
    }
    if (!y) /* 功能上等同于 if (y == 0) 或 if (y == false) */
    {
        puts("这也将被打印！");
    }
}
```

bool 只是数据类型 _Bool 的一种简洁写法。当数字或指针转换为该类型时，有特殊规则。

第5.2节：使用#define

C语言的所有版本中，除了0以外，任何整数值在比较操作中都会被视为true，整数值0则被视为false。如果你没有C99版本中的_Bool或bool类型，也可以使用#define宏在C语言中模拟布尔数据类型，你可能仍会在遗留代码中看到这种用法。

```
#include <stdio.h>

#define bool int
#define true 1
#define false 0

int main(void) {
    bool x = true; /* 等同于 int x = 1; */
    bool y = false; /* 等同于 int y = 0; */
    if (x) /* 功能上等同于 if (x != 0) 或 if (x != false) */
    {
        puts("这将被打印！");
    }
    if (!y) /* 功能上等同于 if (y == 0) 或 if (y == false) */
    {
        puts("这也将被打印！");
    }
}
```

不要在新代码中引入这种写法，因为这些宏的定义可能与现代的<stdbool.h>的用法冲突。

Chapter 5: Boolean

Section 5.1: Using stdbool.h

Version ≥ C99

Using the system header file stdbool.h allows you to use bool as a Boolean data type. `true` evaluates to 1 and `false` evaluates to 0.

```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool x = true; /* equivalent to bool x = 1; */
    bool y = false; /* equivalent to bool y = 0; */
    if (x) /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

bool is just a nice spelling for the data type _Bool. It has special rules when numbers or pointers are converted to it.

Section 5.2: Using #define

C of all versions, will effectively treat any integer value other than 0 as `true` for comparison operators and the integer value 0 as `false`. If you don't have _Bool or bool as of C99 available, you could simulate a Boolean data type in C using #define macros, and you might still find such things in legacy code.

```
#include <stdio.h>

#define bool int
#define true 1
#define false 0

int main(void) {
    bool x = true; /* Equivalent to int x = 1; */
    bool y = false; /* Equivalent to int y = 0; */
    if (x) /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

Don't introduce this in new code since the definition of these macros might clash with modern uses of <stdbool.h>.

第5.3节：使用内置类型 _Bool

版本 ≥ C99

在C标准版本C99中新增的 _Bool 也是一种原生的C数据类型。它能够存储值0（表示false）和1（表示true）。

```
#include <stdio.h>

int main(void) {
    _Bool x = 1;
    _Bool y = 0;
    if(x) /* 等同于 if (x == 1) */
    {
        puts("这将被打印！");
    }
    if (!y) /* 等同于 if (y == 0) */
    {
        puts("这也将被打印！");
    }
}
```

_Bool 是一种整数类型，但对其他类型的转换有特殊规则。其结果类似于其他类型在 if 表达式中的用法。以下内容

```
_Bool z = X;
```

- 如果 X 是算术类型（任何数字类型），当 X == 0 时，z 变为 0。否则 z 变为 1。
- 如果 X 是指针类型，当 X 是空指针时，z 变为 0，否则变为 1。

要使用更友好的拼写 bool、false 和 true，需要包含头文件 <stdbool.h>。

第5.4节：布尔表达式中的整数和指针

所有整数或指针都可以用于被解释为“真值”的表达式中。

```
int main(int argc, char* argv[]) {
    if (argc % 4) {
        puts("参数数量不能被4整除");
    } else {
        puts("参数数量能被4整除");
    }
    ...
}
```

表达式 argc % 4 被计算后会得到值 0、1、2 或 3 中的一个。第一个值 0 是唯一的“假”，会使程序执行 else 部分。其他所有值都是“真”，会进入 if 部分。

```
double* A = malloc(n*sizeof *A);
if (!A) {
    perror("分配问题");
    exit(EXIT_FAILURE);
}
```

这里对指针 A 进行判断，如果它是空指针，则检测到错误并退出程序。

许多人更喜欢写成 A == NULL，但如果你在其他复杂表达式中使用这样的指针比较，阅读起来会很快变得困难。

Section 5.3: Using the Intrinsic (built-in) Type _Bool

Version ≥ C99

Added in the C standard version C99, _Bool is also a native C data type. It is capable of holding the values 0 (for false) and 1 (for true).

```
#include <stdio.h>

int main(void) {
    _Bool x = 1;
    _Bool y = 0;
    if(x) /* Equivalent to if (x == 1) */
    {
        puts("This will print!");
    }
    if (!y) /* Equivalent to if (y == 0) */
    {
        puts("This will also print!");
    }
}
```

_Bool is an integer type but has special rules for conversions from other types. The result is analogous to the usage of other types in if expressions. In the following

```
_Bool z = X;
```

- If X has an arithmetic type (is any kind of number), z becomes 0 if X == 0. Otherwise z becomes 1.
- If X has a pointer type, z becomes 0 if X is a null pointer and 1 otherwise.

To use nicer spellings bool, false and true you need to use <stdbool.h>.

Section 5.4: Integers and pointers in Boolean expressions

All integers or pointers can be used in an expression that is interpreted as "truth value".

```
int main(int argc, char* argv[]) {
    if (argc % 4) {
        puts("arguments number is not divisible by 4");
    } else {
        puts("argument number is divisible by 4");
    }
    ...
}
```

The expression argc % 4 is evaluated and leads to one of the values 0, 1, 2 or 3. The first, 0 is the only value that is "false" and brings execution into the else part. All other values are "true" and go into the if part.

```
double* A = malloc(n*sizeof *A);
if (!A) {
    perror("allocation problems");
    exit(EXIT_FAILURE);
}
```

Here the pointer A is evaluated and if it is a null pointer, an error is detected and the program exits.

Many people prefer to write something as A == NULL, instead, but if you have such pointer comparisons as part of

其他复杂表达式中使用这样的指针比较，阅读起来会很快变得困难。

```
char const* s = ....; /* 我们接收到的某个指针 */
if (s != NULL && s[0] != '\0' && isalpha(s[0])) {
    printf("这开始得很好, %c 是字母", s[0]);}
```

要检查这一点，你必须扫描表达式中的复杂代码，并确定运算符的优先级。

```
char const* s = ....; /* 我们接收到的某个指针 */
if (s && s[0] && isalpha(s[0])) {
    printf("这开始得很好, %c 是字母", s[0]);}
```

相对容易捕捉：如果指针有效，我们检查第一个字符是否非零，然后检查它是否是一个字母。

第5.5节：使用typedef定义bool类型

考虑到大多数调试器无法识别#define宏，但可以检查enum常量，可能希望这样做：

```
#if __STDC_VERSION__ < 199900L
typedef enum { false, true } bool;
/* 现代C代码可能期望这些是宏。 */
#ifndef bool
#define bool bool
#endif
#ifndef true
#define true true
#endif
#ifndef false
#define false false
#endif
#else
#include <stdbool.h>
#endif

/* 程序后续部分 ... */
bool b = true;
```

这允许历史版本的C编译器正常工作，但如果代码使用现代C编译器编译，则仍然向前兼容。

有关typedef的更多信息，请参见Typedef，关于enum的更多信息，请参见枚举（Enumerations）

other complicated expressions, things become quickly difficult to read.

```
char const* s = ....; /* some pointer that we receive */
if (s != NULL && s[0] != '\0' && isalpha(s[0])) {
    printf("this starts well, %c is alphabetic\n", s[0]);
}
```

For this to check, you'd have to scan a complicated code in the expression and be sure about operator preference.

```
char const* s = ....; /* some pointer that we receive */
if (s && s[0] && isalpha(s[0])) {
    printf("this starts well, %c is alphabetic\n", s[0]);
}
```

is relatively easy to capture: if the pointer is valid we check if the first character is non-zero and then check if it is a letter.

Section 5.5: Defining a bool type using typedef

Considering that most debuggers are not aware of #define macros, but can check enum constants, it may be desirable to do something like this:

```
#if __STDC_VERSION__ < 199900L
typedef enum { false, true } bool;
/* Modern C code might expect these to be macros. */
#ifndef bool
#define bool bool
#endif
#ifndef true
#define true true
#endif
#ifndef false
#define false false
#endif
#else
#include <stdbool.h>
#endif

/* Somewhere later in the code ... */
bool b = true;
```

This allows compilers for historic versions of C to function, but remains forward compatible if the code is compiled with a modern C compiler.

For more information on [typedef](#), see [Typedef](#), for more on [enum](#) see [Enumerations](#)

第6章：字符串

在C语言中，字符串不是内置类型。C字符串的约定是使用一个以空字符'\0'结尾的一维字符数组。

这意味着内容为"abc"的C字符串将包含四个字符'a'、'b'、'c'和'\0'。

请参见字符串基础介绍示例。

第6.1节：分词：strtok()、strtok_r() 和 strtok_s()

函数strtok 使用一组分隔符将字符串拆分成更小的字符串或标记 (tokens) 。

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int toknum = 0;
    char src[] = "Hello,, world!";
    const char delimiters[] = ", !";
    char *token = strtok(src, delimiters);
    while (token != NULL)
    {
        printf("%d: [%s]", ++toknum, token);      to
ken = strtok(NULL, delimiters);}

    /* 源字符串现在变为 "Hello\0, world\0\0" */
}
```

输出：

```
1: [Hello]
2: [world]
```

分隔符字符串可以包含一个或多个分隔符，并且每次调用strtok时可以使用不同的分隔符字符串。

继续对同一源字符串调用strtok时，不应再次传入源字符串，而应将第一个参数传为NULL。如果再次传入相同的源字符串，则会重新对第一个标记进行分词。

也就是说，给定相同的分隔符，strtok 只会再次返回第一个标记。

请注意，由于strtok不会为标记分配新的内存，它会修改源字符串。也就是说，在上述示例中，字符串 src将被操作以生成由strtok调用返回的指针所引用的标记。这意味着源字符串不能是const（因此不能是字符串字面量）。这也意味着分隔字节的身份丢失（即在示例中，“,”和“!”实际上从源字符串中被删除，且无法判断匹配的是哪个分隔符字符）。

还要注意，源字符串中多个连续的分隔符被视为一个；在示例中，第二个逗号被忽略。

strtok既不是线程安全的，也不是可重入的，因为它在解析时使用了静态缓冲区。这意味着如果一个函数调用了strtok，那么在它使用strtok期间，它调用的任何函数都不能使用strtok，并且任何正在使用strtok的函数也不能调用它。

Chapter 6: Strings

In C, a string is not an intrinsic type. A C-string is the convention to have a one-dimensional array of characters which is terminated by a null-character, by a '\0'.

This means that a C-string with a content of "abc" will have four characters 'a', 'b', 'c' and '\0'.

See the basic introduction to strings example.

Section 6.1: Tokenisation: strtok(), strtok_r() and strtok_s()

The function `strtok` breaks a string into smaller strings, or tokens, using a set of delimiters.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int toknum = 0;
    char src[] = "Hello,, world!";
    const char delimiters[] = ", !";
    char *token = strtok(src, delimiters);
    while (token != NULL)
    {
        printf("%d: [%s]\n", ++toknum, token);
        token = strtok(NULL, delimiters);
    }
    /* source is now "Hello\0, world\0\0" */
}
```

Output:

```
1: [Hello]
2: [world]
```

The string of delimiters may contain one or more delimiters and different delimiter strings may be used with each call to `strtok`.

Calls to `strtok` to continue tokenizing the same source string should not pass the source string again, but instead pass NULL as the first argument. If the same source string is passed then the first token will instead be re-tokenized. That is, given the same delimiters, `strtok` would simply return the first token again.

Note that as `strtok` does not allocate new memory for the tokens, it modifies the source string. That is, in the above example, the string src will be manipulated to produce the tokens that are referenced by the pointer returned by the calls to `strtok`. This means that the source string cannot be `const` (so it can't be a string literal). It also means that the identity of the delimiting byte is lost (i.e. in the example the "," and "!" are effectively deleted from the source string and you cannot tell which delimiter character matched).

Note also that multiple consecutive delimiters in the source string are treated as one; in the example, the second comma is ignored.

`strtok` is neither thread safe nor re-entrant because it uses a static buffer while parsing. This means that if a function calls `strtok`, no function that it calls while it is using `strtok` can also use `strtok`, and it cannot be called by any function that is itself using `strtok`.

下面的示例演示了`strtok`不可重入所导致的问题：

```
char src[] = "1.2,3.5,4.2";
char *first = strtok(src, ",");

do
{
    char *part;
    /* 嵌套调用 strtok 不会按预期工作 */
    printf("[%s]", first); part =
strtok(first, ".");while (part != NULL)

    {
        printf(" [%s]", part); p
part = strtok(NULL, ".");
    }

} 当 ((first = strtok(NULL, ",")) != NULL);
```

输出：

```
[1.2]
[1]
[2]
```

预期的操作是外层的 `do while` 循环应创建三个由每个十进制数字字符串组成的标记 ("1.2"、"3.5"、"4.2")，对于每个字符串，内层循环中的 `strtok` 调用应将其拆分为单独的数字字符串 ("1"、"2"、"3"、"5"、"4"、"2")。

然而，由于 `strtok` 不是可重入的，这种情况并未发生。相反，第一次 `strtok` 正确创建了 "1.2\0" 标记，内层循环也正确创建了标记 "1" 和 "2"。但随后外层循环中的 `strtok` 位于内层循环使用的字符串末尾，立即返回 `NULL`。数组 `src` 的第二和第三个子字符串根本没有被分析。

版本 < C11

标准 C 库不包含线程安全或可重入版本，但其他一些库包含，例如 POSIX 的 `strtok_r`。注意在 MSVC 上，`strtok` 的等价函数 `strtok_s` 是线程安全的。

版本 ≥ C11

C11 有一个可选部分，附录 K，提供了一个名为 `strtok_s` 的线程安全且可重入的版本。你可以通过 `_STDC_LIB_EXT1` 来检测该特性。该可选部分支持度不广泛。

`strtok_s` 函数与 POSIX 的 `strtok_r` 函数的区别在于它防止存储到被标记的字符串之外，并且会检查运行时约束条件。不过在正确编写的程序中，`strtok_s` 和 `strtok_r` 的行为是相同的。

使用 `strtok_s` 处理该示例现在会得到正确的结果，如下所示：

```
/* 你必须声明你想使用附件K */
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

#ifndef __STDC_LIB_EXT1__
# error "we need strtok_s from Annex K"
#endif
```

An example that demonstrates the problems caused by the fact that `strtok` is not re-entrant is as follows:

```
char src[] = "1.2,3.5,4.2";
char *first = strtok(src, ",");

do
{
    char *part;
    /* Nested calls to strtok do not work as desired */
    printf("[%s]\n", first);
    part = strtok(first, ".");
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok(NULL, ".");
    }
} while ((first = strtok(NULL, ",")) != NULL);
```

Output:

```
[1.2]
[1]
[2]
```

The expected operation is that the outer `do while` loop should create three tokens consisting of each decimal number string ("1.2", "3.5", "4.2"), for each of which the `strtok` calls for the inner loop should split it into separate digit strings ("1", "2", "3", "5", "4", "2").

However, because `strtok` is not re-entrant, this does not occur. Instead the first `strtok` correctly creates the "1.2\0" token, and the inner loop correctly creates the tokens "1" and "2". But then the `strtok` in the outer loop is at the end of the string used by the inner loop, and returns `NULL` immediately. The second and third substrings of the `src` array are not analyzed at all.

Version < C11

The standard C libraries do not contain a thread-safe or re-entrant version but some others do, such as POSIX' `strtok_r`. Note that on MSVC the `strtok` equivalent, `strtok_s` is thread-safe.

Version ≥ C11

C11 has an optional part, Annex K, that offers a thread-safe and re-entrant version named `strtok_s`. You can test for the feature with `_STDC_LIB_EXT1__`. This optional part is not widely supported.

The `strtok_s` function differs from the POSIX `strtok_r` function by guarding against storing outside of the string being tokenized, and by checking runtime constraints. On correctly written programs, though, the `strtok_s` and `strtok_r` behave the same.

Using `strtok_s` with the example now yields the correct response, like so:

```
/* you have to announce that you want to use Annex K */
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

#ifndef __STDC_LIB_EXT1__
# error "we need strtok_s from Annex K"
#endif
```

```

char src[] = "1.2,3.5,4.2";
char *next = NULL;
char *first = strtok_s(src, ",", &next);

do
{
    char *part;
    char *posn;

    printf("[%s]", first); part =
strtok_s(first, ".", &posn);while (part != NULL)

    {
        printf(" [%s]", part);      p
art = strtok_s(NULL, ".", &posn);}

}
while ((first = strtok_s(NULL, ",", &next)) != NULL);

```

输出结果将是：

```
[1.2]
[1]
[2]
[3.5]
[3]
[5]
[4.2]
[4]
[2]
```

第6.2节：字符串字面量

字符串字面量表示以空字符结尾的、具有静态存储期的char数组。由于它们具有静态存储期，字符串字面量或指向同一底层数组的指针可以安全地以多种方式使用，而自动数组的指针则不行。例如，从函数返回字符串字面量具有定义良好的行为：

```
const char *get_hello()
{
    return "Hello, World!"; /* 安全 */
}
```

出于历史原因，与字符串字面量对应的数组元素在形式上并不是const的。

然而，任何试图修改它们的行为都是未定义的。通常，试图修改与字符串字面量对应的数组的程序会崩溃或出现其他故障。

```
char *foo = "hello";
foo[0] = 'y'; /* 未定义行为 - 错误！ */
```

当指针指向字符串字面量——或者有时可能指向字符串字面量时——建议将该指针所指对象声明为const，以避免意外触发此类未定义行为。

```
const char *foo = "hello";
/* 好的：不能修改foo指向的字符串 */
```

另一方面，指向字符串字面量底层数组的指针本身并没有特殊之处；其值可以自由修改以指向其他内容：

```

char src[] = "1.2,3.5,4.2";
char *next = NULL;
char *first = strtok_s(src, ",", &next);

do
{
    char *part;
    char *posn;

    printf("[%s]\n", first);
    part = strtok_s(first, ".", &posn);
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok_s(NULL, ".", &posn);
    }
}
while ((first = strtok_s(NULL, ",", &next)) != NULL);

```

And the output will be:

```
[1.2]
[1]
[2]
[3.5]
[3]
[5]
[4.2]
[4]
[2]
```

Section 6.2: String literals

String literals represent null-terminated, static-duration arrays of `char`. Because they have static storage duration, a string literal or a pointer to the same underlying array can safely be used in several ways that a pointer to an automatic array cannot. For example, returning a string literal from a function has well-defined behavior:

```
const char *get_hello()
{
    return "Hello, World!"; /* safe */
}
```

For historical reasons, the elements of the array corresponding to a string literal are not formally `const`.

Nevertheless, any attempt to modify them has undefined behavior. Typically, a program that attempts to modify the array corresponding to a string literal will crash or otherwise malfunction.

```
char *foo = "hello";
foo[0] = 'y'; /* Undefined behavior - BAD! */
```

Where a pointer points to a string literal -- or where it sometimes may do -- it is advisable to declare that pointer's referent `const` to avoid engaging such undefined behavior accidentally.

```
const char *foo = "hello";
/* GOOD: can't modify the string pointed to by foo */
```

On the other hand, a pointer to or into the underlying array of a string literal is not itself inherently special; its value can freely be modified to point to something else:

```
char *foo = "hello";
foo = "World!"; /* 好的——我们只是改变 foo 指向的内容 */
```

此外，虽然`char`数组的初始化器可以采用与字符串字面量相同的形式，但使用这样的初始化器并不会赋予被初始化数组字符串字面量的特性。初始化器仅仅指定了数组的长度和初始内容。特别是，如果没有显式声明为`const`，元素是可修改的：

```
char foo[] = "hello";
foo[0] = 'y'; /* 可以！ */
```

第6.3节：计算长度：`strlen()`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    /* 如果没有找到第二个参数则退出。*/
    if (argc != 2)
    {
        puts("缺少参数。");
        return EXIT_FAILURE;
    }

    size_t len = strlen(argv[1]);
    printf("第二个参数的长度是 %zu.\n", len);

    return EXIT_SUCCESS;
}
```

该程序计算其第二个输入参数的长度，并将结果存储在`len`中。然后它将该长度打印到终端。例如，当使用参数`program_name "Hello, world!"`运行时，程序将输出第二个参数的长度是13。因为字符串`Hello, world!`的长度是13个字符。

`strlen`计算从字符串开头到但不包括终止的NUL字符'\\0'的所有字节数。因此，它只能在字符串保证以NUL结尾时使用。

还要记住，如果字符串包含任何Unicode字符，`strlen`不会告诉你字符串中有多少个字符（因为有些字符可能是多字节的）。在这种情况下，你需要自己计算字符数（即代码单元）。请考虑以下示例的输出：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char asciiString[50] = "Hello world!";
    char utf8String[50] = "Γειά σου Κόσμε!"; /* 希腊语的"Hello World!" */

    printf("asciiString 在数组中有 %zu 字节\n", sizeof(asciiString));
    printf("utf8String 在数组中有 %zu 字节\n", sizeof(utf8String));
    printf("\\\"%s\\\" 是 %zu 字节\n", asciiString, strlen(asciiString));
    printf("\\\"%s\\\" 是 %zu 字节\n", utf8String, strlen(utf8String));
}
```

```
char *foo = "hello";
foo = "World!"; /* OK - we're just changing what foo points to */
```

Furthermore, although initializers for `char` arrays can have the same form as string literals, use of such an initializer does not confer the characteristics of a string literal on the initialized array. The initializer simply designates the length and initial contents of the array. In particular, the elements are modifiable if not explicitly declared `const`:

```
char foo[] = "hello";
foo[0] = 'y'; /* OK! */
```

Section 6.3: Calculate the Length: `strlen()`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        puts("Argument missing.");
        return EXIT_FAILURE;
    }

    size_t len = strlen(argv[1]);
    printf("The length of the second argument is %zu.\n", len);

    return EXIT_SUCCESS;
}
```

This program computes the length of its second input argument and stores the result in `len`. It then prints that length to the terminal. For example, when run with the parameters `program_name "Hello, world!"`, the program will output `The length of the second argument is 13.` because the string `Hello, world!` is 13 characters long.

`strlen` counts all the **bytes** from the beginning of the string up to, but not including, the terminating NUL character, '`\0`'. As such, it can only be used when the string is *guaranteed* to be NUL-terminated.

Also keep in mind that if the string contains any Unicode characters, `strlen` will not tell you how many characters are in the string (since some characters may be multiple bytes long). In such cases, you need to count the characters (*i.e.*, code units) yourself. Consider the output of the following example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char asciiString[50] = "Hello world!";
    char utf8String[50] = "Γειά σου Κόσμε!"; /* "Hello World!" in Greek */

    printf("asciiString has %zu bytes in the array\n", sizeof(asciiString));
    printf("utf8String has %zu bytes in the array\n", sizeof(utf8String));
    printf("\\\"%s\\\" is %zu bytes\n", asciiString, strlen(asciiString));
    printf("\\\"%s\\\" is %zu bytes\n", utf8String, strlen(utf8String));
}
```

输出：

```
asciiString 在数组中有 50 字节  
utf8String 在数组中有 50 字节  
"Hello world!" 是 12 字节  
"Γειά σου Κόσμε!" 是 27 字节
```

第6.4节：字符串基础介绍

在C语言中，字符串是一串以空字符 ('\0') 结尾的字符序列。

我们可以使用字符串字面量来创建字符串，字符串字面量是被双引号包围的字符序列；例如，字符串字面量"hello world"。字符串字面量会自动以空字符结尾。

我们可以使用多种方法创建字符串。例如，我们可以声明一个char*，并将其初始化为指向字符串的第一个字符：

```
char * string = "hello world";
```

当像上面这样将一个 char * 初始化为字符串常量时，字符串本身通常分配在只读数据区；string 是指向数组第一个元素的指针，即字符 'h'。

由于字符串字面量分配在只读内存中，因此它是不可修改的¹。任何试图修改它的行为都会导致未定义行为，所以最好添加 const 以获得类似的编译时错误

```
char const * string = "hello world";
```

它的效果类似²

```
char const string_arr[] = "hello world";
```

要创建一个可修改的字符串，可以声明一个字符数组并使用字符串字面量初始化其内容，如

下所示：

```
char modifiable_string[] = "hello world";
```

这等同于以下内容：

```
char modifiable_string[] = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
```

由于第二个版本使用了大括号括起来的初始化器，字符串不会自动以空字符结尾，除非字符数组中显式包含了'\0'字符，通常作为其最后一个元素。
\0字符通常作为字符数组的最后一个元素显式包含。

1 不可修改意味着字符串字面量中的字符不能被修改，但请记住指针string可以被修改（可以指向其他位置或可以递增或递减）。
string可以被修改（可以指向其他位置或可以递增或递减）。

2 两个字符串在某种意义上效果相似，即两个字符串的字符都不能被修改。需要注意的是
string是指向char的指针，并且它是一个可修改的左值，因此它可以递增或指向其他
位置，而数组string_arr是一个不可修改的左值，不能被修改。

第6.5节：复制字符串

指针赋值不会复制字符串

Output:

```
asciiString has 50 bytes in the array  
utf8String has 50 bytes in the array  
"Hello world!" is 12 bytes  
"Γειά σου Κόσμε!" is 27 bytes
```

Section 6.4: Basic introduction to strings

In C, a **string** is a sequence of characters that is terminated by a null character ('\0').

We can create strings using **string literals**, which are sequences of characters surrounded by double quotation marks; for example, take the string literal "hello world". String literals are automatically null-terminated.

We can create strings using several methods. For instance, we can declare a **char *** and initialize it to point to the first character of a string:

```
char * string = "hello world";
```

When initializing a **char *** to a string constant as above, the string itself is usually allocated in read-only data; string is a pointer to the first element of the array, which is the character 'h'.

Since the string literal is allocated in read-only memory, it is non-modifiable¹. Any attempt to modify it will lead to undefined behaviour, so it's better to add **const** to get a compile-time error like this

```
char const * string = "hello world";
```

It has similar effect² as

```
char const string_arr[] = "hello world";
```

To create a modifiable string, you can declare a character array and initialize its contents using a string literal, like so:

```
char modifiable_string[] = "hello world";
```

This is equivalent to the following:

```
char modifiable_string[] = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
```

Since the second version uses brace-enclosed initializer, the string is not automatically null-terminated unless a '\0' character is included explicitly in the character array usually as its last element.

1 Non-modifiable implies that the characters in the string literal can't be modified, but remember that the pointer string can be modified (can point somewhere else or can be incremented or decremented).

2 Both strings have similar effect in a sense that characters of both strings can't be modified. It should be noted that string is a pointer to **char** and it is a **modifiable l-value** so it can be incremented or point to some other location while the array string_arr is a non-modifiable l-value, it can't be modified.

Section 6.5: Copying strings

Pointer assignments do not copy strings

你可以使用=运算符复制整数，但在C语言中不能使用=运算符复制字符串。C语言中的字符串表示为以空字符结尾的字符数组，因此使用=运算符只会保存字符串的地址（指针）。

```
#include <stdio.h>

int main(void) {
    int a = 10, b;
    char c[] = "abc", *d;

    b = a; /* 整数被复制 */
    a = 20; /* 修改 a 不会改变 b —— b 是 a 的“深拷贝” */printf("%d %d", a, b); /* 将打印 "20 10" */

    d = c;
    /* 仅复制字符串的地址 ——
       内存中仍然只有一个字符串 */

    c[1] = 'x';
    /* 修改了原始字符串 —— d[1] = 'x' 也会做完全相同的事情 */

    printf("%s %s", c, d); /* 将打印 "axc axc" */return 0;
}
```

上述示例能够编译是因为我们使用了 `char *d` 而不是 `char d[3]`。使用后者会导致编译错误。在 C 语言中不能给数组赋值。

```
#include <stdio.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    b = a; /* 编译错误 */printf("%s", b);

    return 0;
}
```

使用标准函数复制字符串

`strcpy()`

要实际复制字符串，可以使用`strcpy()`函数，该函数定义在`string.h`中。复制前必须为目标分配足够的空间。

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    strcpy(b, a); /* 可以理解为“b 特殊地等于 a” */printf("%s", b); /* 将打印"abc" */return 0;
}

版本 ≥ C99
```

You can use the = operator to copy integers, but you cannot use the = operator to copy strings in C. Strings in C are represented as arrays of characters with a terminating null-character, so using the = operator will only save the address (pointer) of a string.

```
#include <stdio.h>

int main(void) {
    int a = 10, b;
    char c[] = "abc", *d;

    b = a; /* Integer is copied */
    a = 20; /* Modifying a leaves b unchanged - b is a 'deep copy' of a */
    printf("%d %d\n", a, b); /* "20 10" will be printed */

    d = c;
    /* Only copies the address of the string -
       there is still only one string stored in memory */

    c[1] = 'x';
    /* Modifies the original string - d[1] = 'x' will do exactly the same thing */

    printf("%s %s\n", c, d); /* "axc axc" will be printed */

    return 0;
}
```

The above example compiled because we used `char *d` rather than `char d[3]`. Using the latter would cause a compiler error. You cannot assign to arrays in C.

```
#include <stdio.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    b = a; /* compile error */
    printf("%s\n", b);

    return 0;
}
```

Copying strings using standard functions

`strcpy()`

To actually copy strings, `strcpy()` function is available in `string.h`. Enough space must be allocated for the destination before copying.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    strcpy(b, a); /* think "b special equals a" */
    printf("%s\n", b); /* "abc" will be printed */

    return 0;
}

Version ≥ C99
```

snprintf()

为了避免缓冲区溢出，可以使用 `snprintf()`。虽然它的性能不是最佳，因为需要解析模板字符串，但它是标准库中唯一可用的、无需额外步骤即可安全限制缓冲区大小的字符串复制函数。

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "012345678901234567890";
    char b[8];

#if 0
    strcpy(b, a); /* 导致缓冲区溢出（未定义行为），因此此处不要执行！ */
#endif

    snprintf(b, sizeof(b), "%s", a); /* 不会导致缓冲区溢出 */ printf("%s", b); /* 将打印 "0
123456" */

    return 0;
}
```

strncat()

第二种选择，性能更好，是使用 `strncat()` (`strcat()` 的缓冲区溢出检查版本) —— 它接受第三个参数，告诉它最大复制字节数：

```
char dest[32];

dest[0] = '\0';
strncat(dest, source, sizeof(dest) - 1);
/* 复制 source 的前 (sizeof(dest) - 1) 个元素到 dest,
然后在 dest 末尾添加一个 \0 */
```

注意这里使用了 `sizeof(dest) - 1`；这很关键，因为 `strncat()` 总是添加一个空字节（这是好的），但它不将该空字节计入字符串长度（这容易引起混淆和缓冲区覆盖）。

还要注意，另一种情况——在非空字符串后进行连接——更容易出问题。考虑：

```
char dst[24] = "小丑鱼: ";
char src[] = "马文和尼莫";
size_t len = strlen(dst);

strncat(dst, src, sizeof(dst) - len - 1); printf("%zu: [%s]\n", strlen(dst), dst);
```

输出是：

```
23: [小丑鱼: 马文和尼]
```

不过请注意，指定的长度不是目标数组的大小，而是它剩余的空间，不包括终止的空字节。这可能导致严重的覆盖问题。它也有点浪费；为了正确指定长度参数，你需要知道目标中数据的长度，因此你可以改为指定现有内容末尾空字节的地址，从而避免 `strncat()` 重新扫描它：

```
strcpy(dst, "小丑鱼: ");
```

snprintf()

To avoid buffer overrun, `snprintf()` may be used. It is not the best solution performance-wise since it has to parse the template string, but it is the only buffer limit-safe function for copying strings readily-available in standard library, that can be used without any extra steps.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "012345678901234567890";
    char b[8];

#if 0
    strcpy(b, a); /* causes buffer overrun (undefined behavior), so do not execute this here! */
#endif

    snprintf(b, sizeof(b), "%s", a); /* does not cause buffer overrun */
    printf("%s\n", b); /* "0123456" will be printed */

    return 0;
}
```

strncat()

A second option, with better performance, is to use `strncat()` (a buffer overflow checking version of `strcat()`) - it takes a third argument that tells it the maximum number of bytes to copy:

```
char dest[32];

dest[0] = '\0';
strncat(dest, source, sizeof(dest) - 1);
/* copies up to the first (sizeof(dest) - 1) elements of source into dest,
then puts a \0 on the end of dest */
```

Note that this formulation use `sizeof(dest) - 1`; this is crucial because `strncat()` always adds a null byte (good), but doesn't count that in the size of the string (a cause of confusion and buffer overwrites).

Also note that the alternative — concatenating after a non-empty string — is even more fraught. Consider:

```
char dst[24] = "Clownfish: ";
char src[] = "Marvin and Nemo";
size_t len = strlen(dst);

strncat(dst, src, sizeof(dst) - len - 1);
printf("%zu: [%s]\n", strlen(dst), dst);
```

The output is:

```
23: [Clownfish: Marvin and N]
```

Note, though, that the size specified as the length was *not* the size of the destination array, but the amount of space left in it, not counting the terminal null byte. This can cause big overwriting problems. It is also a bit wasteful; to specify the length argument correctly, you know the length of the data in the destination, so you could instead specify the address of the null byte at the end of the existing content, saving `strncat()` from rescanning it:

```
strcpy(dst, "Clownfish: ");
```

```
assert(len < sizeof(dst) - 1);strncat(d
st + len, src, sizeof(dst) - len - 1);printf("%zu: [%s]", strle
n(dst), dst);
```

这会产生与之前相同的输出，但`strncat()`不必在开始复制之前扫描 `dst` 的现有内容。

`strncpy()`

最后一个选项是`strncpy()`函数。虽然你可能认为它应该排在第一位，但它是一个相当具有迷惑性的函数，有两个主要的陷阱：

1. 如果通过`strncpy()`复制时达到缓冲区限制，将不会写入终止的空字符。
2. `strncpy()` 总是完全填充目标缓冲区，必要时用空字节填充。

(这种古怪的实现是历史遗留的，最初是为了处理 [UNIX 文件名](#)而设计的)

使用它的唯一正确方法是手动确保字符串以空字符结尾：

```
strncpy(b, a, sizeof(b)); /* 第三个参数是目标缓冲区大小 */
b[sizeof(b)/sizeof(*b) - 1] = '\0'; /* 终止字符串 */
printf("%s", b); /* 将打印 "0123456" */
```

即便如此，如果缓冲区很大，使用`strncpy()`仍然非常低效，因为会有额外的空字符填充。

第6.6节：遍历字符串中的字符

如果我们知道字符串的长度，可以使用 `for` 循环遍历其字符：

```
char * string = "hello world"; /* 该字符串长度为11，不包括结尾的0字符。 */
size_t i = 0;
for (; i < 11; i++) {
    printf("%c", string[i]); /* 打印字符串的每个字符。 */}
```

或者，如果我们不知道字符串是什么，可以使用标准函数 `strlen()` 来获取字符串的长度：

```
size_t length = strlen(string);
size_t i = 0;
for (; i < length; i++) {
    printf("%c", string[i]); /* 打印字符串的每个字符。 */}
```

最后，我们可以利用 C 语言中字符串保证以空字符结尾的特性（这在之前传递给 `strlen()` 时已经用过了 ;-））。我们可以不管数组大小，直接遍历，直到遇到空字符停止：

```
size_t i = 0;
while (string[i] != '\0') { /* 遇到空字符时停止循环。 */
    printf("%c", string[i]); /* 打印字符串的每个字符。 */
    i++;
}
```

```
assert(len < sizeof(dst) - 1);
strncat(dst + len, src, sizeof(dst) - len - 1);
printf("%zu: [%s]\n", strlen(dst), dst);
```

This produces the same output as before, but `strncat()` doesn't have to scan over the existing content of `dst` before it starts copying.

`strncpy()`

The last option is the `strncpy()` function. Although you might think it should come first, it is a rather deceptive function that has two main gotchas:

1. If copying via `strncpy()` hits the buffer limit, a terminating null-character won't be written.
2. `strncpy()` always completely fills the destination, with null bytes if necessary.

(Such quirky implementation is historical and [was initially intended for handling UNIX file names](#))

The only correct way to use it is to manually ensure null-termination:

```
strncpy(b, a, sizeof(b)); /* the third parameter is destination buffer size */
b[sizeof(b)/sizeof(*b) - 1] = '\0'; /* terminate the string */
printf("%s\n", b); /* "0123456" will be printed */
```

Even then, if you have a big buffer it becomes very inefficient to use `strncpy()` because of additional null padding.

Section 6.6: Iterating Over the Characters in a String

If we know the length of the string, we can use a `for` loop to iterate over its characters:

```
char * string = "hello world"; /* This 11 chars long, excluding the 0-terminator. */
size_t i = 0;
for (; i < 11; i++) {
    printf("%c\n", string[i]); /* Print each character of the string. */}
```

Alternatively, we can use the standard function `strlen()` to get the length of a string if we don't know what the string is:

```
size_t length = strlen(string);
size_t i = 0;
for (; i < length; i++) {
    printf("%c\n", string[i]); /* Print each character of the string. */}
```

Finally, we can take advantage of the fact that strings in C are guaranteed to be null-terminated (which we already did when passing it to `strlen()` in the previous example ;-)). We can iterate over the array regardless of its size and stop iterating once we reach a null-character:

```
size_t i = 0;
while (string[i] != '\0') { /* Stop looping when we reach the null-character. */
    printf("%c\n", string[i]); /* Print each character of the string. */
    i++;
}
```

第6.7节：创建字符串数组

字符串数组可以有几种含义：

1. 一个元素为char *的数组
2. 一个元素为char数组的数组

我们可以这样创建一个字符指针数组：

```
char * string_array[] = {  
    "foo",  
    "bar",  
    "baz"  
};
```

请记住：当我们将字符串字面量赋值给char *时，字符串本身被分配在只读内存中。

然而，数组string_array被分配在可读写内存中。这意味着我们可以修改数组中的指针，但不能修改它们所指向的字符串。

在C语言中，main函数的参数argv（程序运行时传入的命令行参数数组）是一个char *数组：char * argv[]。

我们也可以创建字符串数组的数组。由于字符串是字符数组，字符串数组就是元素为字符数组的数组：

```
char modifiable_string_array_literals[][4] = {  
    "foo",  
    "bar",  
    "baz"  
};
```

这相当于：

```
char 可修改字符串数组[][4] = {  
    {'f', 'o', 'o', '\0'},  
    {'b', 'a', 'r', '\0'},  
    {'b', 'a', 'z', '\0'}  
};
```

注意我们指定了4作为数组第二维的大小；数组中的每个字符串实际上是4字节，因为必须包含空字符终止符。

第6.8节：字符串转换为数字：atoi(), atof() (危险，不要使用它们)

警告：函数atoi、atol、atoll和atof本质上是不安全的，因为：如果结果的值无法表示，行为是未定义的。(7.20.1p1)

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(int argc, char** argv)  
{  
    int val;  
    if (argc < 2)  
    {
```

Section 6.7: Creating Arrays of Strings

An array of strings can mean a couple of things:

1. An array whose elements are char *
2. An array whose elements are arrays of chars

We can create an array of character pointers like so:

```
char * string_array[] = {  
    "foo",  
    "bar",  
    "baz"  
};
```

Remember: when we assign string literals to char *, the strings themselves are allocated in read-only memory. However, the array string_array is allocated in read/write memory. This means that we can modify the pointers in the array, but we cannot modify the strings they point to.

In C, the parameter to main argv (the array of command-line arguments passed when the program was run) is an array of char *: char * argv[].

We can also create arrays of character arrays. Since strings are arrays of characters, an array of strings is simply an array whose elements are arrays of characters:

```
char modifiable_string_array_literals[][4] = {  
    "foo",  
    "bar",  
    "baz"  
};
```

This is equivalent to:

```
char modifiable_string_array[][4] = {  
    {'f', 'o', 'o', '\0'},  
    {'b', 'a', 'r', '\0'},  
    {'b', 'a', 'z', '\0'}  
};
```

Note that we specify 4 as the size of the second dimension of the array; each of the strings in our array is actually 4 bytes since we must include the null-terminating character.

Section 6.8: Convert Strings to Number: atoi(), atof() (dangerous, don't use them)

Warning: The functions atoi, atol, atoll and atof are inherently unsafe, because: If the value of the result cannot be represented, the behavior is undefined. (7.20.1p1)

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(int argc, char** argv)  
{  
    int val;  
    if (argc < 2)  
    {
```

```

    printf("用法 : %s <整数>", argv[0]);return 0;
}

val = atoi(argv[1]);

printf("字符串值 = %s, 整数值 = %d", argv[1], val);return 0;

}

```

当要转换的字符串是有效的十进制整数且在范围内时，函数正常工作：

```

$ ./atoi 100
字符串值 = 100, 整数值 = 100
$ ./atoi 200
字符串值 = 200, 整数值 = 200

```

对于以数字开头，后面跟有其他字符的字符串，只解析开头的数字部分：

```

$ ./atoi 0x200
0
$ ./atoi 0123x300
123

```

在所有其他情况下，行为未定义：

```

$ ./atoi hello
格式化硬盘中...

```

由于上述歧义和未定义行为，atoi系列函数绝不应被使用。

- 要转换为long int，使用strtol()代替atoi()。
- 要转换为double，使用strtod()代替atof()。

版本 ≥ C99

- 要转换为long long int，使用strtoll()代替atoll()。

第6.9节：字符串格式化数据读写

将格式化数据写入字符串

```
int sprintf ( char * str, const char * format, ... );
```

使用sprintf函数将浮点数据写入字符串。

```

#include <stdio.h>
int main ()
{
    char buffer [50];
    double PI = 3.1415926;spri
    ntf (buffer, "PI = %.7f", PI);printf ("%s",b
    uffer);
    return 0;
}

```

```

    printf("Usage: %s <integer>\n", argv[0]);
    return 0;
}

val = atoi(argv[1]);

printf("String value = %s, Int value = %d\n", argv[1], val);

return 0;
}

```

When the string to be converted is a valid decimal integer that is in range, the function works:

```

$ ./atoi 100
String value = 100, Int value = 100
$ ./atoi 200
String value = 200, Int value = 200

```

For strings that start with a number, followed by something else, only the initial number is parsed:

```

$ ./atoi 0x200
0
$ ./atoi 0123x300
123

```

In all other cases, the behavior is undefined:

```

$ ./atoi hello
Formatting the hard disk...

```

Because of the ambiguities above and this undefined behavior, the `atoi` family of functions should never be used.

- To convert to `long int`, use `strtol()` instead of `atoi()`.
- To convert to `double`, use `strtod()` instead of `atof()`.

Version ≥ C99

- To convert to `long long int`, use `strtoll()` instead of `atoll()`.

Section 6.9: string formatted data read/write

Write formatted data to string

```
int sprintf ( char * str, const char * format, ... );
```

use `sprintf` function to write float data to string.

```

#include <stdio.h>
int main ()
{
    char buffer [50];
    double PI = 3.1415926;
    sprintf (buffer, "PI = %.7f", PI);
    printf ("%s\n",buffer);
    return 0;
}

```

```
int sscanf ( const char * s, const char * format, ...);
```

使用sscanf函数解析格式化数据。

```
#include <stdio.h>
int main ()
{
    char sentence []="date : 06-06-2012";
    char str [50];
    int year;
    int month;
    int day;
    sscanf (sentence,"%s : %2d-%2d-%4d", str, &day, &month, &year);printf ("%s
-> %02d-%02d-%4d",str, day, month, year);
    return 0;
}
```

第6.10节：查找特定字符的第一次/最后一次出现： strchr(), strrchr()

函数strchr和strrchr用于在字符串中查找字符，即在以NUL结尾的字符数组中查找。函数strchr返回第一个匹配字符的指针，函数strrchr返回最后一个匹配字符的指针。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char toSearchFor = 'A';

    /* 如果没有找到第二个参数则退出。 */
    if (argc != 2)
    {
        printf("参数缺失。");return EXIT_FAILURE;
    }

    {
        char *firstOcc = strchr(argv[1], toSearchFor);
        if (firstOcc != NULL)
        {
            printf("%c 在 %s 中的第一个位置是 %td。",
                toSearchFor, argv[1], firstOcc-argv[1]); /* 指针差的结果
                , 使用长度修饰符 't'。 */
        }
        else
        {
            printf("%c 不在 %s 中。", toSearchFor, argv[1]);
        }
    }

    {
        char *lastOcc = strrchr(argv[1], toSearchFor);
        if (lastOcc != NULL)
        {
            printf("%c 在 %s 中的最后一个位置是 %td。",
                toSearchFor, argv[1], lastOcc-argv[1]);
        }
    }
}
```

是有符号整数

```
int sscanf ( const char * s, const char * format, ...);
```

use sscanf function to parse formatted data.

```
#include <stdio.h>
int main ()
{
    char sentence []="date : 06-06-2012";
    char str [50];
    int year;
    int month;
    int day;
    sscanf (sentence,"%s : %2d-%2d-%4d", str, &day, &month, &year);
    printf ("%s -> %02d-%02d-%4d\n",str, day, month, year);
    return 0;
}
```

Section 6.10: Find first/last occurrence of a specific character: strchr(), strrchr()

The `strchr` and `strrchr` functions find a character in a string, that is in a NUL-terminated character array. `strchr` return a pointer to the first occurrence and `strrchr` to the last one.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char toSearchFor = 'A';

    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        printf("Argument missing.\n");
        return EXIT_FAILURE;
    }

    {
        char *firstOcc = strchr(argv[1], toSearchFor);
        if (firstOcc != NULL)
        {
            printf("First position of %c in %s is %td.\n",
                toSearchFor, argv[1], firstOcc-argv[1]); /* A pointer difference's result
                is a signed integer and uses the length modifier 't'. */
        }
        else
        {
            printf("%c is not in %s.\n", toSearchFor, argv[1]);
        }
    }

    {
        char *lastOcc = strrchr(argv[1], toSearchFor);
        if (lastOcc != NULL)
        {
            printf("Last position of %c in %s is %td.\n",
                toSearchFor, argv[1], lastOcc-argv[1]);
        }
    }
}
```

```

    toSearchFor, argv[1], lastOcc-argv[1]);
}
}

return EXIT_SUCCESS;
}

```

输出（生成名为pos的可执行文件后）：

```

$ ./pos AAAAAAA
AAAAAAA中A的第一个位置是0。
AAAAAAA中A的最后一个位置是6。
$ ./pos BAbbbbBAcccAAAzzz
BAbbbbBAcccAAAzzz中A的第一个位置是1。
BAbbbbBAcccAAAzzz中A的最后一个位置是15。
$ ./pos qwerty
qwerty中没有A。

```

一个常见的`strrchr`用法是从路径中提取文件名。例如，从

C:\Users\eak\myfile.txt中提取myfile.txt：

```

char *getFileName(const char *path)
{
    char *pend;

    if ((pend = strrchr(path, '\\')) != NULL)
        return pend + 1;

    return NULL;
}

```

第6.11节：复制与连接：`strcpy()`, `strcat()`

```

#include <stdio.h>
#include <string.h>

int main(void)
{
    /* 始终确保你的字符串足够大，可以容纳字符
     * 以及终止的NUL字符 ('\0') !
     */
    char mystring[10];

    /* 将 "foo" 复制到 `mystring` 中，直到遇到NUL字符。*/
    strcpy(mystring, "foo");printf(
        "%s", mystring);

    /* 此时，我们使用了 `mystring` 的4个字符，3个 "foo" 字符，
     * 以及NUL终止字节。
     */

    /* 将 "bar" 追加到 `mystring`。*/
    strcat(mystring, "bar");printf(
        "%s", mystring);

    /* 我们现在使用 `mystring` 的7个字符："foo"需要3个，"bar"需要3个
     * 并且未尾有一个终止的空字符 ('\0') 。
     */
}

```

```

    toSearchFor, argv[1], lastOcc-argv[1]);
}
}

return EXIT_SUCCESS;
}

```

Outputs (after having generate an executable named pos):

```

$ ./pos AAAAAAA
First position of A in AAAAAAA is 0.
Last position of A in AAAAAAA is 6.
$ ./pos BAbbbbBAcccAAAzzz
First position of A in BAbbbbBAcccAAAzzz is 1.
Last position of A in BAbbbbBAcccAAAzzz is 15.
$ ./pos qwerty
A is not in qwerty.

```

One common use for `strrchr` is to extract a file name from a path. For example to extract `myfile.txt` from C:\Users\leak\myfile.txt:

```

char *getFileName(const char *path)
{
    char *pend;

    if ((pend = strrchr(path, '\\')) != NULL)
        return pend + 1;

    return NULL;
}

```

Section 6.11: Copy and Concatenation: `strcpy()`, `strcat()`

```

#include <stdio.h>
#include <string.h>

int main(void)
{
    /* Always ensure that your string is large enough to contain the characters
     * and a terminating NUL character ('\0')!
     */
    char mystring[10];

    /* Copy "foo" into `mystring` , until a NUL character is encountered. */
    strcpy(mystring, "foo");
    printf("%s\n", mystring);

    /* At this point, we used 4 chars of `mystring` , the 3 characters of "foo",
     * and the NUL terminating byte.
     */

    /* Append "bar" to `mystring` . */
    strcat(mystring, "bar");
    printf("%s\n", mystring);

    /* We now use 7 characters of `mystring` : "foo" requires 3, "bar" requires 3
     * and there is a terminating NUL character ('\0') at the end.
     */
}

```

```
/* 将 "bar" 复制到 `mystring`，覆盖原有内容。 */
strcpy(mystring, "bar");printf(
"%s", mystring);return 0;

}
```

输出：

```
foo
foobar
bar
```

如果你追加到、来自或复制自己有字符串，确保它是以 NUL 结尾的！

字符串字面量（例如 "foo"）总是由编译器以 NUL 结尾。

第6.12节：比较：strcmp(), strncmp(), strcasecmp(), strncasecmp()

strcasecmp*函数不是标准C，而是POSIX扩展。

strcmp函数按字典序比较两个以空字符结尾的字符数组。如果第一个参数在字典序中排在第二个之前，函数返回负值；如果两者相等，返回零；如果第一个参数在第二个之后，返回正值。

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcmp(lhs, rhs); // 只计算一次比较结果
    if (result < 0) {
        printf("%s 在 %s 之前", lhs, rhs);} else if (result =
= 0) {printf("%s 等于 %s", lhs,
rhs);} else { // 最后一种情况：result > 0
printf("%s 在 %s 之后", lhs, rhs);}
}

int main(void)
{
compare("BBB", "BBB");
compare("BBB", "CCCCC");
compare("BBB", "AAAAAA");
return 0;
}
```

输出：

```
BBB 等于 BBB
BBB 在 CCCCC 之前
BBB 在 AAAAAA 之后
```

与 strcmp 函数类似，strcasecmp 函数也会在比较其参数时，将每个字符转换为对应的小写字母后进行字典序比较：

```
/* Copy "bar" into `mystring`，overwriting the former contents. */
strcpy(mystring, "bar");
printf("%s\n", mystring);

return 0;
}
```

Outputs:

```
foo
foobar
bar
```

If you append to or from or copy from an existing string, ensure it is NUL-terminated!

String literals (e.g. "foo") will always be NUL-terminated by the compiler.

Section 6.12: Comparsion: strcmp(), strncmp(), strcasecmp(), strncasecmp()

The strcasecmp*-functions are not Standard C, but a POSIX extension.

The **strcmp** function lexicographically compare two null-terminated character arrays. The functions return a negative value if the first argument appears before the second in lexicographical order, zero if they compare equal, or positive if the first argument appears after the second in lexicographical order.

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcmp(lhs, rhs); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
compare("BBB", "BBB");
compare("BBB", "CCCCC");
compare("BBB", "AAAAAA");
return 0;
}
```

Outputs:

```
BBB equals BBB
BBB comes before CCCCC
BBB comes after AAAAAA
```

As **strcmp**, **strcasecmp** function also compares lexicographically its arguments after translating each character to its lowercase correspondent:

```

#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcasecmp(lhs, rhs); // 只计算一次不区分大小写的比较结果
    if (result < 0) {
        printf("%s 在 %s 之前", lhs, rhs);} else if (result =
= 0) {printf("%s 等于 %s", lhs,
rhs);} else { // 最后一种情况：result > 0
intf("%s 在 %s 之后", lhs, rhs);}
}

int main(void)
{
compare("BBB", "bBB");
compare("BBB", "ccCCC");
compare("BBB", "aaaaaa");
return 0;
}

```

输出：

```

BBB 等于 bBB
BBB 在 ccCCC 之前
BBB 在 aaaaaa 之后

```

strcmp 和 strncasecmp 函数最多比较 n 个字符：

```

#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs, int n)
{
    int result = strncasecmp(lhs, rhs, n); // 只计算一次比较结果
    if (result < 0) {
        printf("%s 在 %s 之前", lhs, rhs);} else if (result =
= 0) {printf("%s 等于 %s", lhs,
rhs);} else { // 最后一种情况：result > 0
intf("%s 在 %s 之后", lhs, rhs);}
}

int main(void)
{
compare("BBB", "Bb", 1);
compare("BBB", "Bb", 2);
compare("BBB", "Bb", 3);
return 0;
}

```

输出：

```

BBB 等于 Bb
BBB 排在 Bb 之前
BBB 排在 Bb 之前

```

```

#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcasecmp(lhs, rhs); // compute case-insensitive comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "bBB");
    compare("BBB", "ccCCC");
    compare("BBB", "aaaaaa");
    return 0;
}

```

Outputs:

```

BBB equals bBB
BBB comes before ccCCC
BBB comes after aaaaaa

```

strcmp 和 strncasecmp compare at most n characters:

```

#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs, int n)
{
    int result = strncasecmp(lhs, rhs, n); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "Bb", 1);
    compare("BBB", "Bb", 2);
    compare("BBB", "Bb", 3);
    return 0;
}

```

Outputs:

```

BBB equals Bb
BBB comes before Bb
BBB comes before Bb

```

第6.13节：安全地将字符串转换为数字：strtoX 函数

版本 ≥ C99

自 C99 起，C 库提供了一组安全的转换函数，用于将字符串解释为数字。它们的名称形式为 strtoX，其中 X 是 l、ul、d 等，用于确定转换的目标类型

```
double strtod(char const* p, char** endptr);  
long double strtold(char const* p, char** endptr);
```

它们提供了检查转换是否发生溢出或下溢的功能：

```
double ret = strtod(argv[1], 0); /* 尝试转换 */  
  
/* 检查转换结果。 */  
if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)  
    return; /* 字符串中的数值溢出 */  
else if (ret == HUGE_VAL && errno == ERANGE)  
    return; /* 字符串中的数值下溢 */  
  
/* 此时我们知道一切正常，可以使用 ret */
```

如果字符串实际上根本不包含数字，则此处使用的strtod返回0.0。

如果这不令人满意，可以使用额外的参数endptr。它是一个指向指针的指针，将指向字符串中检测到的数字的结尾。如果设置为0（如上所述）或NULL，则会被忽略。

该endptr参数用于指示是否成功转换，如果成功，数字结束的位置是哪里：

```
char *check = 0;  
double ret = strtod(argv[1], &check); /* 尝试转换 */  
  
/* 检查转换结果。 */  
if (argv[1] == check)  
    return; /* 字符串中未检测到数字 */  
else if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)  
    return; /* 字符串中数字溢出 */  
else if (ret == HUGE_VAL && errno == ERANGE)  
    return; /* 字符串中的数值下溢 */  
  
/* 此时我们知道一切正常，可以使用 ret */
```

有类似的函数用于转换为更宽的整数类型：

```
long strtol(char const* p, char** endptr, int nbase);  
long long strtoll(char const* p, char** endptr, int nbase);  
unsigned long strtoul(char const* p, char** endptr, int nbase);  
unsigned long long strtoull(char const* p, char** endptr, int nbase);
```

这些函数有第三个参数 nbase，表示数字所使用的进制。

```
long a = strtol("101", 0, 2); /* a = 5L */  
long b = strtol("101", 0, 8); /* b = 65L */  
long c = strtol("101", 0, 10); /* c = 101L */  
long d = strtol("101", 0, 16); /* d = 257L */  
long e = strtol("101", 0, 0); /* e = 101L */
```

Section 6.13: Safely convert Strings to Number: strtoX functions

Version ≥ C99

Since C99 the C library has a set of safe conversion functions that interpret a string as a number. Their names are of the form strtoX, where X is one of l, ul, d, etc to determine the target type of the conversion

```
double strtod(char const* p, char** endptr);  
long double strtold(char const* p, char** endptr);
```

They provide checking that a conversion had an over- or underflow:

```
double ret = strtod(argv[1], 0); /* attempt conversion */  
  
/* check the conversion result. */  
if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)  
    return; /* numeric overflow in in string */  
else if (ret == HUGE_VAL && errno == ERANGE)  
    return; /* numeric underflow in in string */  
  
/* At this point we know that everything went fine so ret may be used */
```

If the string in fact contains no number at all, this usage of strtod returns 0.0.

If this is not satisfactory, the additional parameter endptr can be used. It is a pointer to pointer that will be pointed to the end of the detected number in the string. If it is set to 0, as above, or NULL, it is simply ignored.

This endptr parameter provides indicates if there has been a successful conversion and if so, where the number ended:

```
char *check = 0;  
double ret = strtod(argv[1], &check); /* attempt conversion */  
  
/* check the conversion result. */  
if (argv[1] == check)  
    return; /* No number was detected in string */  
else if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)  
    return; /* numeric overflow in in string */  
else if (ret == HUGE_VAL && errno == ERANGE)  
    return; /* numeric underflow in in string */  
  
/* At this point we know that everything went fine so ret may be used */
```

There are analogous functions to convert to the wider integer types:

```
long strtol(char const* p, char** endptr, int nbase);  
long long strtoll(char const* p, char** endptr, int nbase);  
unsigned long strtoul(char const* p, char** endptr, int nbase);  
unsigned long long strtoull(char const* p, char** endptr, int nbase);
```

These functions have a third parameter nbase that holds the number base in which the number is written.

```
long a = strtol("101", 0, 2); /* a = 5L */  
long b = strtol("101", 0, 8); /* b = 65L */  
long c = strtol("101", 0, 10); /* c = 101L */  
long d = strtol("101", 0, 16); /* d = 257L */  
long e = strtol("101", 0, 0); /* e = 101L */
```

```
long f = strtol("0101", 0, 0); /* f = 65L */
long g = strtol("0x101", 0, 0); /* g = 257L */
```

特殊值0对于nbase意味着字符串的解释方式与C程序中数字字面量的解释方式相同：以0x开头表示十六进制，否则以0开头表示八进制，其他数字则视为十进制。

因此，将命令行参数解释为数字的最实用方法是

```
int main(int argc, char* argv[] {
    if (argc < 1)
        return EXIT_FAILURE; /* 未提供数字。 */

    /* 使用 strtoull 因为 size_t 可能较宽 */
    size_t mySize = strtoull(argv[1], 0, 0);

    /* 然后检查转换结果。 */
    ...

    return EXIT_SUCCESS;
}
```

这意味着程序可以用八进制、十进制或十六进制的参数调用。

第6.14节：strspn 和 strcspn

给定一个字符串，`strspn` 计算由特定字符列表组成的初始子串（跨度）的长度。`strcspn` 类似，但它计算由除所列字符之外的任意字符组成的初始子串的长度：

```
/*
给定一个由“分隔符”分隔的“标记”字符串，打印标记以及被跳过的标记分隔符。
*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    const char sepchars[] = ",.;!?";
    char foo[] = ";ball call,.fall gall hall!?.";
    char *s;
    int n;

    for (s = foo; *s != 0; /*空*/) {
        /* 获取标记分隔符字符的数量。 */
        n = (int)strspn(s, sepchars);

        if (n > 0)
            printf("跳过的分隔符:<< %.*s >> (长度=%d)", n, s, n);

        /* 现在实际跳过分隔符。 */
        s += n;

        /* 获取标记（非分隔符）字符的数量。 */
        n = (int)strcspn(s, sepchars);
    }
}
```

```
long f = strtol("0101", 0, 0); /* f = 65L */
long g = strtol("0x101", 0, 0); /* g = 257L */
```

The special value 0 for nbase means the string is interpreted in the same way as number literals are interpreted in a C program: a prefix of 0x corresponds to a hexadecimal representation, otherwise a leading 0 is octal and all other numbers are seen as decimal.

Thus the most practical way to interpret a command-line argument as a number would be

```
int main(int argc, char* argv[] {
    if (argc < 1)
        return EXIT_FAILURE; /* No number given. */

    /* use strtoull because size_t may be wide */
    size_t mySize = strtoull(argv[1], 0, 0);

    /* then check conversion results. */
    ...

    return EXIT_SUCCESS;
}
```

This means that the program can be called with a parameter in octal, decimal or hexadecimal.

Section 6.14: strspn and strcspn

Given a string, `strspn` calculates the length of the initial substring (span) consisting solely of a specific list of characters. `strcspn` is similar, except it calculates the length of the initial substring consisting of any characters except those listed:

```
/*
Provided a string of "tokens" delimited by "separators", print the tokens along
with the token separators that get skipped.
*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    const char sepchars[] = ",.;!?";
    char foo[] = ";ball call,.fall gall hall!?.";
    char *s;
    int n;

    for (s = foo; *s != 0; /*empty*/) {
        /* Get the number of token separator characters. */
        n = (int)strspn(s, sepchars);

        if (n > 0)
            printf("skipping separators: << %.*s >> (length=%d)\n", n, s, n);

        /* Actually skip the separators now. */
        s += n;

        /* Get the number of token (non-separator) characters. */
        n = (int)strcspn(s, sepchars);
    }
}
```

```
if (n > 0)
    printf("找到标记: << %.*s >> (长度=%d)", n, s, n);

    /* 现在跳过该标记。 */
s += n;
}

printf("== 标记列表已耗尽 ==");return 0;
```

使用宽字符字符串的类似函数是 `wcsspn` 和 `wcscspn`；它们的用法相同。

```
if (n > 0)
    printf("token found: << %.*s >> (length=%d)\n", n, s, n);

    /* Skip the token now. */
s += n;
}

printf("== token list exhausted ==\n");

return 0;
}
```

Analogous functions using wide-character strings are `wcsspn` and `wcscspn`; they're used the same way.

第7章：数字、字符和字符串的字面量

第7.1节：浮点字面量

浮点字面量用于表示带符号的实数。可以使用以下后缀来指定字面量的类型：

后缀	类型	示例
无双精度	double	3.1415926 -3E6
f, F	单精度浮点数	3.1415926f 2.1E-6F
l, L	长双精度浮点数	3.1415926L 1E126L

为了使用这些后缀，字面量必须是浮点字面量。例如，3f是错误的，因为3是整数字面量，而3.f或3.0f是正确的。对于长双精度浮点数，建议始终使用大写L以提高可读性。

第7.2节：字符串字面量

字符串字面量用于指定字符数组。它们是被双引号括起来的字符序列（例如"abcd"），类型为char*。

前缀L使字面量成为宽字符数组，类型为wchar_t*。例如，L"abcd"。

自C11起，还有其他类似于L的编码前缀：

前缀	基础类型	编码
无char	依赖平台	
L	wchar_t	依赖平台
u8	char	UTF-8
u	char16_t	通常为 UTF-16
U	char32_t	通常为 UTF-32

对于后两者，可以通过特性测试宏查询编码是否实际上是对应的UTF编码。

第7.3节：字符字面量

字符字面量是一种特殊类型的整数字面量，用于表示单个字符。它们用单引号括起来，例如'a'，类型为int。字面量的值是根据机器字符集的整数值。它们不允许有后缀。

字符字面量前的L前缀使其成为类型为wchar_t的宽字符。同样，自C11起，u和U前缀分别使其成为类型为char16_t和char32_t的宽字符。

当表示某些特殊字符时，例如不可打印字符，使用转义序列。转义序列使用一串字符，转换为另一个字符。所有转义序列由两个或更多字符组成，第一个字符是反斜杠\。紧跟反斜杠的字符决定该序列被解释为何种字符字面量。

转义序列表示的字符

Chapter 7: Literals for numbers, characters and strings

Section 7.1: Floating point literals

Floating point literals are used to represent signed real numbers. The following suffixes can be used to specify type of a literal:

Suffix	Type	Examples
none	double	3.1415926 -3E6
f, F	float	3.1415926f 2.1E-6F
l, L	long double	3.1415926L 1E126L

In order to use these suffixes, the literal *must* be a floating point literal. For example, 3f is an error, since 3 is an integer literal, while 3.f or 3.0f are correct. For long double, the recommendation is to always use capital L for the sake of readability.

Section 7.2: String literals

String literals are used to specify arrays of characters. They are sequences of characters enclosed within double quotes (e.g. "abcd" and have the type char*).

The L prefix makes the literal a wide character array, of type wchar_t*. For example, L"abcd".

Since C11, there are other encoding prefixes, similar to L:

prefix	base type	encoding
none	char	platform dependent
L	wchar_t	platform dependent
u8	char	UTF-8
u	char16_t	usually UTF-16
U	char32_t	usually UTF-32

For the latter two, it can be queried with feature test macros if the encoding is effectively the corresponding UTF encoding.

Section 7.3: Character literals

Character literals are a special type of integer literals that are used to represent one character. They are enclosed in single quotes, e.g. 'a' and have the type int. The value of the literal is an integer value according to the machine's character set. They do not allow suffixes.

The L prefix before a character literal makes it a wide character of type wchar_t. Likewise since C11 u and U prefixes make it wide characters of type char16_t and char32_t, respectively.

When intending to represent certain special characters, such as a character that is non-printing, escape sequences are used. Escape sequences use a sequence of characters that are translated into another character. All escape sequences consist of two or more characters, the first of which is a backslash \. The characters immediately following the backslash determine what character literal the sequence is interpreted as.

Escape Sequence Represented Character

\b	退格键
\f	换页符
\n	换行符 (新行)
\r	回车符
\t	水平制表符
\v	垂直制表符
\\\	反斜杠
\'	单引号
\"	双引号
\?	问号
	八进制值
\xnn...	十六进制值

版本 ≥ C89

转义序列表示的字符

\a	警报 (蜂鸣声, 铃声)
----	--------------

版本 ≥ C99

转义序列表示的字符

\unnnn	通用字符名
--------	-------

\Unnnnnnnn	通用字符名
------------	-------

通用字符名是一个Unicode代码点。通用字符名可能映射到多个字符。数字 n 被解释为十六进制数字。根据所使用的UTF编码，通用字符名序列可能导致一个由多个字符组成的代码点，而不是单个普通 char 字符。

在文本模式输入输出中使用换行符转义序列时，它会被转换为操作系统特定的换行字节或字节序列。

问号转义序列用于避免三字符序列。例如，??/ 被编译为表示反斜杠字符 \' 的三字符序列，但使用 ?\?/ 会得到字符串 "??/"。

八进制值转义序列中可能包含一个、两个或三个八进制数字 n。

第7.4节：整数字面量

整数字面量用于提供整数值。支持三种数字进制，由前缀表示：

进制	前缀	示例
十进制	无	5
八进制	0	0345
十六进制	0x 或 0X	0x12AB, 0X12AB, 0x12ab, 0x12Ab

注意，这种写法不包含任何符号，因此整数字面量总是正数。类似 -1 的表达式被视为一个整数字面量 (1) 前面带有负号 -

十进制整数字面量的类型是从 int 和 long 中第一个能容纳该值的数据类型。自 C99 起，也支持用于非常大字面量的 long long。

八进制或十六进制整数字面量的类型是从 int、unsigned、long 和 unsigned long 中第一个能容纳该值的数据类型。自 C99 起，也支持用于非常大字面量的 long long 和 unsigned long long。

\b	Backspace
\f	Form feed
\n	Line feed (new line)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\nnn	Octal value
\xnn...	Hexadecimal value

Version ≥ C89

Escape Sequence Represented Character

\a	Alert (beep, bell)
----	--------------------

Version ≥ C99

Escape Sequence Represented Character

\unnnn	Universal character name
--------	--------------------------

\Unnnnnnnn	Universal character name
------------	--------------------------

A universal character name is a Unicode code point. A universal character name may map to more than one character. The digits n are interpreted as hexadecimal digits. Depending on the UTF encoding in use, a universal character name sequence may result in a code point that consists of multiple characters, instead of a single normal char character.

When using the line feed escape sequence in text mode I/O, it is converted to the OS-specific newline byte or byte sequence.

The question mark escape sequence is used to avoid trigraphs. For example, ??/ is compiled as the trigraph representing a backslash character \' , but using ?\?/ would result in the string "??/" .

There may be one, two or three octal numerals n in the octal value escape sequence.

Section 7.4: Integer literals

Integer literals are used to provide integral values. Three numerical bases are supported, indicated by prefixes:

Base	Prefix	Example
Decimal	None	5
Octal	0	0345
Hexadecimal	0x or 0X	0x12AB, 0X12AB, 0x12ab, 0x12Ab

Note that this writing doesn't include any sign, so integer literals are always positive. Something like -1 is treated as an expression that has one integer literal (1) that is negated with a -

The type of a decimal integer literal is the first data type that can fit the value from int and long. Since C99, long long is also supported for very large literals.

The type of an octal or hexadecimal integer literal is the first data type that can fit the value from int, unsigned, long, and unsigned long. Since C99, long long and unsigned long long are also supported for very large literals.

通过使用各种后缀，可以改变字面量的默认类型。

后缀	说明
L, l	长整型
LL, ll (自C99起)	long long int
U, u	unsigned

U和L/LL后缀可以以任意顺序和大小写组合。重复后缀（例如提供两个U后缀）即使大小写不同也是错误的。

Using various suffixes, the default type of a literal can be changed.

Suffix	Explanation
L, l	long int
LL, ll (since C99)	long long int
U, u	unsigned

The U and L/LL suffixes can be combined in any order and case. It is an error to duplicate suffixes (e.g. provide two U suffixes) even if they have different cases.

第8章：复合字面量

第8.1节：复合字面量的定义/初始化

复合字面量是一个未命名的对象，在定义的作用域内创建。该概念首次在C99标准中引入。复合字面量的一个例子是

C标准C11-§6.5.2.5/9中的示例：

```
int *p = (int [2]){ 2, 4 };
```

p被初始化为一个未命名的两个整数数组第一个元素的地址。

复合字面量是一个左值。未命名对象的存储期限要么是静态的（如果字面量出现在文件作用域），要么是自动的（如果字面量出现在块作用域），在后一种情况下，对象的生命周期在控制流离开包含它的块时结束。

```
void f(void)
{
    int *p;
    /*...*/
    p = (int [2]){ *p };
    /*...*/
}
```

p 被赋值为一个包含两个整数的数组的第一个元素的地址，第一个元素的值是之前指针 p 指向的值，第二个元素为零。[...]

这里 p 在代码块结束之前保持有效。

带designators的复合字面量

(同样来自 C11)

```
struct point {
    unsigned x;
    unsigned y;
};

extern void drawline(struct point, struct point);

// 用法示例
drawline((struct point){.x=1, .y=1}, (struct point){.x=3, .y=4});
```

一个虚构的函数 drawline接收两个类型为`struct point`的参数。第一个的坐标值为 `x == 1` 且 `y == 1`，第二个的坐标为 `x == 3`且 `y == 4`

未指定数组长度的复合字面量

```
int *p = (int []){ 1, 2, 3};
```

在这种情况下，数组的大小未指定，将由初始化器的长度决定。

初始化器长度小于指定数组大小的复合字面量

```
int *p = (int [10]){1, 2, 3};
```

Chapter 8: Compound Literals

Section 8.1: Definition/Initialisation of Compound Literals

A compound literal is an unnamed object which is created in the scope where it is defined. The concept was first introduced in C99 standard. An example for compound literal is

Examples from C standard, C11-§6.5.2.5/9:

```
int *p = (int [2]){ 2, 4 };
```

p is initialized to the address of the first element of an unnamed array of two ints.

The compound literal is an lvalue. The storage duration of the unnamed object is either static (if the literal appears at file scope) or automatic (if the literal appears at block scope), and in the latter case the object's lifetime ends when control leaves the enclosing block.

```
void f(void)
{
    int *p;
    /*...*/
    p = (int [2]){ *p };
    /*...*/
}
```

p is assigned the address of the first element of an array of two ints, the first having the value previously pointed to by p and the second, zero.[...]

Here p remains valid until the end of the block.

Compound literal with designators

(also from C11)

```
struct point {
    unsigned x;
    unsigned y;
};

extern void drawline(struct point, struct point);

// used somewhere like this
drawline((struct point){.x=1, .y=1}, (struct point){.x=3, .y=4});
```

A fictive function drawline receives two arguments of type `struct point`. The first has coordinate values `x == 1` and `y == 1`，whereas the second has `x == 3` and `y == 4`

Compound literal without specifying array length

```
int *p = (int []){ 1, 2, 3};
```

In this case the size of the array is no specified then it will be determined by the length of the initializer.

Compound literal having length of initializer less than array size specified

```
int *p = (int [10]){1, 2, 3};
```

复合字面量中其余元素将隐式初始化为0。

只读复合字面量

请注意，复合字面量是一个左值，因此其元素可以被修改。使用`const`限定符可以指定一个只读复合字面量，例如`(const int[]){1,2}`。

包含任意表达式的复合字面量

在函数内部，复合字面量（自C99起，任何初始化都可以）可以包含任意表达式。

```
void foo()
{
    int *p;
    int i = 2; j = 5;
    /*...*/
    p = (int [2]){ i+j, i*j };
    /*...*/
}
```

rest of the elements of compound literal will be initialized to 0 implicitly.

Read-only compound literal

Note that a compound literal is an lvalue and therefore its elements can be modifiable. A *read-only* compound literal can be specified using `const` qualifier as `(const int []){1, 2}`.

Compound literal containing arbitrary expressions

Inside a function, a compound literal, as for any initialization since C99, can have arbitrary expressions.

```
void foo()
{
    int *p;
    int i = 2; j = 5;
    /*...*/
    p = (int [2]){ i+j, i*j };
    /*...*/
}
```

第9章：位域

参数	描述
类型说明符 <code>signed</code> , <code>unsigned</code> , <code>int</code> 或 <code>_Bool</code>	
标识符	结构中该字段的名称
大小	该字段使用的位数

C语言中的大多数变量大小都是字节的整数倍。位域是结构体的一部分，它们不一定占用整数个字节；它们可以是任意数量的位。多个位域可以打包到一个存储单元中。它们是标准C语言的一部分，但有许多方面是由实现定义的。它们是C语言中可移植性最差的部分之一。

第9.1节：位域

简单的位域可以用来描述可能涉及特定位数的事物。

```
struct encoderPosition {
    unsigned int encoderCounts : 23;
    unsigned int encoderTurns : 4;
    unsigned int _reserved : 5;
};
```

在这个例子中，我们考虑一个具有23位单精度和4位多圈描述的编码器。位域通常用于与输出特定位数数据的硬件接口。另一个例子可能是与FPGA的通信，其中FPGA以32位为单位将数据写入你的内存，从而支持硬件读取：

```
struct FPGAInfo {
    union {
        struct bits {
            unsigned int bulb10n : 1;
            unsigned int bulb20n : 1;
            unsigned int bulb10ff : 1;
            unsigned int bulb20ff : 1;
            unsigned int jetOn : 1;
        };
        unsigned int data;
    };
};
```

在此示例中，我们展示了一个常用的结构，可以访问数据的各个位，或者整体写入数据包（模拟FPGA可能执行的操作）。然后我们可以这样访问各个位：

```
FPGAInfo fInfo;fIn
fo.data = 0xFF34F;if (fIn
o.bits.bulb1On) {printf("灯泡1
已开启");}
```

这是有效的，但根据C99标准6.7.2.1条款10：

位域在一个单元内的分配顺序（高位到低位或低位到高位）是由实现定义的。

Chapter 9: Bit-fields

Parameter	Description
type-specifier <code>signed</code> , <code>unsigned</code> , <code>int</code> or <code>_Bool</code>	
identifier	The name for this field in the structure
size	The number of bits to use for this field

Most variables in C have a size that is an integral number of bytes. Bit-fields are a part of a structure that don't necessarily occupy a integral number of bytes; they can any number of bits. Multiple bit-fields can be packed into a single storage unit. They are a part of standard C, but there are many aspects that are implementation defined. They are one of the least portable parts of C.

Section 9.1: Bit-fields

A simple bit-field can be used to describe things that may have a specific number of bits involved.

```
struct encoderPosition {
    unsigned int encoderCounts : 23;
    unsigned int encoderTurns : 4;
    unsigned int _reserved : 5;
};
```

In this example we consider an encoder with 23 bits of single precision and 4 bits to describe multi-turn. Bit-fields are often used when interfacing with hardware that outputs data associated with specific number of bits. Another example could be communication with an FPGA, where the FPGA writes data into your memory in 32 bit sections allowing for hardware reads:

```
struct FPGAInfo {
    union {
        struct bits {
            unsigned int bulb10n : 1;
            unsigned int bulb20n : 1;
            unsigned int bulb10ff : 1;
            unsigned int bulb20ff : 1;
            unsigned int jetOn : 1;
        };
        unsigned int data;
    };
};
```

For this example we have shown a commonly used construct to be able to access the data in its individual bits, or to write the data packet as a whole (emulating what the FPGA might do). We could then access the bits like this:

```
FPGAInfo fInfo;
fInfo.data = 0xFF34F;
if (fInfo.bits.bulb1On) {
    printf("Bulb 1 is on\n");
}
```

This is valid, but as per the C99 standard 6.7.2.1, item 10:

The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined.

在以这种方式定义位域时，需要注意字节序。因此，可能需要使用预处理指令来检查机器的字节序。以下是一个示例：

```
typedef union {
    struct bits {
#if defined(WIN32) || defined(LITTLE_ENDIAN)
        uint8_t commFailure :1;
        uint8_t hardwareFailure :1;
        uint8_t _reserved :6;
#else
        uint8_t _reserved :6;
        uint8_t hardwareFailure :1;
        uint8_t commFailure :1;
#endif
    };
    uint8_t data;
} hardwareStatus;
```

第9.2节：将位域用作小整数

```
#include <stdio.h>

int main(void)
{
    /* 定义一个可以存储0到7之间值的小位域 */
    struct
    {
        unsigned int uint3: 3;
    } small;

    /* 从一个值中提取右边的3位 */
    unsigned int value = 255 - 2; /* 二进制 11111101 */
    small.uint3 = value;          /* 二进制      101 */
    printf("%d", small.uint3);

    /* 这实际上是一个无限循环 */
    for (小型.uint3 = 0; 小型.uint3 < 8; 小型.uint3++)
    {
        printf("%d", 小型.uint3);
    }

    return 0;
}
```

第9.3节：位域对齐

位域允许声明比字符宽度更小的结构字段。位域通过字节级或字级掩码实现。以下示例生成一个8字节的结构体。

结构体 C

```
{
    短整型 s;           /* 2字节 */
    字符型 c;           /* 1字节 */
    整型 bit1 : 1;       /* 1位 */
    整型 nib : 4;        /* 4位，填充到8位边界。因此填充了3位 */
    整型 sept : 7;        /* 7位七元组，填充到32位边界。*/
};
```

You need to be aware of endianness when defining bit-fields in this way. As such it may be necessary to use a preprocessor directive to check for the endianness of the machine. An example of this follows:

```
typedef union {
    struct bits {
#if defined(WIN32) || defined(LITTLE_ENDIAN)
        uint8_t commFailure :1;
        uint8_t hardwareFailure :1;
        uint8_t _reserved :6;
#else
        uint8_t _reserved :6;
        uint8_t hardwareFailure :1;
        uint8_t commFailure :1;
#endif
    };
    uint8_t data;
} hardwareStatus;
```

Section 9.2: Using bit-fields as small integers

```
#include <stdio.h>

int main(void)
{
    /* define a small bit-field that can hold values from 0 .. 7 */
    struct
    {
        unsigned int uint3: 3;
    } small;

    /* extract the right 3 bits from a value */
    unsigned int value = 255 - 2; /* Binary 11111101 */
    small.uint3 = value;          /* Binary      101 */
    printf("%d", small.uint3);

    /* This is in effect an infinite loop */
    for (small.uint3 = 0; small.uint3 < 8; small.uint3++)
    {
        printf("%d\n", small.uint3);
    }

    return 0;
}
```

Section 9.3: Bit-field alignment

Bit-fields give an ability to declare structure fields that are smaller than the character width. Bit-fields are implemented with byte-level or word-level mask. The following example results in a structure of 8 bytes.

```
struct C
{
    short s;           /* 2 bytes */
    char c;            /* 1 byte */
    int bit1 : 1;       /* 1 bit */
    int nib : 4;        /* 4 bits padded up to boundary of 8 bits. Thus 3 bits are padded */
    int sept : 7;        /* 7 Bits septet, padded up to boundary of 32 bits. */
};
```

注释描述了一种可能的布局，但由于标准规定可寻址存储单元的对齐方式未指定，因此也可能存在其他布局。

未命名的位域可以是任意大小，但不能被初始化或引用。

零宽度位域不能被命名，并且将下一个字段对齐到位域数据类型定义的边界。这是通过在位域之间填充位实现的。

结构体 'A' 的大小是1字节。

```
struct A
{
    unsigned char c1 : 3;
    unsigned char c2 : 4;
    unsigned char c3 : 1;
};
```

在结构体 B 中，第一个未命名的位域跳过了2位；c2 后的零宽度位域使得 c3 从字符边界开始（因此在 c2 和 c3 之间跳过了3位）。c4 后有3个位的填充。因此结构体的大小是2字节。

```
struct B
{
    unsigned char c1 : 1;
    unsigned char : 2; /* 在布局中跳过2位 */
    unsigned char c2 : 2;
    unsigned char : 0; /* 导致填充到下一个容器边界 */
    unsigned char c3 : 4;
    unsigned char c4 : 1;
};
```

第9.4节：位域的禁忌事项

1. 不允许使用位域数组、指向位域的指针以及返回位域的函数。
2. 地址运算符 (&) 不能应用于位域成员。
3. 位域的数据类型必须足够宽，以容纳该字段的大小。
4. sizeof() 操作符不能应用于位域。
5. 无法单独为位域创建typedef（尽管您当然可以为一个包含位域的结构体）。

```
typedef struct mybitfield
{
    unsigned char c1 : 20; /* 错误，见第3点 */
    unsigned char c2 : 4; /* 正确 */
    unsigned char c3 : 1;
    unsigned int x[10]: 5; /* 错误，见第1点 */
} A;

int SomeFunction(void)
{
    // 代码中的某处
    A a = { ... };
    printf("Address of a.c2 is %p", &a.c2); /* 错误，见第2点 */
    printf("Size of a.c2 is %zu", sizeof(a.c2)); /* 错误，见第4点 */
}
```

The comments describe one possible layout, but because the standard says *the alignment of the addressable storage unit is unspecified*, other layouts are also possible.

An unnamed bit-field may be of any size, but they can't be initialized or referenced.

A zero-width bit-field cannot be given a name and aligns the next field to the boundary defined by the datatype of the bit-field. This is achieved by padding bits between the bit-fields.

The size of structure 'A' is 1 byte.

```
struct A
{
    unsigned char c1 : 3;
    unsigned char c2 : 4;
    unsigned char c3 : 1;
};
```

In structure B, the first unnamed bit-field skips 2 bits; the zero width bit-field after c2 causes c3 to start from the char boundary (so 3 bits are skipped between c2 and c3. There are 3 padding bits after c4. Thus the size of the structure is 2 bytes.

```
struct B
{
    unsigned char c1 : 1;
    unsigned char : 2; /* Skips 2 bits in the layout */
    unsigned char c2 : 2;
    unsigned char : 0; /* Causes padding up to next container boundary */
    unsigned char c3 : 4;
    unsigned char c4 : 1;
};
```

Section 9.4: Don'ts for bit-fields

1. Arrays of bit-fields, pointers to bit-fields and functions returning bit-fields are not allowed.
2. The address operator (&) cannot be applied to bit-field members.
3. The data type of a bit-field must be wide enough to contain the size of the field.
4. The sizeof() operator cannot be applied to a bit-field.
5. There is no way to create a typedef for a bit-field in isolation (though you can certainly create a typedef for a structure containing bit-fields).

```
typedef struct mybitfield
{
    unsigned char c1 : 20; /* incorrect, see point 3 */
    unsigned char c2 : 4; /* correct */
    unsigned char c3 : 1;
    unsigned int x[10]: 5; /* incorrect, see point 1 */
} A;

int SomeFunction(void)
{
    // Somewhere in the code
    A a = { ... };
    printf("Address of a.c2 is %p\n", &a.c2); /* incorrect, see point 2 */
    printf("Size of a.c2 is %zu\n", sizeof(a.c2)); /* incorrect, see point 4 */
}
```

第9.5节：位域什么时候有用？

位域用于将多个变量合并为一个对象，类似于结构体。这可以减少内存使用，尤其在嵌入式环境中非常有用。

例如，考虑以下变量，其取值范围如下。

```
a --> 范围 0 - 3  
b --> 范围 0 - 1  
c --> 范围 0 - 7  
d --> 范围 0 - 1  
e --> 范围 0 - 1
```

如果我们分别声明这些变量，那么每个变量至少需要一个8位整数，总共需要5个字节的空间。此外，这些变量不会使用8位无符号整数（0-255）的全部范围。这里我们可以使用位域。

```
typedef struct {  
    unsigned int a:2;  
    unsigned int b:1;  
    unsigned int c:3;  
    unsigned int d:1;  
    unsigned int e:1;  
} bit_a;
```

结构体中的位域访问方式与其他结构体成员相同。程序员需要确保变量的值在范围内。如果超出范围，行为是未定义的。

```
int main(void)  
{  
bit_a bita_var;  
bita_var.a = 2;           // 写入元素 a  
printf ("%d",bita_var.a); // 读取元素 a。  
return 0;  
}
```

程序员经常希望将一组位域清零。这可以逐个元素完成，但还有第二种方法。只需将上述结构体与一个无符号类型（其大小大于或等于结构体大小）联合起来。然后通过将该无符号整数清零，即可将整组位域清零。

```
typedef union {  
    struct {  
        unsigned int a:2;  
        unsigned int b:1;  
        unsigned int c:3;  
        unsigned int d:1;  
        unsigned int e:1;  
    };  
    uint8_t data;  
} union_bit;
```

用法如下

```
int main(void)  
{  
union_bit un_bit;  
un_bit.data = 0x00;      // 清除整个位域  
un_bit.a = 2;            // 写入元素 a
```

Section 9.5: When are bit-fields useful?

A bit-field is used to club together many variables into one object, similar to a structure. This allows for reduced memory usage and is especially useful in an embedded environment.

e.g. consider the following variables having the ranges as given below.

```
a --> range 0 - 3  
b --> range 0 - 1  
c --> range 0 - 7  
d --> range 0 - 1  
e --> range 0 - 1
```

If we declare these variables separately, then each has to be at least an 8-bit integer and the total space required will be 5 bytes. Moreover the variables will not use the entire range of an 8 bit unsigned integer (0-255). Here we can use bit-fields.

```
typedef struct {  
    unsigned int a:2;  
    unsigned int b:1;  
    unsigned int c:3;  
    unsigned int d:1;  
    unsigned int e:1;  
} bit_a;
```

The bit-fields in the structure are accessed the same as any other structure. The programmer needs to take care that the variables are written in range. If out of range the behaviour is undefined.

```
int main(void)  
{  
bit_a bita_var;  
bita_var.a = 2;           // to write into element a  
printf ("%d",bita_var.a); // to read from element a.  
return 0;  
}
```

Often the programmer wants to zero the set of bit-fields. This can be done element by element, but there is second method. Simply create a union of the structure above with an unsigned type that is greater than, or equal to, the size of the structure. Then the entire set of bit-fields may be zeroed by zeroing this unsigned integer.

```
typedef union {  
    struct {  
        unsigned int a:2;  
        unsigned int b:1;  
        unsigned int c:3;  
        unsigned int d:1;  
        unsigned int e:1;  
    };  
    uint8_t data;  
} union_bit;
```

Usage is as follows

```
int main(void)  
{  
union_bit un_bit;  
un_bit.data = 0x00;      // clear the whole bit-field  
un_bit.a = 2;            // write into element a
```

```
printf ("%d",un_bit.a); // 从元素 a 读取。  
return 0;  
}
```

总之，位域通常用于内存受限的情况下，这时你有许多变量且它们的取值范围有限。

```
printf ("%d",un_bit.a); // read from element a.  
return 0;  
}
```

In conclusion, bit-fields are commonly used in memory constrained situations where you have a lot of variables which can take on limited ranges.

第10章：数组

数组是派生数据类型，表示另一种类型的有序值集合（“元素”）。大多数C语言中的数组具有固定数量的同一类型元素，其表示方式是将元素连续存储在内存中，没有间隙或填充。C语言允许多维数组，其元素是其他数组，也允许指针数组。

C语言支持动态分配的数组，其大小在运行时确定。C99及以后版本支持变长数组（VLA）。

第10.1节：声明和初始化数组

声明一维数组的一般语法是

```
类型 arrName[大小];
```

其中类型可以是任何内置类型或用户定义的类型，如结构体，数组名是用户定义的标识符，而大小是一个整数常量。

声明一个数组（此处为包含10个int变量的数组）的方法如下：

```
int array[10];
```

此时它包含未确定的值。若要在声明时确保其值为零，可以这样做：

```
int array[10] = {0};
```

数组也可以有初始化器，此例声明了一个包含10个int的数组，其中前三个int将包含值1、2、3，其他所有值将为零：

```
int array[10] = {1, 2, 3};
```

在上述初始化方法中，列表中的第一个值将赋给数组的第一个元素，第二个值将赋给数组的第二个元素，依此类推。如果列表大小小于数组大小，则如上例所示，数组剩余的元素将被初始化为零。通过指定列表初始化（ISO C99），可以显式初始化数组成员。例如，

```
int array[5] = {[2] = 5, [1] = 2, [4] = 9}; /* 数组为 {0, 2, 5, 0, 9} */
```

在大多数情况下，编译器可以为你推断数组的长度，这可以通过将方括号留空来实现：

```
int array[] = {1, 2, 3}; /* 一个包含3个整数的数组 */  
int array[] = {[3] = 8, [0] = 9}; /* 大小为4 */
```

声明长度为零的数组是不允许的。

版本 ≥ C99 版本 < C11

可变长度数组（简称VLA）是在C99中添加的，并在C11中变为可选。它们与普通数组相同，只有一个重要的区别：长度不必在编译时已知。VLA具有自动存储期。只有指向VLA的指针可以具有静态存储期。

Chapter 10: Arrays

Arrays are derived data types, representing an ordered collection of values ("elements") of another type. Most arrays in C have a fixed number of elements of any one type, and its representation stores the elements contiguously in memory without gaps or padding. C allows multidimensional arrays whose elements are other arrays, and also arrays of pointers.

C supports dynamically allocated arrays whose size is determined at run time. C99 and later supports variable length arrays or VLAs.

Section 10.1: Declaring and initializing an array

The general syntax for declaring a one-dimensional array is

```
type arrName[size];
```

where type could be any built-in type or user-defined types such as structures, arrName is a user-defined identifier, and size is an integer constant.

Declaring an array (an array of 10 int variables in this case) is done like this:

```
int array[10];
```

it now holds indeterminate values. To ensure it holds zero values while declaring, you can do this:

```
int array[10] = {0};
```

Arrays can also have initializers, this example declares an array of 10 int's, where the first 3 int's will contain the values 1, 2, 3, all other values will be zero:

```
int array[10] = {1, 2, 3};
```

In the above method of initialization, the first value in the list will be assigned to the first member of the array, the second value will be assigned to the second member of the array and so on. If the list size is smaller than the array size, then as in the above example, the remaining members of the array will be initialized to zeros. With designated list initialization (ISO C99), explicit initialization of the array members is possible. For example,

```
int array[5] = {[2] = 5, [1] = 2, [4] = 9}; /* array is {0, 2, 5, 0, 9} */
```

In most cases, the compiler can deduce the length of the array for you, this can be achieved by leaving the square brackets empty:

```
int array[] = {1, 2, 3}; /* an array of 3 int's */  
int array[] = {[3] = 8, [0] = 9}; /* size is 4 */
```

Declaring an array of zero length is not allowed.

Version ≥ C99 Version < C11

Variable Length Arrays (VLA for short) were added in C99, and made optional in C11. They are equal to normal arrays, with one, important, difference: The length doesn't have to be known at compile time. VLA's have automatic storage duration. Only pointers to VLA's can have static storage duration.

```
size_t m = calc_length(); /* 在运行时计算数组长度 */
int vla[m]; /* 创建具有计算长度的数组 */
```

重要：

VLA可能存在危险。如果上例中的数组vla在栈上需要的空间超过可用空间，栈将会溢出。因此，风格指南、书籍和练习中通常不推荐使用VLA。

第10.2节：高效遍历数组及行优先顺序

C语言中的数组可以看作是一块连续的内存。更准确地说，数组的最后一个维度是连续的部分。我们称之为行优先顺序。理解这一点以及缓存未命中时会将完整的缓存行加载到缓存中以防止后续缓存未命中，可以理解为什么访问一个 10000×10000 维度的数组时，访问array[0][0]会可能将array[0][1]加载到缓存中，但紧接着访问array[1][0]会产生第二次缓存未命中，因为它距离array[0][0]有 $\text{sizeof}(\text{type}) * 10000$ 字节，显然不在同一缓存行上。这就是为什么这样遍历效率低下的原因：

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[j][i] = 0;
    }
}
```

这样迭代更高效：

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[i][j] = 0;
    }
}
```

同理，这就是为什么在处理一维数组但有多个索引（这里为了简单起见假设是二维，索引为 i 和 j）时，重要的是这样遍历数组：

```
#define DIM_X 10
#define DIM_Y 20

int array[DIM_X*DIM_Y];

size_t i, j;
for (i = 0; i < DIM_X; ++i)
```

```
size_t m = calc_length(); /* calculate array length at runtime */
int vla[m]; /* create array with calculated length */
```

Important:

VLA's are potentially dangerous. If the array vla in the example above requires more space on the stack than available, the stack will overflow. Usage of VLA's is therefore often discouraged in style guides and by books and exercises.

Section 10.2: Iterating through an array efficiently and row-major order

Arrays in C can be seen as a contiguous chunk of memory. More precisely, the last dimension of the array is the contiguous part. We call this the *row-major order*. Understanding this and the fact that a cache fault loads a complete cache line into the cache when accessing uncached data to prevent subsequent cache faults, we can see why accessing an array of dimension 10000×10000 with array[0][0] would **potentially** load array[0][1] in cache, but accessing array[1][0] right after would generate a second cache fault, since it is $\text{sizeof}(\text{type}) * 10000$ bytes away from array[0][0], and therefore certainly not on the same cache line. Which is why iterating like this is inefficient:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[j][i] = 0;
    }
}
```

And iterating like this is more efficient:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[i][j] = 0;
    }
}
```

In the same vein, this is why when dealing with an array with one dimension and multiple indexes (let's say 2 dimensions here for simplicity with indexes i and j) it is important to iterate through the array like this:

```
#define DIM_X 10
#define DIM_Y 20

int array[DIM_X*DIM_Y];

size_t i, j;
for (i = 0; i < DIM_X; ++i)
```

```

{
    for(j = 0; j < DIM_Y; ++j)
    {
        array[i*DIM_Y+j] = 0;
    }
}

```

或者对于三维数组，索引为 i、j 和 k：

```

#define DIM_X 10
#define DIM_Y 20
#define DIM_Z 30

int array[DIM_X*DIM_Y*DIM_Z];

size_t i, j, k;
for (i = 0; i < DIM_X; ++i)
{
    for(j = 0; j < DIM_Y; ++j)
    {
        for (k = 0; k < DIM_Z; ++k)
        {
            array[i*DIM_Y*DIM_Z+j*DIM_Z+k] = 0;
        }
    }
}

```

或者更通用的方式，当我们有一个包含 **N₁ × N₂ × ... × N_d** 个元素的数组，**d** 维度，索引记为 n_{1,n_{2,...,n_d 时，偏移量的计算方式如下}}

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\dots + N_2 n_1) \dots)) = \sum_{k=1}^d \left(\prod_{\ell=k+1}^d N_\ell \right) n_k$$

图片/公式来源：https://en.wikipedia.org/wiki/Row-major_order

第10.3节：数组长度

数组具有在其声明范围内已知的固定长度。然而，有时计算数组长度是可能且方便的。特别是当数组长度由初始化器自动确定时，这可以使代码更灵活：

```

int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

/* `array` 的字节大小 */
size_t size = sizeof(array);

/* `array` 中元素的数量 */
size_t length = sizeof(array) / sizeof(array[0]);

```

然而，在大多数表达式中出现数组的情况下，数组会自动转换为指向其第一个元素的指针（称为“衰减”）。数组作为 size of 操作符的操作数是少数例外之一。得到的指针本身不是数组，也不包含任何关于其来源数组长度的信息。因此，如果需要结合指针使用该长度，比如指针被传递给函数时，必须单独传递该长度。

例如，假设我们想编写一个函数来返回 int 类型数组的最后一个元素。接着上文，

```

{
    for(j = 0; j < DIM_Y; ++j)
    {
        array[i*DIM_Y+j] = 0;
    }
}

```

Or with 3 dimensions and indexes i,j and k:

```

#define DIM_X 10
#define DIM_Y 20
#define DIM_Z 30

int array[DIM_X*DIM_Y*DIM_Z];

size_t i, j, k;
for (i = 0; i < DIM_X; ++i)
{
    for(j = 0; j < DIM_Y; ++j)
    {
        for (k = 0; k < DIM_Z; ++k)
        {
            array[i*DIM_Y*DIM_Z+j*DIM_Z+k] = 0;
        }
    }
}

```

Or in a more generic way, when we have an array with **N₁ × N₂ × ... × N_d** elements, **d** dimensions and indices noted as **n_{1,n_{2,...,n_d}}** the offset is calculated like this

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\dots + N_2 n_1) \dots)) = \sum_{k=1}^d \left(\prod_{\ell=k+1}^d N_\ell \right) n_k$$

Picture/formula taken from: https://en.wikipedia.org/wiki/Row-major_order

Section 10.3: Array length

Arrays have fixed lengths that are known within the scope of their declarations. Nevertheless, it is possible and sometimes convenient to calculate array lengths. In particular, this can make code more flexible when the array length is determined automatically from an initializer:

```

int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

/* size of `array` in bytes */
size_t size = sizeof(array);

/* number of elements in `array` */
size_t length = sizeof(array) / sizeof(array[0]);

```

However, in most contexts where an array appears in an expression, it is automatically converted to ("decays to") a pointer to its first element. The case where an array is the operand of the **sizeof** operator is one of a small number of exceptions. The resulting pointer is not itself an array, and it does not carry any information about the length of the array from which it was derived. Therefore, if that length is needed in conjunction with the pointer, such as when the pointer is passed to a function, then it must be conveyed separately.

For example, suppose we want to write a function to return the last element of an array of **int**. Continuing from the

我们可以这样调用它：

```
/* 数组会衰减为指针，因此长度必须单独传递 */
int last = get_last(array, length);
```

该函数可以这样实现：

```
int get_last(int input[], size_t length) {
    return input[length - 1];
}
```

特别注意，虽然参数 `input` 的声明看起来像数组，但它实际上声明了 `input` 为指针（指向 `int`）。这与声明 `input` 为 `int *input` 完全等价。即使给出了维度也是如此。这是因为数组永远不能作为函数的实际参数（它们在函数调用表达式中会衰减为指针），这也可以说是一种助记方式。

试图通过指针确定数组大小是一个非常常见的错误，这是行不通的。不要这样做：

```
int BAD_get_last(int input[]) {
    /* 错误地计算了 input 指向的数组的长度：*/
    size_t length = sizeof(input) / sizeof(input[0]);

    return input[length - 1]; /* 啊呀——这不是我们要找的目标 */
}
```

事实上，这种错误非常常见，以至于一些编译器能够识别并发出警告。例如，clang 会发出如下警告：

```
warning: sizeof 对数组函数参数将返回 'int *' 的大小，而不是 'int []' [-Wsizeof-array-argument]
        int length = sizeof(input) / sizeof(input[0]);
        ^
note: 在此处声明
int BAD_get_last(int input[])
        ^
```

第10.4节：将多维数组传递给函数

多维数组在传递给函数时遵循与一维数组相同的规则。

然而，指针的衰变、运算符优先级以及声明多维数组的两种不同方式（数组的数组与指针的数组）相结合，可能会使此类函数的声明变得不直观。以下示例展示了传递多维数组的正确方法。

```
#include <assert.h>
#include <stdlib.h>

/* 当传递多维数组（即数组的数组）给函数时，它会像往常一样衰变为指向第一个元素的指针
   。但只有顶层会衰变，因此传递的是指向某个固定大小数组（此处为4）的指针。 */
void f(int x[][4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* 这个原型等价于 f(int x[][4])。
   因为 [index] 的优先级高于 *expr，所以需要在 *x 周围加括号，否则 int *x[4] 通常等价
   于 int
```

above, we might call it like so:

```
/* array will decay to a pointer, so the length must be passed separately */
int last = get_last(array, length);
```

The function could be implemented like this:

```
int get_last(int input[], size_t length) {
    return input[length - 1];
}
```

Note in particular that although the declaration of parameter `input` resembles that of an array, **it in fact declares `input` as a pointer** (to `int`). It is exactly equivalent to declaring `input` as `int *input`. The same would be true even if a dimension were given. This is possible because arrays cannot ever be actual arguments to functions (they decay to pointers when they appear in function call expressions), and it can be viewed as mnemonic.

It is a very common error to attempt to determine array size from a pointer, which cannot work. DO NOT DO THIS:

```
int BAD_get_last(int input[]) {
    /* INCORRECTLY COMPUTES THE LENGTH OF THE ARRAY INTO WHICH input POINTS: */
    size_t length = sizeof(input) / sizeof(input[0]);

    return input[length - 1]; /* Oops -- not the droid we are looking for */
}
```

In fact, that particular error is so common that some compilers recognize it and warn about it. clang, for instance, will emit the following warning:

```
warning: sizeof on array function parameter will return size of 'int *' instead of 'int []' [-Wsizeof-array-argument]
        int length = sizeof(input) / sizeof(input[0]);
        ^
note: declared here
int BAD_get_last(int input[])
        ^
```

Section 10.4: Passing multidimensional arrays to a function

Multidimensional arrays follow the same rules as single-dimensional arrays when passing them to a function. However the combination of decay-to-pointer, operator precedence, and the two different ways to declare a multidimensional array (array of arrays vs array of pointers) may make the declaration of such functions non-intuitive. The following example shows the correct ways to pass multidimensional arrays.

```
#include <assert.h>
#include <stdlib.h>

/* When passing a multidimensional array (i.e. an array of arrays) to a
   function, it decays into a pointer to the first element as usual. But only
   the top level decays, so what is passed is a pointer to an array of some fixed
   size (4 in this case). */
void f(int x[][4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* This prototype is equivalent to f(int x[][4]).
   The parentheses around *x are required because [index] has a higher
   precedence than *expr, thus int *x[4] would normally be equivalent to int
```

```

*(x[4]), 即一个包含4个指向int指针的数组。但如果它被声明为函数参数，则会衰变为指针，变成
int **x,
这与 x[2][4] 不兼容。 */
void g(int (*x)[4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* 可以传递指针数组给这个参数，因为它会衰变为指向指针的指针，但不能传递数组的数组。 */
void h(int **x) {
    assert(sizeof(*x) == sizeof(int*));
}

int main(void) {
    int foo[2][4];
    f(foo);
    g(foo);

    /* 这里我们动态创建了一个指针数组。注意，每个维度的大小不是数据类型的一部分，因此类型系统仅将其视为指向指针的指针，而不是指向数组的指针或数组的数组。 */
    int **bar = malloc(sizeof(*bar) * 2);
    assert(bar);
    for (size_t i = 0; i < 2; i++) {
        bar[i] = malloc(sizeof(*bar[i]) * 4);
        assert(bar[i]);
    }

    h(bar);

    for (size_t i = 0; i < 2; i++) {
        free(bar[i]);
    }
    free(bar);
}

```

另见

[将数组传递给函数](#)

第10.5节：多维数组

C语言允许多维数组。以下是多维数组声明的一般形式

类型 名称[大小1][大小2]..[大小N];

例如，以下声明创建了一个三维 ($5 \times 10 \times 4$) 整数数组：

```
int arr[5][10][4];
```

二维数组

多维数组中最简单的形式是二维数组。二维数组本质上是一维数组的列表。要声明一个维度为 $m \times n$ 的二维整数数组，我们可以写成如下形式：

```

*(x[4]), i.e. an array of 4 pointers to int. But if it's declared as a
function parameter, it decays into a pointer and becomes int **x,
which is not comparable with x[2][4]. */
void g(int (*x)[4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* An array of pointers may be passed to this, since it'll decay into a pointer
   to pointer, but an array of arrays may not. */
void h(int **x) {
    assert(sizeof(*x) == sizeof(int*));
}

int main(void) {
    int foo[2][4];
    f(foo);
    g(foo);

    /* Here we're dynamically creating an array of pointers. Note that the
       size of each dimension is not part of the datatype, and so the type
       system just treats it as a pointer to pointer, not a pointer to array
       or array of arrays. */
    int **bar = malloc(sizeof(*bar) * 2);
    assert(bar);
    for (size_t i = 0; i < 2; i++) {
        bar[i] = malloc(sizeof(*bar[i]) * 4);
        assert(bar[i]);
    }

    h(bar);

    for (size_t i = 0; i < 2; i++) {
        free(bar[i]);
    }
    free(bar);
}

```

See also

[Passing in Arrays to Functions](#)

Section 10.5: Multi-dimensional arrays

The C programming language allows [multidimensional arrays](#). Here is the general form of a multidimensional array declaration –

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional ($5 \times 10 \times 4$) integer array:

```
int arr[5][10][4];
```

Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of dimensions $m \times n$, we can write as follows:

类型 数组名[行数][列数];

其中 **类型** 可以是任何有效的 C 数据类型 (int、float 等) , 数组名 可以是任何有效的 C 标识符。二维数组可以被视为一个有 行数 行和 列数 列的表格。 注意：在 C 语言中顺序 很重要。数组 int a[4][3] 与数组 int a[3][4] 不同。行数优先，因为 C 是 行优先 语言。

一个包含三行四列的二维数组 a 可以表示如下：

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

因此，数组 a 中的每个元素都由形式为 a[i][j] 的元素名标识，其中 a 是数组名，i 表示第几行，j 表示第几列。请记住，行和列都是从零开始索引的。这与数学中二维矩阵的下标表示法非常相似。

二维数组的初始化

多维数组可以通过为每一行指定括号内的值来初始化。以下定义了一个有 3 行，每行有 4 列的数组。

```
int a[3][4] = {  
    {0, 1, 2, 3} , /* 第 0 行的初始化值 */  
    {4, 5, 6, 7} , /* 第 1 行的初始化值 */  
    {8, 9, 10, 11} /* 第 2 行的初始化值 */  
};
```

表示行的嵌套大括号是可选的。以下初始化与前面的例子等价：

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

虽然使用嵌套大括号创建数组的方法是可选的，但强烈建议采用，因为这样更易读且更清晰。

访问二维数组元素

二维数组中的元素通过下标访问，即数组的行索引和列索引。例如-

```
int val = a[2][3];
```

上述语句将取数组第3行的第4个元素。让我们检查下面的程序
其中使用了嵌套循环来处理二维数组：

```
#include <stdio.h>  
  
int main () {  
  
    /* 一个有5行2列的数组 */  
    int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6}, {4,8} };  
}
```

type arrayName[m][n];

Where type can be any valid C data type (int, float, etc.) and arrayName can be any valid C identifier. A two-dimensional array can be visualized as a table with m rows and n columns. **Note:** The order does matter in C. The array int a[4][3] is not the same as the array int a[3][4]. The number of rows comes first as C is a row-major language.

A two-dimensional array a, which contains three rows and four columns can be shown as follows:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array a is identified by an element name of the form a[i][j], where a is the name of the array, i represents which row, and j represents which column. Recall that rows and columns are zero indexed. This is very similar to mathematical notation for subscripting 2-D matrices.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. The following define an array with 3 rows where each row has 4 columns.

```
int a[3][4] = {  
    {0, 1, 2, 3} , /* initializers for row indexed by 0 */  
    {4, 5, 6, 7} , /* initializers for row indexed by 1 */  
    {8, 9, 10, 11} /* initializers for row indexed by 2 */  
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

While the method of creating arrays with nested braces is optional, it is strongly encouraged as it is more readable and clearer.

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example –

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array. Let us check the following program where we have used a nested loop to handle a two-dimensional array:

```
#include <stdio.h>  
  
int main () {  
  
    /* an array with 5 rows and 2 columns*/  
    int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6}, {4,8} };  
}
```

```

int i, j;

/* 输出每个数组元素的值 */
for ( i = 0; i < 5; i++ ) {

    for ( j = 0; j < 2; j++ ) {printf("a
[%d][%d] = %d", i,j, a[i][j] );}

}

return 0;
}

```

当上述代码被编译并执行时，产生以下结果：

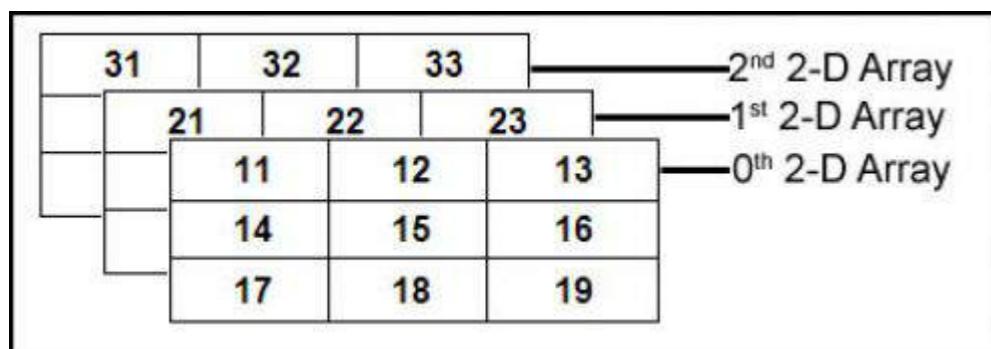
```

a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8

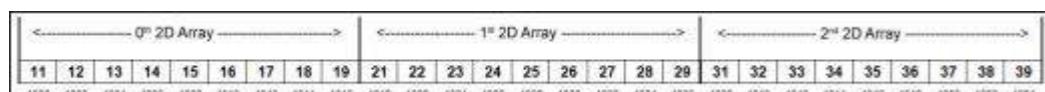
```

三维数组：

三维数组本质上是数组的数组的数组：它是二维数组的数组或集合，而二维数组是由一维数组组成的数组。



三维数组内存映射：



初始化三维数组：

```

double cprogram[3][2][4]={
{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
{{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
{{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};

```

我们可以拥有任意维数的数组，尽管大多数创建的数组很可能是一维或二维的。

```

int i, j;

/* output each array element's value */
for ( i = 0; i < 5; i++ ) {

    for ( j = 0; j < 2; j++ ) {
        printf("a[%d][%d] = %d\n", i,j, a[i][j] );
    }
}

return 0;
}

```

When the above code is compiled and executed, it produces the following result:

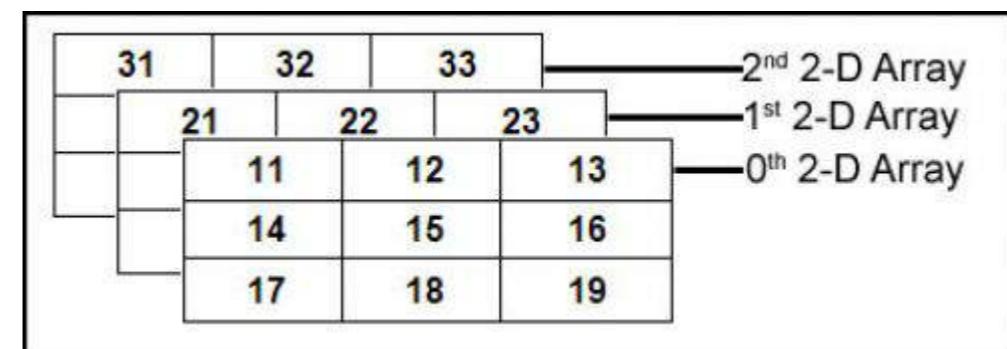
```

a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8

```

Three-Dimensional array:

A 3D array is essentially an array of arrays of arrays: it's an array or collection of 2D arrays, and a 2D array is an array of 1D arrays.



3D array memory map:



Initializing a 3D Array:

```

double cprogram[3][2][4]={
{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
{{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
{{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};

```

We can have arrays with any number of dimensions, although it is likely that most of the arrays that are created will be of one or two dimensions.

第10.6节：定义数组并访问数组元素

```
#include <stdio.h>

#define ARRLEN (10)

int main (void)
{
    int n[ ARRLEN ]; /* n 是一个包含10个整数的数组 */
    size_t i, j; /* 使用 size_t 来访问内存，即索引数组，因为它保证
                    足够宽以访问所有可能的可用内存。*/
    使用有符号整数来做这件事应被视为特殊用例，          并应限制在需要负索引的少见情况下。 */

    /* 初始化数组 n 的元素。*/
    for ( i = 0; i < ARRLEN ; i++ )
    {
        n[ i ] = i + 100; /* 将位置 i 处的元素设置为 i + 100。 */
    }

    /* 输出每个数组元素的值。*/
    for ( j = 0; j < ARRLEN ; j++ )
    {
        printf("Element[%zu] = %d\n", j, n[j]);
    }

    return 0;
}
```

第10.7节：清空数组内容（置零）

有时在初始化完成后，需要将数组设置为零。

```
#include <stdlib.h> /* 用于 EXIT_SUCCESS */

#define ARRLEN (10)

int main(void)
{
    int array[ARRLEN]; /* 已分配但未初始化，因为未定义为静态或全局变量。 */

    size_t i;
    for(i = 0; i < ARRLEN; ++i)
    {
        array[i] = 0;
    }

    return EXIT_SUCCESS;
}
```

上述循环的一个常用简写是使用`memset()`函数，来自`<string.h>`。传入`array`如下面所示，会使其退化为指向第一个元素的指针。

```
memset(array, 0, ARRLEN * sizeof (int)); /* 明确使用指定类型的大小（此处为 int）。 */
```

或

Section 10.6: Define array and access array element

```
#include <stdio.h>

#define ARRLEN (10)

int main (void)
{
    int n[ ARRLEN ]; /* n is an array of 10 integers */
    size_t i, j; /* Use size_t to address memory, that is to index arrays, as its guaranteed to
                    be wide enough to address all of the possible available memory.
                    Using signed integers to do so should be considered a special use case,
                    and should be restricted to the uncommon case of being in the need of
                    negative indexes. */

    /* Initialize elements of array n. */
    for ( i = 0; i < ARRLEN ; i++ )
    {
        n[ i ] = i + 100; /* Set element at location i to i + 100. */
    }

    /* Output each array element's value. */
    for ( j = 0; j < ARRLEN ; j++ )
    {
        printf("Element[%zu] = %d\n", j, n[j]);
    }

    return 0;
}
```

Section 10.7: Clearing array contents (zeroing)

Sometimes it's necessary to set an array to zero, after the initialization has been done.

```
#include <stdlib.h> /* for EXIT_SUCCESS */

#define ARRLEN (10)

int main(void)
{
    int array[ARRLEN]; /* Allocated but not initialised, as not defined static or global. */

    size_t i;
    for(i = 0; i < ARRLEN; ++i)
    {
        array[i] = 0;
    }

    return EXIT_SUCCESS;
}
```

A common short cut to the above loop is to use `memset()` from `<string.h>`. Passing `array` as shown below makes it decay to a pointer to its 1st element.

```
memset(array, 0, ARRLEN * sizeof (int)); /* Use size explicitly provided type (int here). */
```

or

```
memset(array, 0, ARRLEN * sizeof *array); /* 使用指针所指向类型的大小。 */
```

如本例中，array是一个数组，而不仅仅是指向数组第一个元素的指针（参见数组长度部分，了解其重要性），因此还有第三种将数组清零的方式：

```
memset(array, 0, sizeof array); /* 使用数组本身的大小。 */
```

第10.8节：在数组中设置值

访问数组值通常通过方括号进行：

```
int val;
int array[10];

/* 将第五个元素的值设置为5：*/
array[4] = 5;

/* 上述等同于：*/
*(array + 4) = 5;

/* 读取第五个元素的值：*/
val = array[4];
```

由于+运算符的操作数可交换（--> 交换律），以下表达式等价：

```
*(array + 4) = 5;
*(4 + array) = 5;
```

因此，以下语句是等价的：

```
array[4] = 5;
4[array] = 5; /* 奇怪但有效的C代码... */
```

这两条语句也是等价的：

```
val = array[4];
val = 4[array]; /* 奇怪但有效的C代码... */
```

C语言不会执行任何边界检查，访问声明数组之外的内容是未定义行为（访问分配块之外的内存）：

```
int val;
int array[10];

array[4] = 5; /* 正确 */
val = array[4]; /* 正确 */
array[19] = 20; /* 未定义行为 */
val = array[15]; /* 未定义行为 */
```

第10.9节：分配并零初始化一个用户定义大小的数组

```
#include <stdio.h>
#include <stdlib.h>
```

```
memset(array, 0, ARRLEN * sizeof *array); /* Use size of type the pointer is pointing to. */
```

As in this example array is an array and not just a pointer to an array's 1st element (see Array length on why this is important) a third option to 0-out the array is possible:

```
memset(array, 0, sizeof array); /* Use size of the array itself. */
```

Section 10.8: Setting values in arrays

Accessing array values is generally done through square brackets:

```
int val;
int array[10];

/* Setting the value of the fifth element to 5: */
array[4] = 5;

/* The above is equal to: */
*(array + 4) = 5;

/* Reading the value of the fifth element: */
val = array[4];
```

As a side effect of the operands to the + operator being exchangeable (--> commutative law) the following is equivalent:

```
*(array + 4) = 5;
*(4 + array) = 5;
```

so as well the next statements are equivalent:

```
array[4] = 5;
4[array] = 5; /* Weird but valid C ... */
```

and those two as well:

```
val = array[4];
val = 4[array]; /* Weird but valid C ... */
```

C doesn't perform any boundary checks, accessing contents outside of the declared array is undefined (Accessing memory beyond allocated chunk):

```
int val;
int array[10];

array[4] = 5; /* ok */
val = array[4]; /* ok */
array[19] = 20; /* undefined behavior */
val = array[15]; /* undefined behavior */
```

Section 10.9: Allocate and zero-initialize an array with user defined size

```
#include <stdio.h>
#include <stdlib.h>
```

```

int main (void)
{
    int * pdata;
    size_t n;

    printf ("请输入数组的大小: ");
    fflush(stdout); /* 确保提示信息被打印到缓冲的标准输出。 */

    if (1 != scanf ("%zu", &n)) /* 如果不支持 %zu (Windows?) 则使用 %lu。 */
    {
        fprintf("scanf() 未能正确读取 a 的值。");exit(EXIT_FAILURE);
    }

    pdata = calloc(n, sizeof *pdata);
    if (NULL == pdata)
    {
        perror("calloc() 失败"); /* 打印错误信息。 */
        exit(EXIT_FAILURE);
    }

    free(pdata); /* 清理内存。 */

    return EXIT_SUCCESS;
}

```

该程序尝试从标准输入扫描一个无符号整数值，调用 `calloc()` 函数为一个包含 `n` 个 `int` 类型元素的数组分配内存块。后者会将内存初始化为全零。

如果成功，内存会通过调用 `free()` 释放。

第10.10节：使用指针遍历数组

```

#include <stdio.h>
#define SIZE (10)
int main()
{
    size_t i = 0;
    int *p = NULL;
    int a[SIZE];

    /* 设置值为  $i \cdot i$  */
    for(i = 0; i < SIZE; ++i)
    {
        a[i] = i * i;
    }

    /* 使用指针读取值 */
    for(p = a; p < a + SIZE; ++p)
    {
        printf("%d", *p);
    }

    return 0;
}

```

这里，在第一个for循环条件中对p的初始化时，数组a会衰减为指向其第一个元素的指针，就像在几乎所有使用该数组变量的地方一样。

然后，`++p`对指针p执行指针运算，逐个遍历数组元素

```

int main (void)
{
    int * pdata;
    size_t n;

    printf ("Enter the size of the array: ");
    fflush(stdout); /* Make sure the prompt gets printed to buffered stdout. */

    if (1 != scanf ("%zu", &n)) /* If zu is not supported (Windows?) use lu. */
    {
        fprintf("scanf() did not read a in proper value.\n");
        exit(EXIT_FAILURE);
    }

    pdata = calloc(n, sizeof *pdata);
    if (NULL == pdata)
    {
        perror("calloc() failed"); /* Print error. */
        exit(EXIT_FAILURE);
    }

    free(pdata); /* Clean up. */

    return EXIT_SUCCESS;
}

```

This program tries to scan in an unsigned integer value from standard input, allocate a block of memory for an array of `n` elements of type `int` by calling the `calloc()` function. The memory is initialized to all zeros by the latter.

In case of success the memory is released by the call to `free()`.

Section 10.10: Iterating through an array using pointers

```

#include <stdio.h>
#define SIZE (10)
int main()
{
    size_t i = 0;
    int *p = NULL;
    int a[SIZE];

    /* Setting up the values to be  $i \cdot i$  */
    for(i = 0; i < SIZE; ++i)
    {
        a[i] = i * i;
    }

    /* Reading the values using pointers */
    for(p = a; p < a + SIZE; ++p)
    {
        printf("%d\n", *p);
    }

    return 0;
}

```

Here, in the initialization of `p` in the first `for` loop condition, the array `a` *decays* to a pointer to its first element, as it would in almost all places where such an array variable is used.

Then, the `++p` performs pointer arithmetic on the pointer `p` and walks one by one through the elements of the

数组，并通过解引用*p来引用它们。

array, and refers to them by dereferencing them with *p.

第11章：链表

第11.1节：双向链表

一个示例代码，展示如何在双向链表中插入节点，如何轻松地反转链表，以及如何逆序打印链表。

```
#include <stdio.h>
#include <stdlib.h>

/* 这些数据不总是存储在结构体中，但有时为了方便会这样做 */
struct 节点 {
    /* 有时也会存储一个键，并在函数中使用 */
    int 数据;
    struct 节点* 下一个;
    struct 节点* 上一个;
};

void 在开头插入(struct 节点 **头节点指针, int 值);
void 在末尾插入(struct 节点 **头节点指针, int 值);

void print_list(struct Node *headNode);
void print_list_backwards(struct Node *headNode);

void free_list(struct Node *headNode);

int main(void) {
    /* 有时在双向链表中，最后一个节点也会被存储 */
    struct Node *head = NULL;

    printf("在链表开头插入一个节点。"); insert_at_beginning(&head, 5);
    print_list(head);

    printf("在开头插入一个节点，然后反向打印链表"); insert_at_beginning(&head, 10);
    print_list_backwards(head);

    printf("在链表末尾插入一个节点，然后正向打印链表。");

    insert_at_end(&head, 15);
    print_list(head);

    free_list(head);

    return 0;
}

void print_list_backwards(struct Node *headNode) {
    if (NULL == headNode)
    {
        return;
    }
    /*
    遍历列表，一旦到达末尾，反向遍历以倒序打印项目（这是通过指向前一个节点的指针完成的）。
    如果存储了指向最后一个节点的指针，这可以更容易地完成。
    */
    struct Node *i = headNode;
    while (i->next != NULL) {
```

Chapter 11: Linked lists

Section 11.1: A doubly linked list

An example of code showing how nodes can be inserted at a doubly linked list, how the list can easily be reversed, and how it can be printed in reverse.

```
#include <stdio.h>
#include <stdlib.h>

/* This data is not always stored in a structure, but it is sometimes for ease of use */
struct Node {
    /* Sometimes a key is also stored and used in the functions */
    int data;
    struct Node* next;
    struct Node* previous;
};

void insert_at_beginning(struct Node **pheadNode, int value);
void insert_at_end(struct Node **pheadNode, int value);

void print_list(struct Node *headNode);
void print_list_backwards(struct Node *headNode);

void free_list(struct Node *headNode);

int main(void) {
    /* Sometimes in a doubly linked list the last node is also stored */
    struct Node *head = NULL;

    printf("Insert a node at the beginning of the list.\n");
    insert_at_beginning(&head, 5);
    print_list(head);

    printf("Insert a node at the beginning, and then print the list backwards\n");
    insert_at_beginning(&head, 10);
    print_list_backwards(head);

    printf("Insert a node at the end, and then print the list forwards.\n");

    insert_at_end(&head, 15);
    print_list(head);

    free_list(head);

    return 0;
}

void print_list_backwards(struct Node *headNode) {
    if (NULL == headNode)
    {
        return;
    }
    /*
    Iterate through the list, and once we get to the end, iterate backwards to print
    out the items in reverse order (this is done with the pointer to the previous node).
    This can be done even more easily if a pointer to the last node is stored.
    */
    struct Node *i = headNode;
    while (i->next != NULL) {
```

```

i = i->next; /* 移动到列表末尾 */
}

while (i != NULL) {prin
    tf("Value: %d", i->data);    i = i->prev
ious;}
}
}

void print_list(struct Node *headNode) {
/* 遍历列表并打印每个节点的数据成员 */
struct 节点 *i;
for (i = 头节点; i != NULL; i = i->下一个) {printf("值: %
    d", i->数据);
}
}

```

```
void 在开头插入(struct 节点 **头节点指针, int 值) {
    struct 节点 *当前节点;
```

```

if (NULL == 头节点指针)
{
    return;
}
/*

```

这与我们在单链表开头插入节点的方式类似，区别是我们还设置了结构体的前驱成员

```

当前节点 = malloc(sizeof *当前节点);
```

```

    当前节点->下一个 = NULL;
    当前节点->前驱 = NULL;
    当前节点->数据 = 值;
```

```

if (*头节点指针 == NULL) { /* 链表为空 */
    *头节点指针 = 当前节点;
    return;
}
```

```

当前节点->下一个 = *头节点指针;
(*头节点指针)->前驱 = 当前节点;
*头节点指针 = 当前节点;
}
```

```
void 在末尾插入(struct 节点 **头节点指针, int 值) {
```

```

    struct 节点 *当前节点;

if (NULL == 头节点指针)
{
    return;
}
```

这可以通过能够访问前一个元素轻松完成。拥有指向最后一个节点的指针也会更加有用，这在实际中很常见。

```
*/
```

```

currentNode = malloc(sizeof *currentNode);
    struct Node *i = *pheadNode;
```

```
currentNode->data = value;
```

```

    i = i->next; /* Move to the end of the list */
}
```

```

while (i != NULL) {
    printf("Value: %d\n", i->data);
    i = i->previous;
}
}
```

```
void print_list(struct Node *headNode) {
/* Iterate through the list and print out the data member of each node */
struct Node *i;
for (i = headNode; i != NULL; i = i->next) {
    printf("Value: %d\n", i->data);
}
}
```

```
void insert_at_beginning(struct Node **pheadNode, int value) {
    struct Node *currentNode;
```

```

if (NULL == pheadNode)
{
    return;
}
/*

```

This is done similarly to how we insert a node at the beginning of a singly linked list, instead we set the previous member of the structure as well

```

*/
```

```
currentNode = malloc(sizeof *currentNode);
```

```

    currentNode->next = NULL;
    currentNode->previous = NULL;
    currentNode->data = value;
```

```

if (*pheadNode == NULL) { /* The list is empty */
    *pheadNode = currentNode;
    return;
}
```

```

    currentNode->next = *pheadNode;
    (*pheadNode)->previous = currentNode;
    *pheadNode = currentNode;
}
```

```
void insert_at_end(struct Node **pheadNode, int value) {
    struct Node *currentNode;
```

```

if (NULL == pheadNode)
{
    return;
}
```

/*
This can, again be done easily by being able to have the previous element. It would also be even more useful to have a pointer to the last node, which is commonly used.

```
*/
```

```

currentNode = malloc(sizeof *currentNode);
    struct Node *i = *pheadNode;
```

```
currentNode->data = value;
```

```

currentNode->next = NULL;
currentNode->previous = NULL;

if (*pheadNode == NULL) {
    *pheadNode = currentNode;
    return;
}

while (i->next != NULL) { /* 走到链表末尾 */
    i = i->next;
}

i->next = currentNode;
currentNode->previous = i;
}

void free_list(struct Node *node) {
    while (node != NULL) {
        struct Node *next = node->next;
        free(node);
        node = next;
    }
}

```

注意，有时存储指向最后一个节点的指针是有用的（直接跳到链表末尾比遍历到末尾更高效）：

```
struct Node *lastNode = NULL;
```

在这种情况下，需要在链表发生变化时更新它。

有时，键也用于标识元素。它只是 Node 结构体的一个成员：

```

struct Node {
    int data;
    int key;
    struct Node* next;
    struct Node* previous;
};

```

然后在对特定元素执行任何操作时（如删除元素），会使用该键。

第11.2节：反转链表

你也可以递归地执行此任务，但在本例中我选择使用迭代方法。如果你将所有节点都插入到链表的开头，这个任务非常有用。示例如下：

```

#include <stdio.h>
#include <stdlib.h>

#define NUM_ITEMS 10

struct Node {
    int 数据;
    struct Node *next;
};

void insert_node(struct Node **headNode, int nodeValue, int position);
void print_list(struct Node *headNode);

```

```

currentNode->next = NULL;
currentNode->previous = NULL;

if (*pheadNode == NULL) {
    *pheadNode = currentNode;
    return;
}

while (i->next != NULL) { /* Go to the end of the list */
    i = i->next;
}

i->next = currentNode;
currentNode->previous = i;
}

void free_list(struct Node *node) {
    while (node != NULL) {
        struct Node *next = node->next;
        free(node);
        node = next;
    }
}

```

Note that sometimes, storing a pointer to the last node is useful (it is more efficient to simply be able to jump straight to the end of the list than to need to iterate through to the end):

```
struct Node *lastNode = NULL;
```

In which case, updating it upon changes to the list is needed.

Sometimes, a key is also used to identify elements. It is simply a member of the Node structure:

```

struct Node {
    int data;
    int key;
    struct Node* next;
    struct Node* previous;
};

```

The key is then used when any tasks are performed on a specific element, like deleting elements.

Section 11.2: Reversing a linked list

You can also perform this task recursively, but I have chosen in this example to use an iterative approach. This task is useful if you are inserting all of your nodes at the beginning of a linked list. Here is an example:

```

#include <stdio.h>
#include <stdlib.h>

#define NUM_ITEMS 10

struct Node {
    int data;
    struct Node *next;
};

void insert_node(struct Node **headNode, int nodeValue, int position);
void print_list(struct Node *headNode);

```

```

void reverse_list(struct Node **headNode);

int main(void) {
    int i;
    struct Node *head = NULL;

    for(i = 1; i <= NUM_ITEMS; i++) {
        insert_node(&head, i, i);
    }
    print_list(head);

    printf("我现在将反转链表"); reverse_list(&head);

    print_list(head);
    return 0;
}

void print_list(struct Node *headNode) {
    struct Node *iterator;

    for(iterator = headNode; iterator != NULL; iterator = iterator->next) {printf("值: %d", iterator->data);
    }
}

void insert_node(struct Node **headNode, int nodeValue, int position) {
    int i;
    struct Node *currentNode = (struct Node *)malloc(sizeof(struct Node));
    struct Node *nodeBeforePosition = *headNode;

    currentNode->data = nodeValue;

    if(position == 1) {
        currentNode->next = *headNode;
        *headNode = currentNode;
        return;
    }

    for (i = 0; i < position - 2; i++) {
        nodeBeforePosition = nodeBeforePosition->next;
    }

    currentNode->next = nodeBeforePosition->next;
    nodeBeforePosition->next = currentNode;
}

void reverse_list(struct Node **headNode) {
    struct Node *iterator = *headNode;
    struct Node *previousNode = NULL;
    struct Node *nextNode = NULL;

    while (iterator != NULL) {
        nextNode = iterator->next;
        iterator->next = previousNode;
        previousNode = iterator;
        iterator = nextNode;
    }

    /* 迭代器最终将为 NULL, 因此最后一个节点将存储在 previousNode 中。我们将把最后一个节点设置为头节点 */
    *headNode = previousNode;
}

```

```

void reverse_list(struct Node **headNode);

int main(void) {
    int i;
    struct Node *head = NULL;

    for(i = 1; i <= NUM_ITEMS; i++) {
        insert_node(&head, i, i);
    }
    print_list(head);

    printf("I will now reverse the linked list\n");
    reverse_list(&head);
    print_list(head);
    return 0;
}

void print_list(struct Node *headNode) {
    struct Node *iterator;

    for(iterator = headNode; iterator != NULL; iterator = iterator->next) {
        printf("Value: %d\n", iterator->data);
    }
}

void insert_node(struct Node **headNode, int nodeValue, int position) {
    int i;
    struct Node *currentNode = (struct Node *)malloc(sizeof(struct Node));
    struct Node *nodeBeforePosition = *headNode;

    currentNode->data = nodeValue;

    if(position == 1) {
        currentNode->next = *headNode;
        *headNode = currentNode;
        return;
    }

    for (i = 0; i < position - 2; i++) {
        nodeBeforePosition = nodeBeforePosition->next;
    }

    currentNode->next = nodeBeforePosition->next;
    nodeBeforePosition->next = currentNode;
}

void reverse_list(struct Node **headNode) {
    struct Node *iterator = *headNode;
    struct Node *previousNode = NULL;
    struct Node *nextNode = NULL;

    while (iterator != NULL) {
        nextNode = iterator->next;
        iterator->next = previousNode;
        previousNode = iterator;
        iterator = nextNode;
    }

    /* Iterator will be NULL by the end, so the last node will be stored in previousNode. We will set the last node to be the headNode */
    *headNode = previousNode;
}

```

}

反向列表法说明

我们将previousNode初始化为NULL，因为我们知道在循环的第一次迭代中，如果我们正在寻找第一个头节点之前的节点，它将是NULL。第一个节点将成为列表中的最后一个节点，next变量自然应该是NULL。

基本上，这里反转链表的概念是我们实际上反转了链接本身。每个节点的next成员将变成它之前的节点，具体如下：

```
Head -> 1 -> 2 -> 3 -> 4 -> 5
```

其中每个数字代表一个节点。这个列表将变成：

```
1 <- 2 <- 3 <- 4 <- 5 <- Head
```

最后，头节点应该指向第5个节点，并且每个节点应该指向它之前的节点。

节点1应该指向NULL，因为它之前没有任何节点。节点2应该指向节点1，节点3应该指向节点2，依此类推。

然而，这种方法存在一个小问题。如果我们断开到下一个节点的链接并将其改为指向前一个节点，我们将无法遍历到列表中的下一个节点，因为指向它的链接已经消失。

解决这个问题的方法是，在更改链接之前，简单地将下一个元素存储在一个变量中 (nextNode)。

第11.3节：在第n个位置插入节点

到目前为止，我们已经看过如何在单链表的开头插入节点。然而，大多数情况下你也会希望能够在其他位置插入节点。下面的代码展示了如何编写一个insert()函数以在链表中的任意位置插入节点。

```
#include <stdio.h>
#include <stdlib.h>

struct 节点 {
    int 数据;
    struct 节点* next;
};

struct 节点* insert(struct 节点* head, int value, size_t position);
void print_list (struct 节点* head);

int main(int argc, char *argv[]) {
    struct 节点 *head = NULL; /* 初始化链表为空 */

    /* 在指定位置插入带有值的节点：*/
    head = insert(head, 1, 0);
    head = insert(head, 100, 1);
    head = insert(head, 21, 2);
    head = insert(head, 2, 3);
    head = insert(head, 5, 4);
    head = insert(head, 42, 2);

    print_list(head);
    return 0;
}
```

}

Explanation for the Reverse List Method

We start the previousNode out as NULL, since we know on the first iteration of the loop, if we are looking for the node before the first head node, it will be NULL. The first node will become the last node in the list, and the next variable should naturally be NULL.

Basically, the concept of reversing the linked list here is that we actually reverse the links themselves. Each node's next member will become the node before it, like so:

```
Head -> 1 -> 2 -> 3 -> 4 -> 5
```

Where each number represents a node. This list would become:

```
1 <- 2 <- 3 <- 4 <- 5 <- Head
```

Finally, the head should point to the 5th node instead, and each node should point to the node previous of it.

Node 1 should point to NULL since there was nothing before it. Node 2 should point to node 1, node 3 should point to node 2, et cetera.

However, there is *one small problem* with this method. If we break the link to the next node and change it to the previous node, we will not be able to traverse to the next node in the list since the link to it is gone.

The solution to this problem is to simply store the next element in a variable (nextNode) before changing the link.

Section 11.3: Inserting a node at the nth position

So far, we have looked at inserting a node at the beginning of a singly linked list. However, most of the times you will want to be able to insert nodes elsewhere as well. The code written below shows how it is possible to write an insert() function to insert nodes *anywhere* in the linked lists.

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

struct Node* insert(struct Node* head, int value, size_t position);
void print_list (struct Node* head);

int main(int argc, char *argv[]) {
    struct Node *head = NULL; /* Initialize the list to be empty */

    /* Insert nodes at positions with values: */
    head = insert(head, 1, 0);
    head = insert(head, 100, 1);
    head = insert(head, 21, 2);
    head = insert(head, 2, 3);
    head = insert(head, 5, 4);
    head = insert(head, 42, 2);

    print_list(head);
    return 0;
}
```

```

}

struct 节点* insert(struct 节点* head, int value, size_t position) {
    size_t i = 0;
    struct 节点 *currentNode;

    /* 创建我们的节点 */
    currentNode = malloc(sizeof *currentNode);
    /* 在此检查 malloc() 是否成功! */

    /* 赋值数据 */
    currentNode->data = value;

    /* 保存指向我们需要链接到新节点的"next"字段的指针。 */
    /* 通过将其初始化为 &head, 我们处理了在开头插入的情况。 */
    struct Node **nextForPosition = &head;
    /* 迭代以获取我们正在寻找的"next"字段。 */
    /* 注意：如果位置大于当前元素数量，则在末尾插入。 */
    for (i = 0; i < position && *nextForPosition != NULL; i++) {
        /* nextForPosition 指向节点的"next"字段。 */
        /* 因此 *nextForPosition 是指向下一个节点的指针。 */
        // 用指针更新为下一个节点的"next"字段。
        nextForPosition = &(*nextForPosition)->next;
    }

    /* 这里，我们通过解引用 nextForPosition（它指向我们想插入节点位置的节点的"next"字段）来获取指向下一个节点的链接（即我们新插入的节点应该指向的节点）。 */

    // 我们将此链接赋值给我们的 next 值。
    currentNode->next = *nextForPosition;

    /* 现在，我们想要修正新节点位置之前节点的链接：它将被更改为指向我们的新节点的指针。 */
    *nextForPosition = currentNode;

    return head;
}

```

```

void print_list (struct Node* head) {
    /* 遍历节点列表并打印每个节点中的数据 */
    struct Node* i = head;
    while (i != NULL) {
        printf("%d", i->data);
        i = i->next;
    }
}

```

第11.4节：在单链表开头插入节点

下面的代码将提示输入数字，并将它们依次添加到链表的开头。

```

/* 本程序演示在链表开头插入节点 */

#include <stdio.h>
#include <stdlib.h>

struct 节点 {
    int 数据;
    struct 节点* next;
};

```

```

}

struct Node* insert(struct Node* head, int value, size_t position) {
    size_t i = 0;
    struct Node *currentNode;

    /* Create our node */
    currentNode = malloc(sizeof *currentNode);
    /* Check for success of malloc() here! */

    /* Assign data */
    currentNode->data = value;

    /* Holds a pointer to the 'next' field that we have to link to the new node.
       By initializing it to &head we handle the case of insertion at the beginning. */
    struct Node **nextForPosition = &head;
    /* Iterate to get the 'next' field we are looking for.
       Note: Insert at the end if position is larger than current number of elements. */
    for (i = 0; i < position && *nextForPosition != NULL; i++) {
        /* nextForPosition is pointing to the 'next' field of the node.
           So *nextForPosition is a pointer to the next node.
           Update it with a pointer to the 'next' field of the next node. */
        nextForPosition = &(*nextForPosition)->next;
    }

    /* Here, we are taking the link to the next node (the one our newly inserted node should
       point to) by dereferencing nextForPosition, which points to the 'next' field of the node
       that is in the position we want to insert our node at.
       We assign this link to our next value. */
    currentNode->next = *nextForPosition;

    /* Now, we want to correct the link of the node before the position of our
       new node: it will be changed to be a pointer to our new node. */
    *nextForPosition = currentNode;

    return head;
}

void print_list (struct Node* head) {
    /* Go through the list of nodes and print out the data in each node */
    struct Node* i = head;
    while (i != NULL) {
        printf("%d\n", i->data);
        i = i->next;
    }
}

```

Section 11.4: Inserting a node at the beginning of a singly linked list

The code below will prompt for numbers and continue to add them to the beginning of a linked list.

```

/* This program will demonstrate inserting a node at the beginning of a linked list */

#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

```

```

void insert_node (struct Node **head, int nodeValue);
void print_list (struct Node *head);

int main(int argc, char *argv[]) {
    struct Node* headNode;
    headNode = NULL; /* 初始化我们的第一个节点指针为NULL。 */
    size_t listSize, i;
    do {
        printf("您想输入多少个数字？"); while(1 != scanf("%zu", &listSize));
        for (i = 0; i < listSize; i++) {
            int numToAdd;
            do {
                printf("请输入一个数字："); while (1 != scanf("%d", &numToAdd));
            } insert_node (&headNode, numToAdd); printf("插入节点后当前列表: ");
            print_list(headNode);
        }
    } return 0;
}

void print_list (struct Node *head) {
    struct node* currentNode = head;
    /* 遍历每个链结点。 */
    while (currentNode != NULL) {printf("值: %d", currentNode->data);
        currentNode = currentNode -> next;
    }

    void insert_node (struct Node **head, int nodeValue) {
        struct Node *currentNode = malloc(sizeof *currentNode);
        currentNode->data = nodeValue;
        currentNode->next = (*head);
        *head = currentNode;
    }
}

```

节点插入说明

为了理解如何在开头添加节点，让我们看看可能的情况：

1. 列表为空，因此我们需要添加一个新节点。在这种情况下，我们的内存结构如下，其中HEAD是一个指向第一个节点的指针：

| 头节点 | --> 空指针

这行代码 `currentNode->next = *headNode;` 会将 `currentNode->next` 的值赋为 `NULL`，因为 `headNode` 最初的值是 `NULL`。

现在，我们想将头节点指针设置为指向当前节点。

| 头节点 | --> |当前节点| --> 空指针 /* 头节点指向当前节点 */

```

void insert_node (struct Node **head, int nodeValue);
void print_list (struct Node *head);

int main(int argc, char *argv[]) {
    struct Node* headNode;
    headNode = NULL; /* Initialize our first node pointer to be NULL. */
    size_t listSize, i;
    do {
        printf("How many numbers would you like to input?\n");
        } while(1 != scanf("%zu", &listSize));
        for (i = 0; i < listSize; i++) {
            int numToAdd;
            do {
                printf("Enter a number:\n");
            } while (1 != scanf("%d", &numToAdd));
            insert_node (&headNode, numToAdd);
            printf("Current list after your inserted node: \n");
            print_list(headNode);
        }
    } return 0;
}

void print_list (struct Node *head) {
    struct node* currentNode = head;
    /* Iterate through each link. */
    while (currentNode != NULL) {
        printf("Value: %d\n", currentNode->data);
        currentNode = currentNode -> next;
    }

    void insert_node (struct Node **head, int nodeValue) {
        struct Node *currentNode = malloc(sizeof *currentNode);
        currentNode->data = nodeValue;
        currentNode->next = (*head);
        *head = currentNode;
    }
}

```

Explanation for the Insertion of Nodes

In order to understand how we add nodes at the beginning, let's take a look at possible scenarios:

1. The list is empty, so we need to add a new node. In which case, our memory looks like this where HEAD is a pointer to the first node:

| HEAD | --> NULL

The line `currentNode->next = *headNode;` will assign the value of `currentNode->next` to be `NULL` since `headNode` originally starts out at a value of `NULL`.

Now, we want to set our head node pointer to point to our current node.

|HEAD | --> |CURRENTNODE| --> NULL /* The head node points to the current node */

这通过 `*headNode = currentNode;` 来完成

2. 链表已经有数据；我们需要在开头添加一个新节点。为了简单起见，先从1个节点开始：

头节点 --> 第一个节点 --> 空指针

使用`currentNode->next= *headNode`时，数据结构如下所示：

`currentNode --> HEAD --> 指向第一个节点的指针 --> NULL`

显然需要修改，因为`*headNode`应该指向`currentNode`。

`HEAD -> currentNode -> 节点 -> NULL`

这通过 `*headNode = currentNode;` 来完成

This is done with `*headNode = currentNode;`

2. The list is already populated; we need to add a new node to the beginning. For the sake of simplicity, let's start out with 1 node:

`HEAD --> FIRST NODE --> NULL`

With `currentNode->next = *headNode`, the data structure looks like this:

`currentNode --> HEAD --> POINTER TO FIRST NODE --> NULL`

Which, obviously needs to be altered since `*headNode` should point to `currentNode`.

`HEAD -> currentNode -> NODE -> NULL`

This is done with `*headNode = currentNode;`

第12章：枚举

第12.1节：简单枚举

枚举是一种用户定义的数据类型，由整型常量组成，每个整型常量都有一个名称。关键字enum用于定义枚举数据类型。

如果你使用枚举(enum)而不是整数(int)或字符串(string)/字符指针(char*)，你可以增加编译时的检查，避免传入无效常量的错误，并且明确说明哪些值是合法使用的。

示例 1

```
enum color{ RED, GREEN, BLUE };
```

```
void printColor(enum color chosenColor)
{
    const char *color_name = "无效颜色";
    switch (chosenColor)
    {
        case RED:
            color_name = "红色";
            break;
        case GREEN:
            color_name = "绿色";
            break;
        case BLUE:
            color_name = "蓝色";
            break;
    }
    printf("%s", color_name);}
```

定义如下的主函数（例如）：

```
int main(){
    enum color chosenColor;
    printf("请输入一个介于0到2之间的数字");
    scanf("%d", (int*)&chosenColor);
    printColor(chosenColor);
    return 0;
}
```

版本 ≥ C99

示例 2

（此示例使用自C99起标准化的指定初始化器。）

```
enum week{ 周一, 周二, 周三, 周四, 周五, 周六, 周日 };
```

```
static const char* const dow[] = {
    [周一] = "Mon", [周二] = "Tue", [周三] = "Wed",
    [周四] = "Thu", [周五] = "Fri", [周六] = "Sat", [周日] = "Sun" };
```

```
void printDayOfWeek(enum 星期几)
{
    printf("%s", dow[day]);}
```

Chapter 12: Enumerations

Section 12.1: Simple Enumeration

An enumeration is a user-defined data type consists of integral constants and each integral constant is given a name. Keyword **enum** is used to define enumerated data type.

If you use **enum** instead of **int** or **string/ char***, you increase compile-time checking and avoid errors from passing in invalid constants, and you document which values are legal to use.

Example 1

```
enum color{ RED, GREEN, BLUE };
```

```
void printColor(enum color chosenColor)
{
    const char *color_name = "Invalid color";
    switch (chosenColor)
    {
        case RED:
            color_name = "RED";
            break;
        case GREEN:
            color_name = "GREEN";
            break;
        case BLUE:
            color_name = "BLUE";
            break;
    }
    printf("%s\n", color_name);}
```

With a main function defined as follows (for example):

```
int main(){
    enum color chosenColor;
    printf("Enter a number between 0 and 2");
    scanf("%d", (int*)&chosenColor);
    printColor(chosenColor);
    return 0;
}
```

Version ≥ C99

Example 2

(This example uses designated initializers which are standardized since C99.)

```
enum week{ MON, TUE, WED, THU, FRI, SAT, SUN };
```

```
static const char* const dow[] = {
    [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
    [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };
```

```
void printDayOfWeek(enum week day)
{
    printf("%s\n", dow[day]);}
```

使用范围检查的相同示例：

```
enum week{ DOW_INVALID = -1,
    MON, TUE, WED, THU, FRI, SAT, SUN,
    DOW_MAX };

static const char* const dow[] = {
    [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
    [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
{
    assert(day > DOW_INVALID && day < DOW_MAX);pr
    intf("%s", dow[day]);
}
```

第12.2节：无类型名的枚举常量

枚举类型也可以在不指定名称的情况下声明：

```
enum { bufsize = 256, };
static unsigned char buffer [bufsize] = { 0 };
```

这使我们能够定义编译时常量，类型为int，正如本例中可以用作数组长度。

第12.3节：具有重复值的枚举

枚举值不一定需要唯一：

```
#include <stdlib.h> /* 用于 EXIT_SUCCESS */
#include <stdio.h> /* 用于 printf() */

enum Dups
{
    Base, /* 取值为 0 */
    One, /* 取值为 Base + 1 */
    Two, /* 取值为 One + 1 */
    Negative = -1,
    AnotherZero /* 取值为 Negative + 1 == 0, 哀 */
};

int main(void)
{
    printf("Base = %d", Base);printf("O
ne = %d", One);printf("Two =
%d", Two);printf("Negative = %
d", Negative);printf("AnotherZero = %d", An
otherZero);return EXIT_SUCCESS;
}
```

示例输出：

```
Base = 0
One = 1
Two = 2
```

The same example using range checking:

```
enum week{ DOW_INVALID = -1,
    MON, TUE, WED, THU, FRI, SAT, SUN,
    DOW_MAX };

static const char* const dow[] = {
    [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
    [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
{
    assert(day > DOW_INVALID && day < DOW_MAX);
    printf("%s\n", dow[day]);
}
```

Section 12.2: enumeration constant without typename

Enumeration types can also be declared without giving them a name:

```
enum { bufsize = 256, };
static unsigned char buffer [bufsize] = { 0 };
```

This enables us to define compile time constants of type int that can as in this example be used as array length.

Section 12.3: Enumeration with duplicate value

An enumerations value in no way needs to be unique:

```
#include <stdlib.h> /* 用于 EXIT_SUCCESS */
#include <stdio.h> /* 用于 printf() */

enum Dups
{
    Base, /* Takes 0 */
    One, /* Takes Base + 1 */
    Two, /* Takes One + 1 */
    Negative = -1,
    AnotherZero /* Takes Negative + 1 == 0, sigh */
};

int main(void)
{
    printf("Base = %d\n", Base);
    printf("One = %d\n", One);
    printf("Two = %d\n", Two);
    printf("Negative = %d\n", Negative);
    printf("AnotherZero = %d\n", AnotherZero);

    return EXIT_SUCCESS;
}
```

The sample prints:

```
Base = 0
One = 1
Two = 2
```

```
Negative = -1  
AnotherZero = 0
```

第12.4节：typedef 枚举

给枚举命名有几种可能性和惯例。第一种是在 `enum` 关键字。

```
enum 颜色  
{  
    红色,  
    绿色,  
    蓝色  
};
```

然后必须始终使用关键字和标签来使用此枚举，如下所示：

```
enum 颜色 选定颜色 = 红色;
```

如果在声明 `enum` 时直接使用 `typedef`，则可以省略标签名，然后使用该类型时无需使用 `enum` 关键字：

```
typedef enum  
{  
    红色,  
    绿色,  
    蓝色  
}颜色;  
  
颜色 chosenColor = 红色;
```

但在后一种情况下，我们不能将其用作枚举颜色，因为我们在定义中没有使用标签名。一种常见的约定是两者都使用，这样同一个名称可以带或不带枚举关键字使用。这具有与 C++ 兼容的特殊优势

```
枚举颜色 /* 如第一个例子 */  
{  
    红色,  
    绿色,  
    蓝色  
};  
typedef 枚举颜色 颜色; /* 同名的 typedef */  
  
颜色 chosenColor = 红色;  
枚举颜色 defaultColor = 蓝色;
```

函数：

```
void 打印颜色()  
{  
    如果 (chosenColor == 红色)  
    {  
        printf("红色");  
    }  
    else if (chosenColor == 绿色)  
    {  
        printf("绿色");  
    }  
}
```

```
Negative = -1  
AnotherZero = 0
```

Section 12.4: Typedef enum

There are several possibilities and conventions to name an enumeration. The first is to use a *tag name* just after the `enum` keyword.

```
enum color  
{  
    RED,  
    GREEN,  
    BLUE  
};
```

This enumeration must then always be used with the keyword *and* the tag like this:

```
enum color chosenColor = RED;
```

If we use `typedef` directly when declaring the `enum`, we can omit the tag name and then use the type without the `enum` keyword:

```
typedef enum  
{  
    RED,  
    GREEN,  
    BLUE  
} color;  
  
color chosenColor = RED;
```

But in this latter case we cannot use it as `enum color`, because we didn't use the tag name in the definition. One common convention is to use both, such that the same name can be used with or without `enum` keyword. This has the particular advantage of being compatible with C++

```
enum color /* as in the first example */  
{  
    RED,  
    GREEN,  
    BLUE  
};  
typedef enum color color; /* also a typedef of same identifier */  
  
color chosenColor = RED;  
enum color defaultColor = BLUE;
```

Function:

```
void printColor()  
{  
    if (chosenColor == RED)  
    {  
        printf("RED\n");  
    }  
    else if (chosenColor == GREEN)  
    {  
        printf("GREEN\n");  
    }  
}
```

```
}

否则如果 (chosenColor == 蓝色)
{
    printf("蓝色");
}

}
```

有关 `typedef` 的更多内容请参见 [Typedef](#)

```
}

else if (chosenColor == BLUE)
{
    printf("BLUE\n");
}

}
```

For more on `typedef` see [Typedef](#)

第13章：结构体

结构体提供了一种将一组不同类型的相关变量组合成单个内存单元的方法。整个结构体可以通过一个名称或指针来引用；结构体成员也可以单独访问。结构体可以作为参数传递给函数，也可以从函数返回。它们使用关键字 `struct` 来定义。

第13.1节：灵活数组成员

版本 ≥ C99

类型声明

一个结构体 至少包含一个成员，且可以在结构体末尾额外包含一个长度未指定的数组成员。

这称为灵活数组成员：

```
struct ex1
{
    size_t foo;
    int flex[];
};

struct ex2_header
{
    int foo;
    char bar;
};

struct ex2
{
    struct ex2_header hdr;
    int flex[];
};

/* 合并的 ex2_header 和 ex2 结构体。*/
struct ex3
{
    int foo;
    char bar;
    int flex[];
};
```

大小和填充的影响

在计算结构体大小时，灵活数组成员被视为没有大小，尽管该成员与结构体前一个成员之间可能仍存在填充：

```
/* 在我的机器上打印“8,8”，所以没有填充。 */
printf("%zu,%zu", sizeof(size_t), sizeof(struct ex1));

/* 在我的机器上也打印“8,8”，所以ex2结构本身没有填充。 */
printf("%zu,%zu", sizeof(struct ex2_header), sizeof(struct ex2));

/* 在我的机器上打印“5,8”，所以有3个字节的填充。 */
printf("%zu,%zu", sizeof(int) + sizeof(char), sizeof(struct ex3));
```

灵活数组成员被视为不完整数组类型，因此其大小不能使用

`sizeof` 进行计算。

Chapter 13: Structs

Structures provide a way to group a set of related variables of diverse types into a single unit of memory. The structure as a whole can be referenced by a single name or pointer; the structure members can be accessed individually too. Structures can be passed to functions and returned from functions. They are defined using the keyword `struct`.

Section 13.1: Flexible Array Members

Version ≥ C99

Type Declaration

A structure *with at least one member* may additionally contain a single array member of unspecified length at the end of the structure. This is called a flexible array member:

```
struct ex1
{
    size_t foo;
    int flex[];
};

struct ex2_header
{
    int foo;
    char bar;
};

struct ex2
{
    struct ex2_header hdr;
    int flex[];
};

/* Merged ex2_header and ex2 structures. */
struct ex3
{
    int foo;
    char bar;
    int flex[];
};
```

Effects on Size and Padding

A flexible array member is treated as having no size when calculating the size of a structure, though padding between that member and the previous member of the structure may still exist:

```
/* Prints "8,8" on my machine, so there is no padding. */
printf("%zu,%zu\n", sizeof(size_t), sizeof(struct ex1));

/* Also prints "8,8" on my machine, so there is no padding in the ex2 structure itself. */
printf("%zu,%zu\n", sizeof(struct ex2_header), sizeof(struct ex2));

/* Prints "5,8" on my machine, so there are 3 bytes of padding. */
printf("%zu,%zu\n", sizeof(int) + sizeof(char), sizeof(struct ex3));
```

The flexible array member is considered to have an incomplete array type, so its size cannot be calculated using `sizeof`.

用法

您可以声明并初始化包含灵活数组成员的结构类型对象，但您不得尝试初始化灵活数组成员，因为它被视为不存在。尝试这样做是被禁止的，会导致编译错误。

同样，您不应尝试在以这种方式声明结构时为灵活数组成员的任何元素赋值，因为结构末尾可能没有足够的填充空间来容纳灵活数组成员所需的任何对象。编译器不一定会阻止您这样做，因此这可能导致未定义行为。

```
/* 无效：不能初始化灵活数组成员 */
struct ex1 e1 = {1, {2, 3}};
/* 无效：hdr={foo=1, bar=2} 可以，但不能初始化灵活数组成员 */
struct ex2 e2 = {{1, 2}, {3}};
/* 有效：初始化 foo=1, bar=2 成员 */
struct ex3 e3 = {1, 2};

e1.flex[0] = 3; /* 未定义行为，就我而言 */
e3.flex[0] = 2; /* 再次未定义行为 */
e2.flex[0] = e3.flex[0]; /* 未定义行为 */
```

您也可以选择使用malloc、calloc或realloc来分配带有额外存储空间的结构体，然后再释放它，这样您就可以按需使用灵活数组成员：

```
/* 有效：分配一个结构体类型 `ex1` 的对象以及一个包含 2 个整数的数组 */
struct ex1 *pe1 = malloc(sizeof(*pe1) + 2 * sizeof(pe1->flex[0]));

/* 有效：分配一个结构体类型 ex2 的对象以及一个包含 4 个整数的数组 */
struct ex2 *pe2 = malloc(sizeof(struct ex2) + sizeof(int[4]));

/* 有效：分配 5 个结构体类型 ex3 的对象，每个对象附带一个包含 3 个整数的数组 */
struct ex3 *pe3 = malloc(5 * (sizeof(*pe3) + sizeof(int[3])));

pe1->flex[0] = 3; /* 有效 */
pe3[0]->flex[0] = pe1->flex[0]; /* 有效 */
```

版本 < C99

“结构体技巧”

在 C99 之前，灵活数组成员不存在且被视为错误。一个常见的解决方法是声明一个长度为 1 的数组，这种技术称为“结构体技巧”：

```
struct ex1
{
    size_t foo;
    int flex[1];
};
```

这将影响结构体的大小，然而，与真正的灵活数组成员不同：

```
/* 在我的机器上打印“8,4,16”，表示有 4 字节的填充。 */
printf("%d,%d,%d", (int)sizeof(size_t), (int)sizeof(int[1]), (int)sizeof(struct ex1));
```

要将flex成员用作灵活数组成员，您应如上所示使用malloc进行分配，只是sizeof(*pe1)（或等价的sizeof(struct ex1））应替换为offsetof(struct ex1, flex)，或者使用更长的、类型无关的表达式sizeof(*pe1)-sizeof(pe1->flex)。或者，您也可以从所需的“灵活”数组长度中减去1，因为它已经包含在结构体大小中，前提是所需长度大于0。

Usage

You can declare and initialize an object with a structure type containing a flexible array member, but you must not attempt to initialize the flexible array member since it is treated as if it does not exist. It is forbidden to try to do this, and compile errors will result.

Similarly, you should not attempt to assign a value to any element of a flexible array member when declaring a structure in this way since there may not be enough padding at the end of the structure to allow for any objects required by the flexible array member. The compiler will not necessarily prevent you from doing this, however, so this can lead to undefined behavior.

```
/* invalid: cannot initialize flexible array member */
struct ex1 e1 = {1, {2, 3}};
/* invalid: hdr={foo=1, bar=2} OK, but cannot initialize flexible array member */
struct ex2 e2 = {{1, 2}, {3}};
/* valid: initialize foo=1, bar=2 members */
struct ex3 e3 = {1, 2};

e1.flex[0] = 3; /* undefined behavior, in my case */
e3.flex[0] = 2; /* undefined behavior again */
e2.flex[0] = e3.flex[0]; /* undefined behavior */
```

You may instead choose to use `malloc`, `calloc`, or `realloc` to allocate the structure with extra storage and later free it, which allows you to use the flexible array member as you wish:

```
/* valid: allocate an object of structure type `ex1` along with an array of 2 ints */
struct ex1 *pe1 = malloc(sizeof(*pe1) + 2 * sizeof(pe1->flex[0]));

/* valid: allocate an object of structure type ex2 along with an array of 4 ints */
struct ex2 *pe2 = malloc(sizeof(struct ex2) + sizeof(int[4]));

/* valid: allocate 5 structure type ex3 objects along with an array of 3 ints per object */
struct ex3 *pe3 = malloc(5 * (sizeof(*pe3) + sizeof(int[3])));

pe1->flex[0] = 3; /* valid */
pe3[0]->flex[0] = pe1->flex[0]; /* valid */
```

Version < C99

The 'struct hack'

Flexible array members did not exist prior to C99 and are treated as errors. A common workaround is to declare an array of length 1, a technique called the 'struct hack':

```
struct ex1
{
    size_t foo;
    int flex[1];
};
```

This will affect the size of the structure, however, unlike a true flexible array member:

```
/* Prints "8,4,16" on my machine, signifying that there are 4 bytes of padding. */
printf("%d,%d,%d\n", (int)sizeof(size_t), (int)sizeof(int[1]), (int)sizeof(struct ex1));
```

To use the `flex` member as a flexible array member, you'd allocate it with `malloc` as shown above, except that `sizeof(*pe1)` (or the equivalent `sizeof(struct ex1)`) would be replaced with `offsetof(struct ex1, flex)` or the longer, type-agnostic expression `sizeof(*pe1)-sizeof(pe1->flex)`. Alternatively, you might subtract 1 from the desired length of the "flexible" array since it's already included in the structure size, assuming the desired length is

相同的逻辑也可以应用于其他使用示例。

兼容性

如果希望兼容不支持灵活数组成员的编译器，可以使用如下定义的宏FLEXMEMB_SIZE：

```
#if __STDC_VERSION__ < 199901L
#define FLEXMEMB_SIZE 1
#else
#define FLEXMEMB_SIZE /* nothing */
#endif

struct ex1
{
    size_t foo;
    int flex[FLEXMEMB_SIZE];
};
```

在分配对象时，应使用`offsetof(struct ex1, flex)`形式来引用结构体大小（不包括灵活数组成员），因为这是在支持灵活数组成员的编译器和不支持的编译器之间保持一致的唯一表达式：

```
struct ex1 *pe10 = malloc(offsetof(struct ex1, flex) + n * sizeof(pe10->flex[0]));
```

另一种方法是使用预处理器有条件地从指定长度中减去1。由于这种形式增加了不一致和人为错误的可能性，我将逻辑移到了一个单独的函数中：

```
struct ex1 *ex1_alloc(size_t n)
{
    struct ex1 tmp;
#if __STDC_VERSION__ < 199901L
    if (n != 0)
        n--;
#endif
    return malloc(sizeof(tmp) + n * sizeof(tmp.flex[0]));
}
...

/* 分配一个"flex"数组长度为3的ex1对象 */
struct ex1 *pe1 = ex1_alloc(3);
```

第13.2节：typedef结构体

将typedef与struct结合使用可以使代码更清晰。例如：

```
typedef 结构体
{
    int x, y;
} Point;
```

与之相对的是：

```
结构体 Point
{
    int x, y;
};
```

greater than 0. The same logic may be applied to the other usage examples.

Compatibility

If compatibility with compilers that do not support flexible array members is desired, you may use a macro defined like FLEXMEMB_SIZE below:

```
#if __STDC_VERSION__ < 199901L
#define FLEXMEMB_SIZE 1
#else
#define FLEXMEMB_SIZE /* nothing */
#endif

struct ex1
{
    size_t foo;
    int flex[FLEXMEMB_SIZE];
};
```

When allocating objects, you should use the `offsetof(struct ex1, flex)` form to refer to the structure size (excluding the flexible array member) since it is the only expression that will remain consistent between compilers that support flexible array members and compilers that do not:

```
struct ex1 *pe10 = malloc(offsetof(struct ex1, flex) + n * sizeof(pe10->flex[0]));
```

The alternative is to use the preprocessor to conditionally subtract 1 from the specified length. Due to the increased potential for inconsistency and general human error in this form, I moved the logic into a separate function:

```
struct ex1 *ex1_alloc(size_t n)
{
    struct ex1 tmp;
#if __STDC_VERSION__ < 199901L
    if (n != 0)
        n--;
#endif
    return malloc(sizeof(tmp) + n * sizeof(tmp.flex[0]));
}
...

/* allocate an ex1 object with "flex" array of length 3 */
struct ex1 *pe1 = ex1_alloc(3);
```

Section 13.2: Typedef Structs

Combining `typedef` with `struct` can make code clearer. For example:

```
typedef struct
{
    int x, y;
} Point;
```

as opposed to:

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

可以声明为：

```
Point point;
```

而不是：

```
struct Point point;
```

更好的做法是使用以下方式

```
typedef struct Point Point;  
  
struct Point  
{  
    int x, y;  
};
```

为了同时利用“点 (point) ”的两种可能定义。这种声明在你先学过C++时最为方便，因为如果名称不模糊，你可以省略“`struct`”关键字。

结构体的`typedef`名称可能会与程序其他部分的标识符冲突。有些人认为这是一个缺点，但对大多数人来说，结构体名和其他标识符相同是相当令人困扰的。

臭名昭著的例子是POSIX的“`stat`”

```
int stat(const char *pathname, struct stat *buf);
```

你会看到一个名为`stat`的函数，它的一个参数是`struct stat`。

没有标签名的`typedef`结构体总是要求整个结构体声明对使用它的代码可见。整个结构体声明必须放在头文件中。

考虑：

```
#include "bar.h"  
  
struct foo  
{  
    bar *aBar;  
};
```

因此，对于没有标签名的`typedef`结构体，`bar.h`文件必须始终包含`bar`的完整定义。如果我们使用

```
typedef struct bar bar;
```

在`bar.h`中，`bar`结构的细节可以被隐藏。

参见 [Typedef](#)

第13.3节：指向结构体的指针

当你有一个包含`struct`的变量时，可以使用点操作符`(.)`访问其字段。然而，如果你有一个指向`struct`的指针，这种方法将不起作用。你必须使用箭头操作符`(->)`来访问其字段。下面是一个非常简单（有些人可能称之为“糟糕且简单”）的栈实现示例，使用了指向`struct`的指针，并演示了箭头操作符的用法。

could be declared as:

```
Point point;
```

instead of:

```
struct Point point;
```

Even better is to use the following

```
typedef struct Point Point;  
  
struct Point  
{  
    int x, y;  
};
```

to have advantage of both possible definitions of `point`. Such a declaration is most convenient if you learned C++ first, where you may omit the `struct` keyword if the name is not ambiguous.

`typedef` names for structs could be in conflict with other identifiers of other parts of the program. Some consider this a disadvantage, but for most people having a `struct` and another identifier the same is quite disturbing. Notorious is e.g POSIX' `stat`

```
int stat(const char *pathname, struct stat *buf);
```

where you see a function `stat` that has one argument that is `struct stat`.

`typedef'd` structs without a tag name always impose that the whole `struct` declaration is visible to code that uses it. The entire `struct` declaration must then be placed in a header file.

Consider:

```
#include "bar.h"  
  
struct foo  
{  
    bar *aBar;  
};
```

So with a `typedef struct` that has no tag name, the `bar.h` file always has to include the whole definition of `bar`. If we use

```
typedef struct bar bar;
```

in `bar.h`, the details of the `bar` structure can be hidden.

See [Typedef](#)

Section 13.3: Pointers to structs

When you have a variable containing a `struct`, you can access its fields using the dot operator`(.)`. However, if you have a pointer to a `struct`, this will not work. You have to use the arrow operator`(->)` to access its fields. Here's an example of a terribly simple (some might say "terrible and simple") implementation of a stack that uses pointers to `structs` and demonstrates the arrow operator.

```

#include <stdlib.h>
#include <stdio.h>

/* 结构体 */
struct stack
{
    struct node *top;
    int size;
};

结构体节点
{
    int 数据;
    struct node *next;
};

/* 函数声明 */
int push(int, struct stack*);
int pop(struct stack*);
void destroy(struct stack*);

int main(void)
{
    int result = EXIT_SUCCESS;

    size_t i;

    /* 为 struct stack 分配内存并记录其指针 */
    struct stack *stack = malloc(sizeof *stack);
    if (NULL == stack)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    /* 初始化栈 */
    stack->top = NULL;
    stack->size = 0;

    /* 压入10个整数 */
    {
        int data = 0;
        for(i = 0; i < 10; i++)
        {
            printf("Pushing: %d", data);if (-1 ==
push(data, stack)){
                perror("push() failed");
                result = EXIT_FAILURE;
                break;
            }

            ++data;
        }
    }

    if (EXIT_SUCCESS == result)
    {
        /* 弹出5个整数 */
        for(i = 0; i < 5; i++)
        {
            printf("Popped: %i", pop(stack));
        }
    }
}

```

```

#include <stdlib.h>
#include <stdio.h>

/* structs */
struct stack
{
    struct node *top;
    int size;
};

struct node
{
    int data;
    struct node *next;
};

/* function declarations */
int push(int, struct stack*);
int pop(struct stack*);
void destroy(struct stack*);

int main(void)
{
    int result = EXIT_SUCCESS;

    size_t i;

    /* allocate memory for a struct stack and record its pointer */
    struct stack *stack = malloc(sizeof *stack);
    if (NULL == stack)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    /* initialize stack */
    stack->top = NULL;
    stack->size = 0;

    /* push 10 ints */
    {
        int data = 0;
        for(i = 0; i < 10; i++)
        {
            printf("Pushing: %d\n", data);
            if (-1 == push(data, stack))
            {
                perror("push() failed");
                result = EXIT_FAILURE;
                break;
            }

            ++data;
        }
    }

    if (EXIT_SUCCESS == result)
    {
        /* pop 5 ints */
        for(i = 0; i < 5; i++)
        {
            printf("Popped: %i\n", pop(stack));
        }
    }
}

```

```

    }

    /* 销毁栈 */
destroy(stack);

    return result;
}

/* 将一个值压入栈中。 */
/* 成功返回0，失败返回-1。 */
int push(int data, struct stack *stack)
{
    int result = 0;

    /* 为新节点分配内存 */
    struct node *new_node = malloc(sizeof *new_node);
    if (NULL == new_node)
    {
        result = -1;
    }
    else
    {
        new_node->data = data;
        new_node->next = stack->top;
        stack->top = new_node;
        stack->size++;
    }

    return result;
}

/* 从栈中弹出一个值。 */
/* 返回从栈中弹出的值 */
int pop(struct stack *stack)
{
    struct node *top = stack->top;
    int data = top->data;
    stack->top = top->next;
    stack->size--;
    free(top);
    return data;
}

/* 销毁栈 */
void destroy(struct stack *stack)
{
    /* 释放所有指针 */
    while(stack->top != NULL)
    {
        pop(stack);
    }
}

```

第13.4节：将结构体传递给函数

在C语言中，所有参数都是按值传递给函数的，包括结构体。对于小型结构体来说，这是件好事，因为这意味着通过指针访问数据不会产生额外开销。然而，这也很容易导致意外传递一个巨大的结构体，从而导致性能下降，特别是当程序员习惯于其他按引用传递参数的语言时。

```

    }

    /* destroy stack */
destroy(stack);

    return result;
}

/* Push a value onto the stack. */
/* Returns 0 on success and -1 on failure. */
int push(int data, struct stack *stack)
{
    int result = 0;

    /* allocate memory for new node */
    struct node *new_node = malloc(sizeof *new_node);
    if (NULL == new_node)
    {
        result = -1;
    }
    else
    {
        new_node->data = data;
        new_node->next = stack->top;
        stack->top = new_node;
        stack->size++;
    }

    return result;
}

/* Pop a value off of the stack. */
/* Returns the value popped off the stack */
int pop(struct stack *stack)
{
    struct node *top = stack->top;
    int data = top->data;
    stack->top = top->next;
    stack->size--;
    free(top);
    return data;
}

/* destroy the stack */
void destroy(struct stack *stack)
{
    /* free all pointers */
    while(stack->top != NULL)
    {
        pop(stack);
    }
}

```

Section 13.4: Passing structs to functions

In C, all arguments are passed to functions by value, including structs. For small structs, this is a good thing as it means there is no overhead from accessing the data through a pointer. However, it also makes it very easy to accidentally pass a huge struct resulting in poor performance, particularly if the programmer is used to other languages where arguments are passed by reference.

```

struct 坐标
{
    int x;
    int y;
    int z;
};

// 按值传递和返回一个小型结构体，速度非常快
struct 坐标 move(struct 坐标 位置, struct 坐标 移动)
{
    位置.x += 移动.x;
    位置.y += 移动.y;
    位置.z += 移动.z;
    return 位置;
}

// 一个非常大的结构体
struct 大量数据
{
    int 参数1;
    char 参数2[80000];
};

// 通过值传递和返回大型结构体，非常慢！
// 由于结构体体积庞大，这甚至可能导致栈溢出
struct lotsOfData doubleParam1(struct lotsOfData value)
{
    value.param1 *= 2;
    return value;
}

// 改为通过指针传递大型结构体，速度相当快
void doubleParam1ByPtr(struct lotsOfData *value)
{
    value->param1 *= 2;
}

```

```

struct coordinates
{
    int x;
    int y;
    int z;
};

// Passing and returning a small struct by value, very fast
struct coordinates move(struct coordinates position, struct coordinates movement)
{
    position.x += movement.x;
    position.y += movement.y;
    position.z += movement.z;
    return position;
}

// A very big struct
struct lotsOfData
{
    int param1;
    char param2[80000];
};

// Passing and returning a large struct by value, very slow!
// Given the large size of the struct this could even cause stack overflow
struct lotsOfData doubleParam1(struct lotsOfData value)
{
    value.param1 *= 2;
    return value;
}

// Passing the large struct by pointer instead, fairly fast
void doubleParam1ByPtr(struct lotsOfData *value)
{
    value->param1 *= 2;
}

```

第13.5节：使用结构体的基于对象的编程

结构体可以用来以面向对象的方式实现代码。结构体类似于类，但缺少通常也构成类一部分的函数，我们可以将这些函数作为函数指针成员变量添加。以我们的坐标示例为例：

```

/* coordinates.h */

typedef struct coordinate_s
{
    /* 指向方法函数的指针 */
    void (*setx)(coordinate *this, int x);
    void (*sety)(coordinate *this, int y);
    void (*print)(coordinate *this);
    /* 数据 */
    int x;
    int y;
} coordinate;

/* 构造函数 */
coordinate *coordinate_create(void);
/* 析构函数 */
void coordinate_destroy(coordinate *this);

```

Section 13.5: Object-based programming using structs

Structs may be used to implement code in an object oriented manner. A struct is similar to a class, but is missing the functions which normally also form part of a class, we can add these as function pointer member variables. To stay with our coordinates example:

```

/* coordinates.h */

typedef struct coordinate_s
{
    /* Pointers to method functions */
    void (*setx)(coordinate *this, int x);
    void (*sety)(coordinate *this, int y);
    void (*print)(coordinate *this);
    /* Data */
    int x;
    int y;
} coordinate;

/* Constructor */
coordinate *coordinate_create(void);
/* Destructor */
void coordinate_destroy(coordinate *this);

```

现在是实现的C文件：

```
/* coordinates.c */

#include "coordinates.h"
#include <stdio.h>
#include <stdlib.h>

/* 构造函数 */
coordinate *coordinate_create(void)
{
    coordinate *c = malloc(sizeof(*c));
    if (c != 0)
    {
        c->setx = &coordinate_setx;
        c->sety = &coordinate_sety;
        c->print = &coordinate_print;
        c->x = 0;
        c->y = 0;
    }
    return c;
}

/* 析构函数 */
void coordinate_destroy(coordinate *this)
{
    if (this != NULL)
    {
        free(this);
    }
}

/* 方法 */
static void coordinate_setx(coordinate *this, int x)
{
    if (this != NULL)
    {
        this->x = x;
    }
}

static void coordinate_sety(coordinate *this, int y)
{
    if (this != NULL)
    {
        this->y = y;
    }
}

static void coordinate_print(coordinate *this)
{
    if (this != NULL)
    {
        printf("坐标: (%i, %i)", this->x, this->y);
    }
    else
    {
        printf("空指针异常!");
    }
}
```

And now the implementing C file:

```
/* coordinates.c */

#include "coordinates.h"
#include <stdio.h>
#include <stdlib.h>

/* Constructor */
coordinate *coordinate_create(void)
{
    coordinate *c = malloc(sizeof(*c));
    if (c != 0)
    {
        c->setx = &coordinate_setx;
        c->sety = &coordinate_sety;
        c->print = &coordinate_print;
        c->x = 0;
        c->y = 0;
    }
    return c;
}

/* Destructor */
void coordinate_destroy(coordinate *this)
{
    if (this != NULL)
    {
        free(this);
    }
}

/* Methods */
static void coordinate_setx(coordinate *this, int x)
{
    if (this != NULL)
    {
        this->x = x;
    }
}

static void coordinate_sety(coordinate *this, int y)
{
    if (this != NULL)
    {
        this->y = y;
    }
}

static void coordinate_print(coordinate *this)
{
    if (this != NULL)
    {
        printf("Coordinate: (%i, %i)\n", this->x, this->y);
    }
    else
    {
        printf("NULL pointer exception!\n");
    }
}
```

我们坐标类的一个示例用法如下：

```
/* main.c */

#include "coordinates.h"
#include <stddef.h>

int main(void)
{
    /* 创建并初始化指向坐标对象的指针 */
    coordinate *c1 = coordinate_create();
    coordinate *c2 = coordinate_create();

    /* 现在我们可以使用方法并传入对象作为参数来操作对象 */
    c1->setx(c1, 1);
    c1->sety(c1, 2);

    c2->setx(c2, 3);
    c2->sety(c2, 4);

    c1->print(c1);
    c2->print(c2);

    /* 使用完对象后，我们使用“析构函数”销毁它们 */
    coordinate_destroy(c1);
    c1 = NULL;
    coordinate_destroy(c2);
    c2 = NULL;

    return 0;
}
```

第13.6节：简单数据结构

结构数据类型是一种将相关数据打包并使其表现得像单个变量的有用方式。

声明一个简单的结构体，包含两个整型成员：

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

x和y被称为point结构体的成员（或字段）。

定义和使用结构体：

```
struct point p; // 声明p为一个point结构体
p.x = 5;        // 给p的成员变量赋值
p.y = 3;
```

结构体可以在定义时初始化。上述等同于：

```
struct point p = {5, 3};
```

结构体也可以使用指定初始化器进行初始化。

An example usage of our coordinate class would be:

```
/* main.c */

#include "coordinates.h"
#include <stddef.h>

int main(void)
{
    /* Create and initialize pointers to coordinate objects */
    coordinate *c1 = coordinate_create();
    coordinate *c2 = coordinate_create();

    /* Now we can use our objects using our methods and passing the object as parameter */
    c1->setx(c1, 1);
    c1->sety(c1, 2);

    c2->setx(c2, 3);
    c2->sety(c2, 4);

    c1->print(c1);
    c2->print(c2);

    /* After using our objects we destroy them using our "destructor" function */
    coordinate_destroy(c1);
    c1 = NULL;
    coordinate_destroy(c2);
    c2 = NULL;

    return 0;
}
```

Section 13.6: Simple data structures

Structure data types are useful way to package related data and have them behave like a single variable.

Declaring a simple `struct` that holds two `int` members:

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

x and y are called the *members* (or *fields*) of point struct.

Defining and using structs:

```
struct point p; // declare p as a point struct
p.x = 5;        // assign p member variables
p.y = 3;
```

Structs can be initialized at definition. The above is equivalent to:

```
struct point p = {5, 3};
```

Structs may also be initialized using designated initializers.

访问字段也使用.操作符

```
printf("point is (x = %d, y = %d)", p.x, p.y);
```

Accessing fields is also done using the . operator

```
printf("point is (x = %d, y = %d)", p.x, p.y);
```

第14章：标准数学

第14.1节：幂函数 - pow(), powf(), powl()

下面的示例代码使用pow()系列标准数学库计算 $1+4(3+3^2+3^3+3^4+\dots+3^N)$ 数列的和。

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <fenv.h>

int main()
{
    double pwr, sum=0;
    int i, n;

    printf("1+4(3+3^2+3^3+3^4+...+3^N)=?请输入 N:");
    scanf("%d", &n);

    if (n<=0) {
        printf("无效的幂 N=%d", n);
        return -1;
    }

    for (i=0; i<n+1; i++) {
        errno = 0;
        feclearexcept(FE_ALL_EXCEPT);
        pwr = powl(3, i);
        if (fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW |
                         FE_UNDERFLOW)) {
            perror("数学错误");
        }
        sum += i ? pwr : 0;
        printf("N= %d S= %g", i, 1+4*sum);
    }

    return 0;
}
```

示例输出：

```
1+4(3+3^2+3^3+3^4+...+3^N)=?
输入 N:10
N= 0      S= 1
N= 1      S= 13
N= 2      S= 49
N= 3      S= 157
N= 4      S= 481
N= 5      S= 1453
N= 6      S= 4369
N= 7      S= 13117
N= 8      S= 39361
N= 9      S= 118093
N= 10     S= 354289
```

Chapter 14: Standard Math

Section 14.1: Power functions - pow(), powf(), powl()

The following example code computes the sum of $1+4(3+3^2+3^3+3^4+\dots+3^N)$ series using pow() family of standard math library.

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <fenv.h>

int main()
{
    double pwr, sum=0;
    int i, n;

    printf("1+4(3+3^2+3^3+3^4+...+3^N)=?Enter N:");
    scanf("%d", &n);
    if (n<=0) {
        printf("Invalid power N=%d", n);
        return -1;
    }

    for (i=0; i<n+1; i++) {
        errno = 0;
        feclearexcept(FE_ALL_EXCEPT);
        pwr = powl(3, i);
        if (fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW |
                         FE_UNDERFLOW)) {
            perror("Math Error");
        }
        sum += i ? pwr : 0;
        printf("N= %d S= %g\n", i, 1+4*sum);
    }

    return 0;
}
```

Example Output:

```
1+4(3+3^2+3^3+3^4+...+3^N)=?
Enter N:10
N= 0      S= 1
N= 1      S= 13
N= 2      S= 49
N= 3      S= 157
N= 4      S= 481
N= 5      S= 1453
N= 6      S= 4369
N= 7      S= 13117
N= 8      S= 39361
N= 9      S= 118093
N= 10     S= 354289
```

第14.2节：双精度浮点余数： fmod()

此函数返回 x/y 除法的浮点余数。返回值的符号与 x 相同。

```
#include <math.h> /* 用于 fmod() */
#include <stdio.h> /* 用于 printf() */

int main(void)
{
    double x = 10.0;
    double y = 5.1;

    double modulus = fmod(x, y);printf("%lf", modulus); /* f 与 lf 相同。 */return 0;
}
```

输出：

4.90000

重要提示： 使用此函数时需谨慎，因为浮点数运算可能导致返回意外的值。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    printf("%f", fmod(1, 0.1));printf("%1
9.17f", fmod(1, 0.1));return 0;
}
```

输出：

0.1
0.0999999999999995

第14.3节：单精度和长双精度浮点余数：fmodf(), fmodl()

版本 ≥ C99

这些函数返回 x/y 除法的浮点余数。返回值与

单精度：

```
#include <math.h> /* 用于 fmodf() */
#include <stdio.h> /* 用于 printf() */
```

Section 14.2: Double precision floating-point remainder: fmod()

This function returns the floating-point remainder of the division of x/y . The returned value has the same sign as x .

```
#include <math.h> /* for fmod() */
#include <stdio.h> /* for printf() */

int main(void)
{
    double x = 10.0;
    double y = 5.1;

    double modulus = fmod(x, y);

    printf("%lf\n", modulus); /* f is the same as lf. */

    return 0;
}
```

Output:

4.90000

Important: Use this function with care, as it can return unexpected values due to the operation of floating point values.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    printf("%f\n", fmod(1, 0.1));
    printf("%19.17f\n", fmod(1, 0.1));
    return 0;
}
```

Output:

0.1
0.0999999999999995

Section 14.3: Single precision and long double precision floating-point remainder: fmodf(), fmodl()

Version ≥ C99

These functions returns the floating-point remainder of the division of x/y . The returned value has the same sign as x .

Single Precision:

```
#include <math.h> /* for fmodf() */
#include <stdio.h> /* for printf() */
```

```

int main(void)
{
    float x = 10.0;
    float y = 5.1;

    float modulus = fmodf(x, y);

    printf("%f", modulus); /* If 也可以, 因为 modulus 会被提升为 double. */
}

```

输出：

4.90000

双倍双精度：

```

#include <math.h> /* 用于 fmodl() */
#include <stdio.h> /* 用于 printf() */

int main(void)
{
    long double x = 10.0;
    long double y = 5.1;

    long double modulus = fmodl(x, y);printf
        ("%Lf", modulus); /* Lf 用于 long double. */
}

```

输出：

4.90000

```

int main(void)
{
    float x = 10.0;
    float y = 5.1;

    float modulus = fmodf(x, y);

    printf("%f\n", modulus); /* If would do as well as modulus gets promoted to double. */
}

```

Output:

4.90000

Double Double Precision:

```

#include <math.h> /* for fmodl() */
#include <stdio.h> /* for printf() */

int main(void)
{
    long double x = 10.0;
    long double y = 5.1;

    long double modulus = fmodl(x, y);

    printf("%Lf\n", modulus); /* Lf is for long double. */
}

```

Output:

4.90000

第15章：迭代语句/循环： for, while, do-while

第15.1节：for 循环

为了反复执行一段代码，循环语句应运而生。当需要执行固定次数的代码块时，应使用for循环。例如，为了用用户输入填充大小为 n 的数组，我们需要执行 `scanf()` 共 n 次。

版本 ≥ C99

```
#include <stddef.h>          // 用于 size_t

int array[10];                // 10个整数的数组

for (size_t i = 0; i < 10; i++) // i 从0开始，到9结束
{
    scanf("%d", &array[i]);
}
```

通过这种方式，`scanf()`函数调用被执行了 n 次（在我们的例子中是10次），但只写了一次。

这里，变量 `i` 是循环索引，最好按示例中那样声明。类型 `size_t` (`size` 类型) 应用于所有计数或遍历数据对象的情况。

这种在 `for` 内声明变量的方式仅适用于已更新到 C99 标准的编译器。如果由于某种原因你仍然使用较旧的编译器，可以在 `for` 循环之前声明循环索引：

版本 < C99

```
#include <stddef.h>          /* 用于 size_t */
size_t i;
int array[10];                /* 10 个 int 的数组 */

for (i = 0; i < 10; i++)      /* i 从 0 开始，到 9 结束 */
{
    scanf("%d", &array[i]);
}
```

第15.2节：循环展开与Duff's Device

有时，直接的循环无法完全包含在循环体内。这是因为循环需要先由一些语句 `B` 进行初始化。然后，迭代以一些语句 `A` 开始，接着再次执行 `B`，之后才进入下一次循环。

```
do_B();
while (condition) {
    do_A();
    do_B();
}
```

为了避免代码中重复出现 `B` 两次而导致的潜在复制粘贴问题，可以应用 Duff's Device，从 `while` 循环体的中间开始循环，使用 `switch` 语句和贯穿行为。

```
switch (true) while (condition) {
case false: do_A(); /* 贯穿 */
```

Chapter 15: Iteration Statements/Loops: for, while, do-while

Section 15.1: For loop

In order to execute a block of code over and over again, loops comes into the picture. The `for` loop is to be used when a block of code is to be executed a fixed number of times. For example, in order to fill an array of size `n` with the user inputs, we need to execute `scanf()` for `n` times.

Version ≥ C99

```
#include <stddef.h>          // for size_t

int array[10];                // array of 10 int

for (size_t i = 0; i < 10; i++) // i starts at 0 and finishes with 9
{
    scanf("%d", &array[i]);
}
```

In this way the `scanf()` function call is executed `n` times (10 times in our example), but is written only once.

Here, the variable `i` is the loop index, and it is best declared as presented. The type `size_t` (`size type`) should be used for everything that counts or loops through data objects.

This way of declaring variables inside the `for` is only available for compilers that have been updated to the C99 standard. If for some reason you are still stuck with an older compiler you can declare the loop index before the `for` loop:

```
Version < C99
#include <stddef.h>          /* for size_t */
size_t i;
int array[10];                /* array of 10 int */

for (i = 0; i < 10; i++)      /* i starts at 0 and finishes at 9 */
{
    scanf("%d", &array[i]);
}
```

Section 15.2: Loop Unrolling and Duff's Device

Sometimes, the straight forward loop cannot be entirely contained within the loop body. This is because, the loop needs to be primed by some statements `B`. Then, the iteration begins with some statements `A`, which are then followed by `B` again before looping.

```
do_B();
while (condition) {
    do_A();
    do_B();
}
```

To avoid potential cut/paste problems with repeating `B` twice in the code, [Duff's Device](#) could be applied to start the loop from the middle of the `while` body, using a `switch` statement and fall through behavior.

```
switch (true) while (condition) {
case false: do_A(); /* FALL THROUGH */
```

```
default: do_B(); /* 贯穿 */  
}
```

Duff's Device实际上是为了实现循环展开而发明的。想象对一块内存应用掩码，其中 n 是一个带符号的整型，且值为正。

```
do {  
    *ptr++ ^= mask;  
} while (--n > 0);
```

如果 n 总是能被4整除，你可以轻松地将其展开为：

```
do {  
    *ptr++ ^= mask;  
    *ptr++ ^= mask;  
    *ptr++ ^= mask;  
    *ptr++ ^= mask;  
} while ((n -= 4) > 0);
```

但是，使用Duff's Device时，如果 n 不能被4整除，代码可以遵循这种展开惯用法，跳转到循环中间的正确位置。

```
switch (n % 4) do {  
case 0: *ptr++ ^= mask; /* 贯穿 */  
case 3: *ptr++ ^= mask; /* 贯穿 */  
case 2: *ptr++ ^= mask; /* 贯穿 */  
case 1: *ptr++ ^= mask; /* 贯穿 */  
} while ((n -= 4) > 0);
```

现代编译器很少需要这种手动展开，因为编译器的优化引擎可以代替程序员展开循环。

第15.3节：while循环

while循环用于在条件为真时执行一段代码。当需要执行一段代码块若干次时，应使用while循环。例如，下面的代码会获取用户输入，只要用户输入的数字不是0。如果用户输入0，while条件不再成立，程序将退出循环并继续执行后续代码：

```
int num = 1;  
  
while (num != 0)  
{  
    scanf("%d", &num);  
}
```

第15.4节：do-while循环

与for和while循环不同，do-while循环在循环末尾检查条件的真假，这意味着do代码块会先执行一次，然后在代码块底部检查while条件。也就是说，do-while循环至少会执行一次。

例如，这个do-while循环会从用户获取数字，直到这些数字的和大于或等于50：

```
default: do_B(); /* FALL THROUGH */  
}
```

Duff's Device was actually invented to implement loop unrolling. Imagine applying a mask to a block of memory, where n is a signed integral type with a positive value.

```
do {  
    *ptr++ ^= mask;  
} while (--n > 0);
```

If n were always divisible by 4, you could unroll this easily as:

```
do {  
    *ptr++ ^= mask;  
    *ptr++ ^= mask;  
    *ptr++ ^= mask;  
    *ptr++ ^= mask;  
} while ((n -= 4) > 0);
```

But, with Duff's Device, the code can follow this unrolling idiom that jumps into the right place in the middle of the loop if n is not divisible by 4.

```
switch (n % 4) do {  
case 0: *ptr++ ^= mask; /* FALL THROUGH */  
case 3: *ptr++ ^= mask; /* FALL THROUGH */  
case 2: *ptr++ ^= mask; /* FALL THROUGH */  
case 1: *ptr++ ^= mask; /* FALL THROUGH */  
} while ((n -= 4) > 0);
```

This kind of manual unrolling is rarely required with modern compilers, since the compiler's optimization engine can unroll loops on the programmer's behalf.

Section 15.3: While loop

A `while` loop is used to execute a piece of code while a condition is true. The `while` loop is to be used when a block of code is to be executed a variable number of times. For example the code shown gets the user input, as long as the user inserts numbers which are not 0. If the user inserts 0, the while condition is not true anymore so execution will exit the loop and continue on to any subsequent code:

```
int num = 1;  
  
while (num != 0)  
{  
    scanf("%d", &num);  
}
```

Section 15.4: Do-While loop

Unlike `for` and `while` loops, `do-while` loops check the truth of the condition at the end of the loop, which means the do block will execute once, and then check the condition of the `while` at the bottom of the block. Meaning that a `do-while` loop will *always* run at least once.

For example this `do-while` loop will get numbers from user, until the sum of these values is greater than or equal to 50:

```

int num, sum;
num = sum = 0;

do
{
    scanf("%d", &num);
    sum += num;
} 当 (总和 < 50);

```

do-while 循环在大多数编程风格中相对较少见。

第15.5节：for循环的结构和控制流程

```

for ([声明-或-表达式]; [表达式2]; [表达式3])
{
    /* 循环体 */
}

```

在一个 for 循环中，循环条件包含三个表达式，均为可选项。

- 第一个表达式，声明-或-表达式，初始化 循环。它在循环开始时执行一次。

版本 ≥ C99

它可以是循环变量的声明和初始化，也可以是一般表达式。如果是声明，该变量的作用域被 `for` 语句限制。

版本 < C99

历史版本的C语言只允许这里使用表达式，循环变量的声明必须放在 `for` 之前。

- 第二个表达式，表达式2，是 测试条件。它在初始化后首次执行。如果条件为 真，则控制进入循环体。如果不为真，则在循环结束时跳出循环体转到循环体外。随后，每次执行完循环体和更新语句后都会检查该条件。当为 真 时，控制流返回循环体开始处。该条件通常用于检查循环体执行的次数。这是退出循环的主要方式，另一种方式是使用跳转语句。
- 第三个表达式，expression3，是更新语句。它在循环体每次执行后执行。它通常用于递增一个变量，该变量用于计数循环体执行的次数，这个变量称为迭代器。

循环体的每次执行称为一次迭代。

示例：

```

版本 ≥ C99
for(int i = 0; i < 10 ; i++)
{
    printf("%d", i);
}

```

输出是：

```

int num, sum;
num = sum = 0;

do
{
    scanf(" %d", &num);
    sum += num;
} while (sum < 50);

```

do-while 循环在大多数编程风格中相对较少见。

Section 15.5: Structure and flow of control in a for loop

```

for ([declaration-or-expression]; [expression2]; [expression3])
{
    /* body of the loop */
}

```

In a `for` loop, the loop condition has three expressions, all optional.

- The first expression, `declaration-or-expression`, initializes the loop. It is executed exactly once at the beginning of the loop.

Version ≥ C99

It can be either a declaration and initialization of a loop variable, or a general expression. If it is a declaration, the scope of the declared variable is restricted by the `for` statement.

Version < C99

Historical versions of C only allowed an expression, here, and the declaration of a loop variable had to be placed before the `for`.

- The second expression, `expression2`, is the test condition. It is first executed after the initialization. If the condition is `true`, then the control enters the body of the loop. If not, it shifts to outside the body of the loop at the end of the loop. Subsequently, this condition is checked after each execution of the body as well as the update statement. When `true`, the control moves back to the beginning of the body of the loop. The condition is usually intended to be a check on the number of times the body of the loop executes. This is the primary way of exiting a loop, the other way being using jump statements.
- The third expression, `expression3`, is the update statement. It is executed after each execution of the body of the loop. It is often used to increment a variable keeping count of the number of times the loop body has executed, and this variable is called an iterator.

Each instance of execution of the loop body is called an iteration.

Example:

```

Version ≥ C99
for(int i = 0; i < 10 ; i++)
{
    printf("%d", i);
}

```

The output is:

在上述示例中，首先执行 `i = 0`，初始化 `i`。然后，检查条件 `i < 10`，结果为真。控制进入循环体，打印 `i` 的值。接着，控制转向 `i++`，将 `i` 的值从0更新为1。然后再次检查条件，过程继续。直到 `i` 的值变为10。此时，条件 `i < 10` 结果为假，控制跳出循环。

第15.6节：无限循环

如果控制进入循环体后永远不离开该循环体，则称该循环为无限循环。当循环的测试条件永远不为假时，就会发生这种情况。

示例：

```
版本 ≥ C99
for (int i = 0; i >= 0; )
{
    /* 循环体, i未被修改 */
}
```

在上述示例中，变量 `i`（迭代器）被初始化为0。测试条件最初为真。然而，`i` 在循环体内未被修改，更新表达式为空。因此，`i` 将保持为0，测试条件永远不会变为假，导致无限循环。

假设没有跳转语句，另一种形成无限循环的方法是显式地保持条件为真：

```
while (true)
{
    /* 循环体 */
}
```

在 `for` 循环中，条件语句是可选的。在这种情况下，条件总是 `true`，导致无限循环。

```
for (;;)
{
    /* 循环体 */
}
```

然而，在某些情况下，条件可能被故意保持为 `true`，目的是使用跳转语句如 `break` 退出循环。

```
while (true)
{
    /* 语句 */
    if (condition)
    {
        /* 更多语句 */
        break;
    }
}
```

In the above example, first `i = 0` is executed, initializing `i`. Then, the condition `i < 10` is checked, which evaluates to be `true`. The control enters the body of the loop and the value of `i` is printed. Then, the control shifts to `i++`, updating the value of `i` from 0 to 1. Then, the condition is again checked, and the process continues. This goes on till the value of `i` becomes 10. Then, the condition `i < 10` evaluates `false`, after which the control moves out of the loop.

Section 15.6: Infinite Loops

A loop is said to be an *infinite loop* if the control enters but never leaves the body of the loop. This happens when the test condition of the loop never evaluates to `false`.

Example:

```
Version ≥ C99
for (int i = 0; i >= 0; )
{
    /* body of the loop where i is not changed */
}
```

In the above example, the variable `i`, the iterator, is initialized to 0. The test condition is initially `true`. However, `i` is not modified anywhere in the body and the update expression is empty. Hence, `i` will remain 0, and the test condition will never evaluate to `false`, leading to an infinite loop.

Assuming that there are no jump statements, another way an infinite loop might be formed is by explicitly keeping the condition true:

```
while (true)
{
    /* body of the loop */
}
```

In a `for` loop, the condition statement optional. In this case, the condition is always `true` vacuously, leading to an infinite loop.

```
for (;;)
{
    /* body of the loop */
}
```

However, in certain cases, the condition might be kept `true` intentionally, with the intention of exiting the loop using a jump statement such as `break`.

```
while (true)
{
    /* statements */
    if (condition)
    {
        /* more statements */
        break;
    }
}
```

第16章：选择语句

第16.1节：if () 语句

控制程序流程的最简单方法之一是使用if选择语句。是否执行一段代码块可以由该语句决定。

C语言中if选择语句的语法如下：

```
if(cond)
{
    statement(s); /*条件为真时执行*/
}
```

例如，

```
if (a > 1) {
    puts("a is larger than 1");
}
```

其中 `a > 1` 是一个条件，必须计算为真，才能执行if块内的语句。在本例中，只有当 `a > 1` 为真时，才会打印“a is larger than 1”。

如果if选择语句块内只有一条语句，可以省略包裹的花括号{和}。上述例子可以改写为

```
if (a > 1)
    puts("a 大于 1");
```

但是要执行块内的多条语句，必须使用大括号。

if的条件可以包含多个表达式。只有当表达式的最终结果为真时，if才会执行操作。

例如

```
if ((a > 1) && (b > 1)) {
    puts("a 大于 1");
    a++;
}
```

只有当a和b都大于1时，才会执行printf和a++。

第16.2节：嵌套 if()...else 与 if()..else 梯形结构

嵌套的if()...else语句相比于if()...else梯形结构执行时间更长（更慢），因为嵌套的if()...else语句在外层条件if()满足后，会检查所有内部条件语句，而if()..else梯形结构一旦有任何if()或else if()条件为真，就会停止条件测试。

一个if()...else梯形结构：

```
#include <stdio.h>
```

Chapter 16: Selection Statements

Section 16.1: if () Statements

One of the simplest ways to control program flow is by using if selection statements. Whether a block of code is to be executed or not to be executed can be decided by this statement.

The syntax for if selection statement in C could be as follows:

```
if(cond)
{
    statement(s); /*to be executed, on condition being true*/
}
```

For example,

```
if (a > 1) {
    puts("a is larger than 1");
}
```

Where `a > 1` is a *condition* that has to evaluate to **true** in order to execute the statements inside the if block. In this example "a is larger than 1" is only printed if `a > 1` is true.

if selection statements can omit the wrapping braces { and } if there is only one statement within the block. The above example can be rewritten to

```
if (a > 1)
    puts("a is larger than 1");
```

However for executing multiple statements within block the braces have to used.

The *condition* for if can include multiple expressions. if will only perform the action if the end result of expression is true.

For example

```
if ((a > 1) && (b > 1)) {
    puts("a is larger than 1");
    a++;
}
```

will only execute the printf and a++ if **both** a and b are greater than 1.

Section 16.2: Nested if()...else VS if()..else Ladder

Nested if()...else statements take more execution time (they are slower) in comparison to an if()...else ladder because the nested if()...else statements check all the inner conditional statements once the outer conditional if() statement is satisfied, whereas the if()..else ladder will stop condition testing once any of the if() or the else if() conditional statements are true.

An if()...else ladder:

```
#include <stdio.h>
```

```

int main(int argc, char *argv[])
{
    int a, b, c;
    printf("请输入三个数字 = ");scanf("%d%d%d"
, &a, &b, &c);
    if ((a < b) && (a < c))
    {
        printf("a = %d 是最小的。", a);}

    else if ((b < a) && (b < c))
    {
        printf("b = %d 是最小的。", b);}

    else if ((c < a) && (c < b))
    {
        printf("c = %d 是最小的。", c);}

    else
    {
        printf("改进你的编码逻辑");}
    return 0;
}

```

一般情况下，被认为优于等效的嵌套if()...else:

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    int a, b, c;
    printf("请输入三个数字 = ");scanf("%d%d%d"
, &a, &b, &c);
    if (a < b)
    {
        if (a < c)
        {
            printf("a = %d 是最小的。", a);}

        else
        {
            printf("c = %d 是最小的。", c);}
    }

    else
    {
        if(b < c)
        {
            printf("b = %d 是最小的。", b);}

        else
        {
            printf("c = %d 是最小的。", c);}
    }
    return 0;
}

```

```

int main(int argc, char *argv[])
{
    int a, b, c;
    printf("\nEnter Three numbers = ");
    scanf("%d%d%d", &a, &b, &c);
    if ((a < b) && (a < c))
    {
        printf("\na = %d is the smallest.", a);
    }
    else if ((b < a) && (b < c))
    {
        printf("\nb = %d is the smallest.", b);
    }
    else if ((c < a) && (c < b))
    {
        printf("\nc = %d is the smallest.", c);
    }
    else
    {
        printf("\nImprove your coding logic");
    }
    return 0;
}

```

Is, in the general case, considered to be better than the equivalent nested `if()...else`:

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    int a, b, c;
    printf("\nEnter Three numbers = ");
    scanf("%d%d%d", &a, &b, &c);
    if (a < b)
    {
        if (a < c)
        {
            printf("\na = %d is the smallest.", a);
        }
        else
        {
            printf("\nc = %d is the smallest.", c);
        }
    }
    else
    {
        if(b < c)
        {
            printf("\nb = %d is the smallest.", b);
        }
        else
        {
            printf("\nc = %d is the smallest.", c);
        }
    }
    return 0;
}

```

第16.3节：switch () 语句

switch 语句在你希望程序根据某个特定测试变量的值执行多种不同操作时非常有用。

switch 语句的一个示例用法如下：

```
int a = 1;

switch (a) {
case 1:
    puts("a is 1");
    break;
case 2:
    puts("a is 2");
    break;
default:
    puts("a 既不是 1 也不是 2");
    break;
}
```

此示例等同于

```
int a = 1;

if (a == 1) {
    puts("a is 1");
} else if (a == 2) {
    puts("a is 2");
} else {
    puts("a 既不是 1 也不是 2");
}
```

如果在使用 switch 语句时 a 的值是 1，则会打印 a 是 1。如果 a 的值是 2，则会打印 a 是 2。否则，会打印 a 既不是 1 也不是 2。

case n: 用于描述当传递给 switch 语句的值是 n 时，执行流程将跳转到哪里。

n 必须是编译时常量，并且同一个 switch 语句中最多只能存在一个相同的 n。

default: 用于描述当值不匹配任何 case n: 的选项时的情况。在每个 switch 语句中包含一个 default 分支是一个良好的编程习惯，以捕获意外行为。

需要使用 break; 语句跳出 switch 代码块。

注意：如果你不小心忘记在 case 结束后添加 break，编译器会假定你打算“贯穿执行”，所有后续的 case 语句（如果有）都会被执行（除非在后续的某个 case 中找到了 break 语句），无论后续的 case 语句是否匹配。这一特性被用来实现 Duff 设备。这种行为通常被认为是 C 语言规范中的一个缺陷。

下面是一个展示缺少 break; 影响的示例：

```
int a = 1;

switch (a) {
case 1:
case 2:
```

Section 16.3: switch () Statements

switch statements are useful when you want to have your program do many different things according to the value of a particular test variable.

An example usage of switch statement is like this:

```
int a = 1;

switch (a) {
case 1:
    puts("a is 1");
    break;
case 2:
    puts("a is 2");
    break;
default:
    puts("a is neither 1 nor 2");
    break;
}
```

This example is equivalent to

```
int a = 1;

if (a == 1) {
    puts("a is 1");
} else if (a == 2) {
    puts("a is 2");
} else {
    puts("a is neither 1 nor 2");
}
```

If the value of a is 1 when the switch statement is used, a is 1 will be printed. If the value of a is 2 then, a is 2 will be printed. Otherwise, a is neither 1 nor 2 will be printed.

case n: is used to describe where the execution flow will jump in when the value passed to switch statement is n. n must be compile-time constant and the same n can exist at most once in one switch statement.

default: is used to describe that when the value didn't match any of the choices for case n:. It is a good practice to include a default case in every switch statement to catch unexpected behavior.

A break; statement is required to jump out of the switch block.

Note: If you accidentally forget to add a break after the end of a case, the compiler will assume that you intend to "fall through" and all the subsequent case statements, if any, will be executed (unless a break statement is found in any of the subsequent cases), regardless of whether the subsequent case statement(s) match or not. This particular property is used to implement Duff's Device. This behavior is often considered a flaw in the C language specification.

Below is an example that shows effects of the absence of break::

```
int a = 1;

switch (a) {
case 1:
case 2:
```

```

    puts("a 是 1 或 2");
case 3:
    puts("a 是 1、2 或 3");
    break;
default:
    puts("a 既不是 1, 也不是 2, 更不是 3");
    break;
}

```

当 a 的值为 1 或 2 时，会打印 “a 是 1 或 2” 和 “a 是 1、2 或 3”。当 a 为 3 时，只会打印 “a 是 1、2 或 3”。否则，会打印 “a 既不是 1, 也不是 2, 更不是 3”。

请注意，默认情况并非必要，尤其是在switch中获取的值集合已结束且在编译时已知的情况下。

使用枚举进行switch的最佳示例。

```

enum msg_type { ACK, PING, ERROR };
void f(enum msg_type t)
{
    switch (t) {
    case ACK:
        // 什么也不做
        break;
    case PING:
        // 做某事
        break;
    case ERROR:
        // 做其他事情
        break;
    }
}

```

这样做有多个优点：

- 如果你没有处理某个值，大多数编译器会发出警告（如果存在default分支，则不会报告此警告）
- 出于同样的原因，如果你向enum中添加一个新值，系统会通知你所有忘记处理新值的地方（如果使用default情况，你需要手动在代码中查找这些情况）
- 读者不需要去猜测“default隐藏了什么：“是否还有其他enum值，或者这是否是“以防万一”的保护措施。如果存在其他enum值，编码者是有意使用default情况处理它们，还是在添加值时引入了错误？
- 处理每个enum值使代码自解释，因为你不能隐藏在通配符后面，必须明确处理每一个值。

尽管如此，你无法阻止有人写出恶意代码，比如：

```
enum msg_type t = (enum msg_type)666; // 我是恶意的
```

因此，如果你确实需要，可以在switch之前添加额外检查来检测它。

```

void f(enum msg_type t)
{
    if (!is_msg_type_valid(t)) {
        // 处理这个不太可能的错误
    }
}

```

```

    puts("a is 1 or 2");
case 3:
    puts("a is 1, 2 or 3");
    break;
default:
    puts("a is neither 1, 2 nor 3");
    break;
}

```

When the value of a is 1 or 2, a is 1 or 2 and a is 1, 2 or 3 will both be printed. When a is 3, only a is 1, 2 or 3 will be printed. Otherwise, a is neither 1, 2 nor 3 will be printed.

Note that the default case is not necessary, especially when the set of values you get in the switch is finished and known at compile time.

The best example is using a switch on an enum.

```

enum msg_type { ACK, PING, ERROR };
void f(enum msg_type t)
{
    switch (t) {
    case ACK:
        // do nothing
        break;
    case PING:
        // do something
        break;
    case ERROR:
        // do something else
        break;
    }
}

```

There are multiple advantages of doing this:

- most compilers will report a warning if you don't handle a value (this would not be reported if a default case were present)
- for the same reason, if you add a new value to the enum, you will be notified of all the places where you forgot to handle the new value (with a default case, you would need to manually explore your code searching for such cases)
- The reader does not need to figure out "what is hidden by the default:", whether there other enum values or whether it is a protection for "just in case". And if there are other enum values, did the coder intentionally use the default case for them or is there a bug that was introduced when he added the value?
- handling each enum value makes the code self explanatory as you can't hide behind a wild card, you must explicitly handle each of them.

Nevertheless, you can't prevent someone to write evil code like:

```
enum msg_type t = (enum msg_type)666; // I'm evil
```

Thus you may add an extra check before your switch to detect it, if you really need it.

```

void f(enum msg_type t)
{
    if (!is_msg_type_valid(t)) {
        // Handle this unlikely error
    }
}

```

```
switch(t) {
    // 与之前相同的代码
}
```

第16.4节：if () ... else语句及语法

虽然if仅在其条件计算为true时执行操作，if / else允许你指定条件为true时和条件为false时的不同操作。

示例：

```
if (a > 1)
    puts("a 大于 1");
else
    puts("a 不大于 1");
```

就像if语句一样，当if或else中的代码块仅包含一条语句时，大括号可以省略（但不推荐这样做，因为这很容易无意中引入问题）。然而，如果if或else代码块中有多条语句，则必须使用大括号包裹该代码块。

```
if (a > 1)
{
    puts("a 大于 1");
    a--;
}
else
{
    puts("a 不大于 1");
    a++;
}
```

第16.5节：if()...else阶梯式链式两个或多个if () ... else语句

虽然if ()... else语句只允许定义一个（默认）行为，当if ()中的条件不满足时执行，链式两个或多个if () ... else语句则允许在最后的else分支（作为“默认”）之前定义更多行为（如果有的话）。

示例：

```
int a = ... /* 初始化为某个值。 */
if (a >= 1)
{
    printf("a 大于或等于 1。");
}
else if (a == 0) // 我们已经知道 a 小于 1
{
    printf("a 等于 0。");
}
else /* a 小于 1 且不等于 0，因此： */
{
    printf("a 是负数。");
}
```

```
switch(t) {
    // Same code than before
}
```

Section 16.4: if () ... else statements and syntax

While if performs an action only when its condition evaluate to **true**, if / **else** allows you to specify the different actions when the condition **true** and when the condition is **false**.

Example:

```
if (a > 1)
    puts("a is larger than 1");
else
    puts("a is not larger than 1");
```

Just like the **if** statement, when the block within **if** or **else** is consisting of only one statement, then the braces can be omitted (but doing so is not recommended as it can easily introduce problems involuntarily). However if there's more than one statement within the **if** or **else** block, then the braces have to be used on that particular block.

```
if (a > 1)
{
    puts("a is larger than 1");
    a--;
}
else
{
    puts("a is not larger than 1");
    a++;
}
```

Section 16.5: if()...else Ladder Chaining two or more if () ... else statements

While the **if ()... else** statement allows to define only one (default) behaviour which occurs when the condition within the **if ()** is not met, chaining two or more **if () ... else** statements allow to define a couple more behaviours before going to the last **else** branch acting as a "default", if any.

Example:

```
int a = ... /* initialise to some value. */

if (a >= 1)
{
    printf("a is greater than or equals 1.\n");
}
else if (a == 0) // we already know that a is smaller than 1
{
    printf("a equals 0.\n");
}
else /* a is smaller than 1 and not equals 0, hence: */
{
    printf("a is negative.\n");
}
```

第17章：初始化

第17.1节：C语言中的变量初始化

在没有显式初始化的情况下，外部变量和static变量保证初始化为零；自动变量（包括register变量）具有不确定1（即垃圾）初始值。

标量变量可以在定义时通过在名称后跟等号和一个

表达式来初始化：

```
int x = 1;
char squota = '\';
long day = 1000L * 60L * 60L * 24L; /* 毫秒/天 */
```

对于外部和static变量，初始化器必须是一个常量表达式²；初始化只执行一次，概念上在程序开始执行之前完成。

对于自动和register变量，初始化器不限制为常量：它可以是任何涉及先前定义值的表达式，甚至是函数调用。

例如，参见下面的代码片段

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

代替

```
int low, high, mid;
low = 0;
high = n - 1;
```

实际上，自动变量的初始化只是赋值语句的简写。选择哪种形式很大程度上取决于个人喜好。我们通常使用显式赋值，因为声明中的初始化器更难察觉，且距离使用点较远。另一方面，变量应尽可能在即将使用时才声明。

初始化数组：

数组可以通过在声明后跟随一组用大括号括起并用逗号分隔的初始化器来初始化。

例如，要初始化一个表示每个月天数的数组 days：

```
int days_of_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

（注意，在此结构中，1月被编码为第零个月。）当省略数组大小

时，编译器将通过计算初始化器的数量来确定长度，此例中有12个初始化器。

Chapter 17: Initialization

Section 17.1: Initialization of Variables in C

In the absence of explicit initialization, external and static variables are guaranteed to be initialized to zero; automatic variables (including register variables) have *indeterminate*¹ (i.e., garbage) initial values.

Scalar variables may be initialized when they are defined by following the name with an equals sign and an expression:

```
int x = 1;
char squota = '\';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

For external and static variables, the initializer must be a *constant expression*²; the initialization is done once, conceptually before the program begins execution.

For automatic and register variables, the initializer is not restricted to being a constant: it may be any expression involving previously defined values, even function calls.

For example, see the code snippet below

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

instead of

```
int low, high, mid;
low = 0;
high = n - 1;
```

In effect, initialization of automatic variables are just shorthand for assignment statements. Which form to prefer is largely a matter of taste. We generally use explicit assignments, because initializers in declarations are harder to see and further away from the point of use. On the other hand, variables should only be declared when they're about to be used whenever possible.

Initializing an array:

An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas.

For example, to initialize an array days with the number of days in each month:

```
int days_of_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

(Note that January is encoded as month zero in this structure.)

When the size of the array is omitted, the compiler will compute the length by counting the initializers, of which there are 12 in this case.

如果数组的初始化器数量少于指定大小，其他元素将被初始化为零，适用于所有类型的变量。

初始化器过多是错误的。没有标准方法来指定初始化器的重复—但GCC有一个extension来实现此功能。

版本 < C99

在C89/C90或更早版本的C中，没有办法仅初始化数组中间的某个元素而不提供所有前面的值。

版本 ≥ C99

从C99及以上版本开始，指定初始化器允许你初始化数组的任意元素，未初始化的值将被置为零。

初始化字符数组：

字符串是一种特殊的初始化情况；可以使用字符串代替大括号和逗号的表示法：

```
char chr_array[] = "hello";
```

是下面更长但等效写法的简写：

```
char chr_array[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

在这种情况下，数组大小为六（五个字符加上终止的'\0'）。

1 在C语言中，声明但未初始化的变量会发生什么？它有值吗？

2 注意，常量表达式被定义为可以在编译时求值的表达式。因此，`int global_var = f();` 是无效的。另一个常见误解是将`const`限定的变量视为常量表达式。在C语言中，`const`意味着“只读”，而不是“编译时常量”。所以，像`const int SIZE = 10; int global_arr[SIZE];` 和 `const int SIZE = 10; int global_var = SIZE;` 在C语言中是不合法的。

第17.2节：使用指定初始化器

版本 ≥ C99

C99 引入了指定初始化器的概念。这允许你指定数组、结构体或联合体的哪些元素由后续的值进行初始化。

数组元素的指定初始化器

对于像普通int这样的简单类型：

```
int array[] = { [4] = 29, [5] = 31, [17] = 101, [18] = 103, [19] = 107, [20] = 109 };
```

方括号中的项，可以是任何常量整数表达式，指定数组的哪个元素由=符号后面的值进行初始化。未指定的元素将被默认初始化，即定义为零。示例中展示了按顺序的指定初始化器；它们不必按顺序排列。示例中显示了间隔；这是合法的。示例中未显示对同一元素的两次不同初始化；这也是允许的（ISO/IEC 9899:2011, §6.7.9 初始化，¶19 初始化应按初始化列表顺序进行，为特定子对象提供的每个初始化器覆盖之前为同一子对象列出的任何初始化器）。

If there are fewer initializers for an array than the specified size, the others will be zero for all types of variables.

It is an error to have too many initializers. There is no standard way to specify repetition of an initializer — but GCC has an [extension](#) to do so.

Version < C99

In C89/C90 or earlier versions of C, there was no way to initialize an element in the middle of an array without supplying all the preceding values as well.

Version ≥ C99

With C99 and above, designated initializers allow you to initialize arbitrary elements of an array, leaving any uninitialized values as zeros.

Initializing Character arrays:

Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation:

```
char chr_array[] = "hello";
```

is a shorthand for the longer but equivalent:

```
char chr_array[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

In this case, the array size is six (five characters plus the terminating '`\0`').

1 What happens to a declared, uninitialized variable in C? Does it have a value?

2 Note that a *constant expression* is defined as something that can be evaluated at compile-time. So, `int global_var = f();` is invalid. Another common misconception is thinking of a `const` qualified variable as a *constant expression*. In C, `const` means "read-only", not "compile time constant". So, global definitions like `const int SIZE = 10; int global_arr[SIZE];` and `const int SIZE = 10; int global_var = SIZE;` are not legal in C.

Section 17.2: Using designated initializers

Version ≥ C99

C99 introduced the concept of *designated initializers*. These allow you to specify which elements of an array, structure or union are to be initialized by the values following.

Designated initializers for array elements

For a simple type like plain `int`:

```
int array[] = { [4] = 29, [5] = 31, [17] = 101, [18] = 103, [19] = 107, [20] = 109 };
```

The term in square brackets, which can be any constant integer expression, specifies which element of the array is to be initialized by the value of the term after the = sign. Unspecified elements are default initialized, which means zeros are defined. The example shows the designated initializers in order; they do not have to be in order. The example shows gaps; those are legitimate. The example doesn't show two different initializations for the same element; that too is allowed (ISO/IEC 9899:2011, §6.7.9 Initialization, ¶19 *The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject*).

在此示例中，数组的大小未明确定义，因此指定初始化器中指定的最大索引决定了数组的大小—示例中为21个元素。如果定义了大小，初始化超出数组末尾的条目将像往常一样是错误的。

结构体的指定初始化器

你可以使用`.元素`符号指定结构体中哪些元素被初始化：

```
struct Date
{
    int 年;
    int 月;
    int 日;
};

结构体 Date us_independence_day = { .day = 4, .month = 7, .year = 1776 };
```

如果元素未被列出，则默认初始化为零。

联合体的指定初始化器

您可以使用指定初始化器来指定联合体中要初始化的元素。

版本 = C89

在C标准之前，没有办法初始化union。C89/C90标准允许您初始化union的第一个成员——因此哪个成员排在第一位很重要。

```
结构体 discriminated_union
{
    枚举 { DU_INT, DU_DOUBLE } discriminant;
    联合体
    {
        int     du_int;
        double  du_double;
    } du;
};

结构体 discriminated_union du1 = { .discriminant = DU_INT, .du = { .du_int = 1 } };
结构体 discriminated_union du2 = { .discriminant = DU_DOUBLE, .du = { .du_double = 3.14159 } };

版本 ≥ C11
```

请注意，C11 允许你在结构体内使用匿名联合成员，因此在前面的例子中你不需要使用du名称

```
结构体 discriminated_union
{
    枚举 { DU_INT, DU_DOUBLE } discriminant;
    联合体
    {
        int     du_int;
        double  du_double;
    };
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du_int = 1 };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du_double = 3.14159 };
```

结构体数组等的指定初始化器

In this example, the size of the array is not defined explicitly, so the maximum index specified in the designated initializers dictates the size of the array — which would be 21 elements in the example. If the size was defined, initializing an entry beyond the end of the array would be an error, as usual.

Designed initializers for structures

You can specify which elements of a structure are initialized by using the `.element` notation:

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { .day = 4, .month = 7, .year = 1776 };
```

If elements are not listed, they are default initialized (zeroed).

Designed initializer for unions

You can specify which element of a union is initialize with a designed initializer.

Version = C89

Prior to the C standard, there was no way to initialize a `union`. The C89/C90 standard allows you to initialize the first member of a `union` — so the choice of which member is listed first matters.

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    } du;
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du = { .du_int = 1 } };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du = { .du_double = 3.14159 } };

Version ≥ C11
```

Note that C11 allows you to use anonymous union members inside a structure, so that you don't need the `du` name in the previous example:

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    };
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du_int = 1 };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du_double = 3.14159 };
```

Designed initializers for arrays of structures, etc

这些构造可以组合用于包含数组元素的结构体数组等。使用完整的大括号集合确保表示法没有歧义。

```
typedef struct Date Date; // See earlier in this example

struct date_range
{
    Date dr_from;
    Date dr_to;
    char dr_what[80];
};

struct date_range ranges[] =
{
    [3] = { .dr_from = { .year = 1066, .month = 10, .day = 14 },
            .dr_to = { .year = 1066, .month = 12, .day = 25 },
            .dr_what = "从黑斯廷斯战役到征服者威廉加冕",
        },
    [2] = { .dr_from = { .month = 7, .day = 4, .year = 1776 },
            .dr_to = { .month = 5, .day = 14, .year = 1787 },
            .dr_what = "从美国独立宣言到制宪会议",
        }
};
```

在数组初始化器中指定范围

GCC 提供了一个扩展，允许你指定数组中一段元素范围使用相同的初始化值：

```
int array[] = { [3 ... 7] = 29, 19 = 107 };
```

三个点需要与数字分开，否则其中一个点可能被解释为浮点数的一部分（最大匹配规则）。

第17.3节：初始化结构体和结构体数组

结构体和结构体数组可以通过一系列用大括号括起来的值来初始化，每个成员对应一个值。

```
struct Date
{
    int 年;
    int 月;
    int 日;
};

struct Date 美国独立日 = { 1776, 7, 4 };

struct Date 英国战役[] =
{
    { 1066, 10, 14 }, // 黑斯廷斯战役
    { 1815, 6, 18 }, // 滑铁卢战役
    { 1805, 10, 21 }, // 特拉法加战役
};
```

注意，数组初始化可以不写内部的大括号，过去（比如1990年前）通常也是这样写的：

These constructs can be combined for arrays of structures containing elements that are arrays, etc. Using full sets of braces ensures that the notation is unambiguous.

```
typedef struct Date Date; // See earlier in this example

struct date_range
{
    Date dr_from;
    Date dr_to;
    char dr_what[80];
};

struct date_range ranges[] =
{
    [3] = { .dr_from = { .year = 1066, .month = 10, .day = 14 },
            .dr_to = { .year = 1066, .month = 12, .day = 25 },
            .dr_what = "Battle of Hastings to Coronation of William the Conqueror",
        },
    [2] = { .dr_from = { .month = 7, .day = 4, .year = 1776 },
            .dr_to = { .month = 5, .day = 14, .year = 1787 },
            .dr_what = "US Declaration of Independence to Constitutional Convention",
        }
};
```

Specifying ranges in array initializers

GCC provides an [extension](#) that allows you to specify a range of elements in an array that should be given the same initializer:

```
int array[] = { [3 ... 7] = 29, 19 = 107 };
```

The triple dots need to be separate from the numbers lest one of the dots be interpreted as part of a floating point number ([maximal munch](#) rule).

Section 17.3: Initializing structures and arrays of structures

Structures and arrays of structures can be initialized by a series of values enclosed in braces, one value per member of the structure.

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { 1776, 7, 4 };

struct Date uk_battles[] =
{
    { 1066, 10, 14 }, // Battle of Hastings
    { 1815, 6, 18 }, // Battle of Waterloo
    { 1805, 10, 21 }, // Battle of Trafalgar
};
```

Note that the array initialization could be written without the interior braces, and in times past (before 1990, say) often would have been written without them:

```
struct Date 英国战役[] =  
{  
    1066, 10, 14,    // 黑斯廷斯战役  
    1815, 6, 18,    // 滑铁卢战役  
    1805, 10, 21,    // 特拉法加战役  
};
```

虽然这样写可以，但这不是现代的良好风格—你不应该在新代码中尝试使用这种表示法，并且应该修正编译器通常会产生的警告。

另见指定初始化器。

```
struct Date uk_battles[] =  
{  
    1066, 10, 14,    // Battle of Hastings  
    1815, 6, 18,    // Battle of Waterloo  
    1805, 10, 21,    // Battle of Trafalgar  
};
```

Although this works, it is not good modern style — you should not attempt to use this notation in new code and should fix the compiler warnings it usually yields.

See also designated initializers.

第18章：声明与定义

第18.1节：理解声明与定义

声明引入一个标识符并描述其类型，无论是类型、对象还是函数。声明是编译器接受对该标识符引用所需的内容。以下是声明：

```
extern int bar;
extern int g(int, int);
double f(int, double); /* 函数声明中 extern 可以省略 */
double h1();           /* 无原型的声明 */
double h2();           /* 同上 */
```

定义实际上实例化/实现了该标识符。这是链接器链接对这些实体的引用所需的内容。以下是对应上述声明的定义：

```
int bar;
int g(int lhs, int rhs) {return lhs*rhs;}
double f(int i, double d) {return i+d;}
double h1(int a, int b) {return -1.5;}
double h2() {} /* 定义中隐含了原型，与 double h2(void) 相同 */
```

定义可以用来代替声明。

但是，必须且只能定义一次。如果你忘记定义某个已经声明并在某处被引用的东西，链接器就不知道该链接哪个引用，并会报缺少符号的错误。如果你定义了多次，链接器就不知道该链接哪个定义，并会报重复符号的错误。

例外情况：

```
extern int i = 0; /* 定义 i */
extern int j; /* 声明 j */
```

这个例外可以用“强符号与弱符号”（从链接器的角度）来解释。
请查看那里（幻灯片22）以获得更多解释。

```
/* 全部是定义 */
struct S { int a; int b; };      /* 定义了 S */
struct X {
    int x;                      /* 定义了 X */
    /* 定义了非静态数据成员 x */
};
struct X anX;                  /* 定义了 anX */
```

Chapter 18: Declaration vs Definition

Section 18.1: Understanding Declaration and Definition

A declaration introduces an identifier and describes its type, be it a type, object, or function. A declaration is what the compiler needs to accept references to that identifier. These are declarations:

```
extern int bar;
extern int g(int, int);
double f(int, double); /* extern can be omitted for function declarations */
double h1();           /* declaration without prototype */
double h2();           /* ditto */
```

A definition actually instantiates/implements this identifier. It's what the linker needs in order to link references to those entities. These are definitions corresponding to the above declarations:

```
int bar;
int g(int lhs, int rhs) {return lhs*rhs;}
double f(int i, double d) {return i+d;}
double h1(int a, int b) {return -1.5;}
double h2() {} /* prototype is implied in definition, same as double h2(void) */
```

A definition can be used in the place of a declaration.

However, it must be defined exactly once. If you forget to define something that's been declared and referenced somewhere, then the linker doesn't know what to link references to and complains about a missing symbols. If you define something more than once, then the linker doesn't know which of the definitions to link references to and complains about duplicated symbols.

Exception:

```
extern int i = 0; /* defines i */
extern int j; /* declares j */
```

This exception can be explained using concepts of "Strong symbols vs Weak symbols" (from a linker's perspective). Please look [here](#) (Slide 22) for more explanation.

```
/* All are definitions */
struct S { int a; int b; };      /* defines S */
struct X {
    int x;                      /* defines X */
    /* defines non-static data member x */
};
struct X anX;                  /* defines anX */
```

第19章：命令行参数

参数	详情
argc	参数计数 - 初始化为从命令行传递给程序的以空格分隔的参数数量，包括程序名本身。
argv	参数向量 - 初始化为包含命令行中给出的参数（以及程序名称）的char指针（字符串）数组。

第19.1节：打印程序参数并转换为整数值

以下代码将打印程序的参数，并尝试将每个参数转换为数字（转换为long类型）：

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <limits.h>

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

    for (int i = 1; i < argc; i++) {printf("参数 %d 是: %s", i, argv[i]);

        errno = 0;
        char *p;
        long argument_numValue = strtol(argv[i], &p, 10);

        if (p == argv[i]) {
            fprintf(stderr, "参数 %d 不是数字。", i);}

        else if ((argument_numValue == LONG_MIN || argument_numValue == LONG_MAX) && errno == ERANGE) {
            fprintf(stderr, "参数 %d 超出范围。", i);}

        else {
            printf("参数 %d 是一个数字, 值为: %ld", i, argument_numValue);}

    }
    return 0;
}
```

参考文献：

- [strtol\(\) 返回错误的值](#)
- [strtol 的正确用法](#)

第19.2节：打印命令行参数

接收参数后，可以按如下方式打印：

```
int main(int argc, char **argv)
{
    for (int i = 1; i < argc; i++)
    {
        printf("参数 %d: [%s]", i, argv[i]);
```

Chapter 19: Command-line arguments

Parameter	Details
argc	argument count - initialized to the number of space-separated arguments given to the program from the command-line as well as the program name itself.
argv	argument vector - initialized to an array of char-pointers (strings) containing the arguments (and the program name) that was given on the command-line.

Section 19.1: Print the arguments to a program and convert to integer values

The following code will print the arguments to the program, and the code will attempt to convert each argument into a number (to a `long`):

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <limits.h>

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

    for (int i = 1; i < argc; i++) {
        printf("Argument %d is: %s\n", i, argv[i]);

        errno = 0;
        char *p;
        long argument_numValue = strtol(argv[i], &p, 10);

        if (p == argv[i]) {
            fprintf(stderr, "Argument %d is not a number.\n", i);}

        else if ((argument_numValue == LONG_MIN || argument_numValue == LONG_MAX) && errno == ERANGE) {
            fprintf(stderr, "Argument %d is out of range.\n", i);}

        else {
            printf("Argument %d is a number, and the value is: %ld\n", i, argument_numValue);}

    }
    return 0;
}
```

References:

- [strtol\(\) returns an incorrect value](#)
- [Correct usage of strtol](#)

Section 19.2: Printing the command line arguments

After receiving the arguments, you can print them as follows:

```
int main(int argc, char **argv)
{
    for (int i = 1; i < argc; i++)
    {
        printf("Argument %d: [%s]\n", i, argv[i]);
```

```
}
```

备注

- 参数`argv`也可以定义为`char *argv[]`。
- `argv[0]`可能包含程序本身的名称（取决于程序是如何执行的）。第一个“真正的”命令行参数位于`argv[1]`，这也是循环变量`i`初始化为1的原因。
- 在打印语句中，你可以使用`*(argv + i)`代替`argv[i]`——它们的结果相同，但前者更冗长。
- 参数值周围的方括号有助于识别起始和结束位置。如果参数中有尾随空格、换行符、回车符或其他奇怪字符，这非常有用。该程序的某些变体是调试shell脚本的有用工具，帮助你理解参数列表实际包含的内容（尽管有一些简单的shell替代方案几乎等效）。

第19.3节：使用GNU getopt工具

在C语言中，应用程序的命令行选项与命令行参数没有区别。在Linux或Unix环境中，它们只是以破折号（-）开头的参数。

在Linux或Unix环境中的glibc中，你可以使用[getopt工具](#)轻松定义、验证和解析命令行选项与其他参数的区别。

这些工具期望您的选项格式符合GNU编码标准，该标准是对POSIX规范中命令行选项格式的扩展。

下面的示例演示了如何使用GNU getopt工具处理命令行选项。

```
#include <stdio.h>
#include <getopt.h>
#include <string.h>

/* 打印所有支持选项的说明 */
void usage (FILE *fp, const char *path)
{
    /* 取路径的最后一部分 */
    const char *basename = strrchr(path, '/');
    basename = basename ? basename + 1 : path;

    fprintf (fp, "usage: %s [OPTION]", basename);fprintf (fp,
    " -h, --help"
        "打印此帮助信息并退出。");
    fprintf (fp, " -f, --file[=FILENAME]" "将所有输出写
        入文件（默认为out.txt）。");fprintf (fp, " -m, --msg=STRING"
        "输出特定消息而不是'Hello world'。");
}

/* 解析命令行选项并打印消息 */
int main(int argc, char *argv[])
{
    /* 为了代码简洁，此示例仅使用固定大小的字符串缓冲区 */
    char filename[256] = { 0 };
    char message[256] = "Hello world";
    FILE *fp;
    int help_flag = 0;
    int opt;
```

```
}
```

Notes

- The `argv` parameter can be also defined as `char *argv[]`.
- `argv[0]` may contain the program name itself (depending on how the program was executed). The first "real" command line argument is at `argv[1]`, and this is the reason why the loop variable `i` is initialized to 1.
- In the print statement, you can use `*(argv + i)` instead of `argv[i]` - it evaluates to the same thing, but is more verbose.
- The square brackets around the argument value help identify the start and end. This can be invaluable if there are trailing blanks, newlines, carriage returns, or other oddball characters in the argument. Some variant on this program is a useful tool for debugging shell scripts where you need to understand what the argument list actually contains (although there are simple shell alternatives that are almost equivalent).

Section 19.3: Using GNU getopt tools

Command-line options for applications are not treated any differently from command-line arguments by the C language. They are just arguments which, in a Linux or Unix environment, traditionally begin with a dash (-).

With glibc in a Linux or Unix environment you can use the [getopt tools](#) to easily define, validate, and parse command-line options from the rest of your arguments.

These tools expect your options to be formatted according to the [GNU Coding Standards](#), which is an extension of what POSIX specifies for the format of command-line options.

The example below demonstrates handling command-line options with the GNU getopt tools.

```
#include <stdio.h>
#include <getopt.h>
#include <string.h>

/* print a description of all supported options */
void usage (FILE *fp, const char *path)
{
    /* take only the last portion of the path */
    const char *basename = strrchr(path, '/');
    basename = basename ? basename + 1 : path;

    fprintf (fp, "usage: %s [OPTION]\n", basename);
    fprintf (fp, " -h, --help\n"
        "Print this help and exit.\n");
    fprintf (fp, " -f, --file[=FILENAME]\n"
        "Write all output to a file (defaults to out.txt).\n");
    fprintf (fp, " -m, --msg=STRING\n"
        "Output a particular message rather than 'Hello world'.\n");
}

/* parse command-line options and print message */
int main(int argc, char *argv[])
{
    /* for code brevity this example just uses fixed buffer sizes for strings */
    char filename[256] = { 0 };
    char message[256] = "Hello world";
    FILE *fp;
    int help_flag = 0;
    int opt;
```

```

/* 所有支持的长格式选项表。
* 字段：名称, has_arg, flag, val
* `has_arg` 指定关联的长格式选项是否可以（或在某些情况下必须）带有参数。`has_arg` 的有效值为
  * `no_argument`, `optional_argument` 和 `required_argument`。
* 如果 `flag` 指向一个变量，则当命令行中出现关联的长格式选项时，该变量将被赋值为 `val`。
* 如果 `flag` 为 NULL，则当在命令行参数中找到关联的长格式选项时，`getopt_long`（见下文）将
  返回 `val`。
*/
struct option longopts[] = {
{ "help", no_argument, &help_flag, 1 },
{ "file", optional_argument, NULL, 'f' },
{ "msg", required_argument, NULL, 'm' },
{ 0 }
};

/* 无限循环，当我们完成选项解析时跳出 */
while (1) {
    /* getopt_long 支持 GNU 风格的完整单词“长”选项，除此之外
   支持的单字符“短”选项。          * 还支持 getopt
   */

    /* 第三个参数是支持的短格式选项集合。
   * 这些不一定要与长格式选项对应。
   * 选项后一个冒号表示该选项有参数，两个冒号表示参数是可选的。顺序无关紧要。
   */
    opt = getopt_long (argc, argv, "hf::m:", longopts, 0);

    if (opt == -1) {
        /* 返回值为 -1 表示没有更多选项 */
        break;
    }

    switch (opt) {
    case 'h':
        /* help_flag 和 value 在 longopts 表中指定，          * 这意味着当指定
   --help 选项（长格式）时，help_flag 变量会自动被设置。
   */
        /* 但是，短格式选项的解析器不支持自动设置标志，          * 因此当指定 -h 选项时，我们仍然
   需要这段代码手动设置 help_flag。
   */
        /*
   help_flag = 1;
        break;
    case 'f':
        /* optarg 是 getopt.h 中的全局变量。它包含该选项的参数。
   * 如果没有参数，则为 null。
   */
        printf ("outarg: '%s'", optarg);strncpy (file
name, optarg ? optarg : "out.txt", sizeof (filename));
        /* strncpy 并不完全保证字符串以 null 结尾 */
filename[sizeof (filename) - 1] = '\0';
        break;
    case 'm':
        /* 由于此选项的参数是必需的，getopt 保证
   * optarg 非空。
   */
        strncpy (message, optarg, sizeof (message));
        message[sizeof (message) - 1] = '\0';
    }
}

```

```

/* table of all supported options in their long form.
* fields: name, has_arg, flag, val
* `has_arg` specifies whether the associated long-form option can (or, in
* some cases, must) have an argument. the valid values for `has_arg` are
* `no_argument`, `optional_argument`, and `required_argument`.
* if `flag` points to a variable, then the variable will be given a value
* of `val` when the associated long-form option is present at the command
* line.
* if `flag` is NULL, then `val` is returned by `getopt_long` (see below)
* when the associated long-form option is found amongst the command-line
* arguments.
*/
struct option longopts[] = {
{ "help", no_argument, &help_flag, 1 },
{ "file", optional_argument, NULL, 'f' },
{ "msg", required_argument, NULL, 'm' },
{ 0 }
};

/* infinite loop, to be broken when we are done parsing options */
while (1) {
    /* getopt_long supports GNU-style full-word "long" options in addition
   to the single-character "short" options which are supported by
   getopt.
   * the third argument is a collection of supported short-form options.
   * these do not necessarily have to correlate to the long-form options.
   * one colon after an option indicates that it has an argument, two
   * indicates that the argument is optional. order is unimportant.
   */
    opt = getopt_long (argc, argv, "hf::m:", longopts, 0);

    if (opt == -1) {
        /* a return value of -1 indicates that there are no more options */
        break;
    }

    switch (opt) {
    case 'h':
        /* the help_flag and value are specified in the longopts table,
   * which means that when the --help option is specified (in its long
   * form), the help_flag variable will be automatically set.
   * however, the parser for short-form options does not support the
   * automatic setting of flags, so we still need this code to set the
   * help_flag manually when the -h option is specified.
   */
        help_flag = 1;
        break;
    case 'f':
        /* optarg is a global variable in getopt.h. it contains the argument
   * for this option. it is null if there was no argument.
   */
        printf ("outarg: '%s'\n", optarg);
        strncpy (filename, optarg ? optarg : "out.txt", sizeof (filename));
        /* strncpy does not fully guarantee null-termination */
filename[sizeof (filename) - 1] = '\0';
        break;
    case 'm':
        /* since the argument for this option is required, getopt guarantees
   * that optarg is non-null.
   */
        strncpy (message, optarg, sizeof (message));
        message[sizeof (message) - 1] = '\0';
    }
}

```

```

        break;
    case '?':
        /* 返回值 '?' 表示选项格式错误。
         * 这可能意味着给出了未识别的选项，或者
         * 需要参数的选项未包含参数。
         */
    usage (stderr, argv[0]);
    return 1;
default:
    break;
}

if (help_flag) {
    usage (stdout, argv[0]);
    return 0;
}

if (filename[0]) {
    fp = fopen (filename, "w");
} else {
fp = stdout;
}

if (!fp) {
    fprintf(stderr, "Failed to open file.");return 1;
}

fprintf (fp, "%s", message);fclose (fp
);

return 0;
}

```

它可以用 gcc 编译：

```
gcc example.c -o example
```

它支持三个命令行选项（--help, --file 和 --msg）。所有选项都有对应的“短格式”（-h, -f 和 -m）。
“file”和“msg”选项都接受参数。如果指定了“msg”选项，则必须提供其参数。

选项的参数格式如下：

- --option=value (用于长格式选项)
- -o"value" 或 -o" value" (用于短格式选项)

```

        break;
    case '?':
        /* a return value of '?' indicates that an option was malformed.
         * this could mean that an unrecognized option was given, or that an
         * option which requires an argument did not include an argument.
         */
    usage (stderr, argv[0]);
    return 1;
default:
    break;
}

if (help_flag) {
    usage (stdout, argv[0]);
    return 0;
}

if (filename[0]) {
    fp = fopen (filename, "w");
} else {
    fp = stdout;
}

if (!fp) {
    fprintf(stderr, "Failed to open file.\n");
    return 1;
}

fprintf (fp, "%s\n", message);
fclose (fp);
return 0;
}

```

It can be compiled with gcc:

```
gcc example.c -o example
```

It supports three command-line options (--help, --file, and --msg). All have a "short form" as well (-h, -f, and -m).
The "file" and "msg" options both accept arguments. If you specify the "msg" option, its argument is required.

Arguments for options are formatted as:

- --option=value (for long-form options)
- -o"value" 或 -o" value" (for short-form options)

第20章：文件和输入/输出流

参数

详情

const char *mode 描述文件支持流打开模式的字符串。有关可能的值，请参见备注。

int whence

可以是SEEK_SET表示从文件开头设置，SEEK_END表示从文件末尾设置，或SEEK_CUR表示相对于当前位置光标位置设置。注意：SEEK_END是非移植性的。

第20.1节：打开并写入文件

```
#include <stdio.h> /* 用于 perror()、fopen()、fputs() 和 fclose() */
#include <stdlib.h> /* 用于 EXIT_* 宏 */

int main(int argc, char **argv)
{
    int e = EXIT_SUCCESS;

    /* 从参数获取路径，否则默认为 output.txt */
    char *path = (argc > 1) ? argv[1] : "output.txt";

    /* 打开文件以进行写入并获取文件指针 */
    FILE *file = fopen(path, "w");

    /* 如果 fopen() 失败，打印错误信息并退出 */
    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* 向文件写入文本。与 puts() 不同，fputs() 不会添加换行符。 */
    if (fputs("Output in file.", file) == EOF){

        perror(path);
        e = EXIT_FAILURE;
    }

    /* 关闭文件 */
    if (fclose(file))
    {
        perror(path);
        return EXIT_FAILURE;
    }
    return e;
}
```

该程序使用传递给 `main` 的参数中给出的文件名打开文件，如果没有给出参数，则默认打开 `output.txt` 文件。如果同名文件已存在，其内容将被丢弃，文件被视为一个新的空文件。如果文件不存在，`fopen()` 调用会创建该文件。

如果 `fopen()` 调用因某种原因失败，它会返回 `NULL` 值并设置全局变量 `errno` 的值。这意味着程序可以在 `fopen()` 调用后测试返回值，如果 `fopen()` 失败，则使用 `perror()`。

如果 `fopen()` 调用成功，它会返回一个有效的 `FILE` 指针。该指针随后可用于引用该文件，直到调用 `fclose()` 关闭它。

`fputs()` 函数将给定文本写入已打开的文件，替换文件的任何先前内容。与 `fopen()` 类似，`fputs()` 函数在失败时也会设置 `errno` 值，不过此时函数返回 `EOF` 以

Chapter 20: Files and I/O streams

Parameter

Details

const char *mode A string describing the opening mode of the file-backed stream. See remarks for possible values.

int whence

Can be SEEK_SET to set from the beginning of the file, SEEK_END to set from its end, or SEEK_CUR to set relative to the current cursor value. Note: SEEK_END is non-portable.

Section 20.1: Open and write to file

```
#include <stdio.h> /* for perror(), fopen(), fputs() and fclose() */
#include <stdlib.h> /* for the EXIT_* macros */

int main(int argc, char **argv)
{
    int e = EXIT_SUCCESS;

    /* Get path from argument to main else default to output.txt */
    char *path = (argc > 1) ? argv[1] : "output.txt";

    /* Open file for writing and obtain file pointer */
    FILE *file = fopen(path, "w");

    /* Print error message and exit if fopen() failed */
    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Writes text to file. Unlike puts(), fputs() does not add a new-line. */
    if (fputs("Output in file.\n", file) == EOF)
    {
        perror(path);
        e = EXIT_FAILURE;
    }

    /* Close file */
    if (fclose(file))
    {
        perror(path);
        return EXIT_FAILURE;
    }
    return e;
}
```

This program opens the file with name given in the argument to main, defaulting to `output.txt` if no argument is given. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file. If the file does not already exist the `fopen()` call creates it.

If the `fopen()` call fails for some reason, it returns a `NULL` value and sets the global `errno` variable value. This means that the program can test the returned value after the `fopen()` call and use `perror()` if `fopen()` fails.

If the `fopen()` call succeeds, it returns a valid `FILE` pointer. This pointer can then be used to reference this file until `fclose()` is called on it.

The `fputs()` function writes the given text to the opened file, replacing any previous contents of the file. Similarly to `fopen()`, the `fputs()` function also sets the `errno` value if it fails, though in this case the function returns `EOF` to

表示失败（否则返回非负值）。

`fclose()` 函数会刷新任何缓冲区，关闭文件并释放 `FILE *` 指针指向的内存。返回值表示完成情况，类似于 `fputs()`（成功时返回'0'），失败时同样会设置 `errno` 值。

第20.2节：运行进程

```
#include <stdio.h>

void print_all(FILE *stream)
{
    int c;
    while ((c = getc(stream)) != EOF)
        putchar(c);
}

int main(void)
{
    FILE *stream;

    /* 调用 netstat 命令。netstat 可用于 Windows 和 Linux */
    if ((stream = popen("netstat", "r")) == NULL)
        return 1;

    print_all(stream);
    pclose(stream);
    return 0;
}
```

该程序通过 `popen()` 运行一个进程（`netstat`），并读取该进程的所有标准输出，然后将其回显到标准输出。

注意：`popen()` 不存在于 标准 C 库 中，而是 POSIX C 的一部分

第20.3节：fprintf

你可以像在控制台使用 `printf` 一样，在文件上使用 `fprintf`。例如，为了记录游戏的胜利、失败和平局，你可以写

```
/* 保存胜利、失败和平局 */
void savewlt(FILE *fout, int wins, int losses, int ties)
{
    fprintf(fout, "胜场: %d平局: %d负场: %d", wins, ties, losses);}
```

附注：某些系统（尤其是Windows）并不使用大多数程序员所称的“正常”换行符。虽然类UNIX系统使用来结束行，Windows则使用一对字符：`\r`（回车）和`\n`（换行）。这个序列通常称为CRLF。然而，在使用C语言时，你无需担心这些高度依赖平台的细节。C编译器必须将每个实例转换为正确的平台换行符。因此，Windows编译器会将转换为`\r`，而UNIX编译器则保持为。

第20.4节：使用getline()从文件中获取行

POSIX C库定义了 `getline()` 函数。该函数分配一个缓冲区来保存行内容，并返回新行、行中字符数以及缓冲区大小。

indicate the fail (it otherwise returns a non-negative value).

The `fclose()` function flushes any buffers, closes the file and frees the memory pointed to by `FILE *`. The return value indicates completion just as `fputs()` does (though it returns '0' if successful), again also setting the `errno` value in the case of a fail.

Section 20.2: Run process

```
#include <stdio.h>

void print_all(FILE *stream)
{
    int c;
    while ((c = getc(stream)) != EOF)
        putchar(c);
}

int main(void)
{
    FILE *stream;

    /* call netstat command. netstat is available for Windows and Linux */
    if ((stream = popen("netstat", "r")) == NULL)
        return 1;

    print_all(stream);
    pclose(stream);
    return 0;
}
```

This program runs a process (`netstat`) via `popen()` and reads all the standard output from the process and echoes that to standard output.

Note: `popen()` does not exist in the [standard C library](#), but it is rather a part of [POSIX C](#)

Section 20.3: fprintf

You can use `fprintf` on a file just like you might on a console with `printf`. For example to keep track of game wins, losses and ties you might write

```
/* saves wins, losses and, ties */
void savewlt(FILE *fout, int wins, int losses, int ties)
{
    fprintf(fout, "Wins: %d\nTies: %d\nLosses: %d\n", wins, ties, losses);}
```

A side note: Some systems (infamously, Windows) do not use what most programmers would call "normal" line endings. While UNIX-like systems use `\n` to terminate lines, Windows uses a pair of characters: `\r` (carriage return) and `\n` (line feed). This sequence is commonly called CRLF. However, whenever using C, you do not need to worry about these highly platform-dependent details. A C compiler is required to convert every instance of `\n` to the correct platform line ending. So a Windows compiler would convert `\n` to `\r\n`, but a UNIX compiler would keep it as-is.

Section 20.4: Get lines from a file using getline()

The POSIX C library defines the `getline()` function. This function allocates a buffer to hold the line contents and returns the new line, the number of characters in the line, and the size of the buffer.

从example.txt获取每行的示例程序：

```
#include <stdlib.h>
#include <stdio.h>

#define FILENAME "example.txt"

int main(void)
{
    /* 打开文件以供读取 */
    char *line_buf = NULL;
    size_t line_buf_size = 0;
    int line_count = 0;
    ssize_t line_size;
    FILE *fp = fopen(FILENAME, "r");
    if (!fp)
    {
        fprintf(stderr, "Error opening file '%s'", FILENAME); return EXIT_FAILURE;
    }

    /* 获取文件的第一行。*/
    line_size = getline(&line_buf, &line_buf_size, fp);

    /* 循环直到文件处理完毕。*/
    while (line_size >= 0)
    {
        /* 行计数加一 */
        line_count++;

        /* 显示行的详细信息 */
        printf("line[%06d]: chars=%06zd, buf size=%06zu, contents: %s", line_count,
               line_size, line_buf_size, line_buf);

        /* 获取下一行 */
        line_size = getline(&line_buf, &line_buf_size, fp);
    }

    /* 释放分配的行缓冲区 */
    free(line_buf);
    line_buf = NULL;

    /* 文件处理完成后关闭文件 */
    fclose(fp);

    return EXIT_SUCCESS;
}
```

输入文件 example.txt

这是一个文件
其中包含
多行文本
带有各种缩进，
空白行

一行非常长的文本，用于展示 getline() 如果一行的长度超过了它所分配的缓冲区，getline() 会重新分配行缓冲区，以及行尾的标点符号。

Example program that gets each line from example.txt:

```
#include <stdlib.h>
#include <stdio.h>

#define FILENAME "example.txt"

int main(void)
{
    /* Open the file for reading */
    char *line_buf = NULL;
    size_t line_buf_size = 0;
    int line_count = 0;
    ssize_t line_size;
    FILE *fp = fopen(FILENAME, "r");
    if (!fp)
    {
        fprintf(stderr, "Error opening file '%s'\n", FILENAME);
        return EXIT_FAILURE;
    }

    /* Get the first line of the file. */
    line_size = getline(&line_buf, &line_buf_size, fp);

    /* Loop through until we are done with the file. */
    while (line_size >= 0)
    {
        /* Increment our line count */
        line_count++;

        /* Show the line details */
        printf("line[%06d]: chars=%06zd, buf size=%06zu, contents: %s", line_count,
               line_size, line_buf_size, line_buf);

        /* Get the next line */
        line_size = getline(&line_buf, &line_buf_size, fp);
    }

    /* Free the allocated line buffer */
    free(line_buf);
    line_buf = NULL;

    /* Close the file now that we are done with it */
    fclose(fp);

    return EXIT_SUCCESS;
}
```

Input file example.txt

This is a file
which has
multiple lines
with various indentation,
blank lines

a really long line to show that getline() will reallocate the line buffer if the length of a line is too long to fit in the buffer it has been given, and punctuation at the end of the lines.

输出

```
line[000001]: 字符数=000015, 缓冲区大小=000016, 内容：这是一个文件  
line[000002]: 字符数=000012, 缓冲区大小=000016, 内容： 它有  
line[000003]: 字符数=000015, 缓冲区大小=000016, 内容：多行  
line[000004]: 字符数=000030, 缓冲区大小=000032, 内容： 带有各种缩进,  
line[000005]: 字符数=000012, 缓冲区大小=000032, 内容：空白行  
line[000006]: 字符数=000001, 缓冲区大小=000032, 内容：  
line[000007]: 字符数=000001, 缓冲区大小=000032, 内容：  
line[000008]: 字符数=000001, 缓冲区大小=000032, 内容：  
line[000009]: 字符数=000150, 缓冲区大小=000160, 内容：一行非常长的文本, 用来展示 getline()如果一行的长度太长, 无法  
放入它所分配的缓冲区, getline() 会重新分配该行缓冲区,  
  
line[000010]: 字符数=000042, 缓冲区大小=000160, 内容： 以及行尾的标点符号。  
line[000011]: 字符数=000001, 缓冲区大小=000160, 内容：
```

在示例中, `getline()` 最初调用时没有分配缓冲区。在第一次调用期间, `getline()` 分配了一个缓冲区, 读取第一行并将其内容放入新缓冲区。在后续调用中, `getline()` 会更新同一个缓冲区, 只有当缓冲区不再足够大以容纳整行时才重新分配缓冲区。处理完文件后, 临时缓冲区会被释放。

另一个选项是`getdelim()`。它与`getline()`相同, 只是你需要指定行结束字符。只有当你的文件类型的行末字符不是"时才需要这样做。`getline()`即使在Windows文本文件中也能正常工作, 因为多字节行结束符 ("\\r") 中, "仍然是行的最后一个字符。

getline()的示例实现

```
#include <stdlib.h>  
#include <stdio.h>  
#include <errno.h>  
#include <stdint.h>  
  
#if !(defined _POSIX_C_SOURCE)  
typedef long int ssize_t;  
#endif  
  
/* 只有在没有可用的POSIX版本时才包含我们自己的getline()版本。 */  
  
#if !(defined _POSIX_C_SOURCE) || _POSIX_C_SOURCE < 200809L  
  
#if !(defined SSIZE_MAX)  
#define SSIZE_MAX (SIZE_MAX >> 1)  
#endif  
  
ssize_t getline(char **pline_buf, size_t *pn, FILE *fin)  
{  
    const size_t INITALLOC = 16;  
    const size_t ALLOCSTEP = 16;  
    size_t num_read = 0;  
  
    /* 首先检查我们的输入指针是否有 NULL。 */  
    if ((NULL == pline_buf) || (NULL == pn) || (NULL == fin))  
    {  
        errno = EINVAL;  
        return -1;  
    }  
  
    /* 如果输出缓冲区为NULL, 则分配一个缓冲区。 */  
    if (NULL == *pline_buf)
```

Output

```
line[000001]: chars=000015, buf size=000016, contents: This is a file  
line[000002]: chars=000012, buf size=000016, contents: which has  
line[000003]: chars=000015, buf size=000016, contents: multiple lines  
line[000004]: chars=000030, buf size=000032, contents: with various indentation,  
line[000005]: chars=000012, buf size=000032, contents: blank lines  
line[000006]: chars=000001, buf size=000032, contents:  
line[000007]: chars=000001, buf size=000032, contents:  
line[000008]: chars=000001, buf size=000032, contents:  
line[000009]: chars=000150, buf size=000160, contents: a really long line to show that getline()  
will reallocate the line buffer if the length of a line is too long to fit in the buffer it has  
been given,  
line[000010]: chars=000042, buf size=000160, contents: and punctuation at the end of the lines.  
line[000011]: chars=000001, buf size=000160, contents:
```

In the example, `getline()` is initially called with no buffer allocated. During this first call, `getline()` allocates a buffer, reads the first line and places the line's contents in the new buffer. On subsequent calls, `getline()` updates the same buffer and only reallocates the buffer when it is no longer large enough to fit the whole line. The temporary buffer is then freed when we are done with the file.

Another option is `getdelim()`. This is the same as `getline()` except you specify the line ending character. This is only necessary if the last character of the line for your file type is not '\n'. `getline()` works even with Windows text files because with the multibyte line ending ("\\r\\n") "\\n" is still the last character on the line.

Example implementation of getline()

```
#include <stdlib.h>  
#include <stdio.h>  
#include <errno.h>  
#include <stdint.h>  
  
#if !(defined _POSIX_C_SOURCE)  
typedef long int ssize_t;  
#endif  
  
/* Only include our version of getline() if the POSIX version isn't available. */  
  
#if !(defined _POSIX_C_SOURCE) || _POSIX_C_SOURCE < 200809L  
  
#if !(defined SSIZE_MAX)  
#define SSIZE_MAX (SIZE_MAX >> 1)  
#endif  
  
ssize_t getline(char **pline_buf, size_t *pn, FILE *fin)  
{  
    const size_t INITALLOC = 16;  
    const size_t ALLOCSTEP = 16;  
    size_t num_read = 0;  
  
    /* First check that none of our input pointers are NULL. */  
    if ((NULL == pline_buf) || (NULL == pn) || (NULL == fin))  
    {  
        errno = EINVAL;  
        return -1;  
    }  
  
    /* If output buffer is NULL, then allocate a buffer. */  
    if (NULL == *pline_buf)
```

```

{
    *pline_buf = malloc(INITALLOC);
    if (NULL == *pline_buf)
    {
        /* 无法分配内存。 */
        return -1;
    }
    else
    {
        /* 记录此时缓冲区的大小。 */
        *pn = INITALLOC;
    }
}

/* 遍历文件，读取字符直到遇到换行符或文件结束。*/
{
    int c;
    while (EOF != (c = getc(fin)))
    {
        /* 注意我们读取了一个字符。*/
        num_read++;

        /* 如果需要更多空间，则重新分配缓冲区 */
        if (num_read >= *pn)
        {
            size_t n_realloc = *pn + ALLOCSTEP;
            char * tmp = realloc(*pline_buf, n_realloc + 1); /* +1 用于末尾的NUL字符。*/
            if (NULL != tmp)
            {
                /* 使用新缓冲区并记录新缓冲区大小。*/
                *pline_buf = tmp;
                *pn = n_realloc;
            }
            else
            {
                /* 出错退出并让调用者释放缓冲区。*/
                return -1;
            }
        }

        /* 检查溢出。*/
        if (SSIZE_MAX < *pn)
        {
            errno = ERANGE;
            return -1;
        }
    }

    /* 将字符添加到缓冲区。*/
    (*pline_buf)[num_read - 1] = (char) c;

    /* 如果遇到结束字符则跳出循环。*/
    if (c == '\n'){
        break;
    }

    /* 记录是否遇到文件结束符 (EOF)。*/
    if (EOF == c)
    {
        errno = 0;
    }
}
}

{
    *pline_buf = malloc(INITALLOC);
    if (NULL == *pline_buf)
    {
        /* Can't allocate memory. */
        return -1;
    }
    else
    {
        /* Note how big the buffer is at this time. */
        *pn = INITALLOC;
    }
}

/* Step through the file, pulling characters until either a newline or EOF. */
{
    int c;
    while (EOF != (c = getc(fin)))
    {
        /* Note we read a character. */
        num_read++;

        /* Reallocate the buffer if we need more room */
        if (num_read >= *pn)
        {
            size_t n_realloc = *pn + ALLOCSTEP;
            char * tmp = realloc(*pline_buf, n_realloc + 1); /* +1 for the trailing NUL. */
            if (NULL != tmp)
            {
                /* Use the new buffer and note the new buffer size. */
                *pline_buf = tmp;
                *pn = n_realloc;
            }
            else
            {
                /* Exit with error and let the caller free the buffer. */
                return -1;
            }
        }

        /* Test for overflow. */
        if (SSIZE_MAX < *pn)
        {
            errno = ERANGE;
            return -1;
        }

        /* Add the character to the buffer. */
        (*pline_buf)[num_read - 1] = (char) c;

        /* Break from the loop if we hit the ending character. */
        if (c == '\n')
        {
            break;
        }

        /* Note if we hit EOF. */
        if (EOF == c)
        {
            errno = 0;
        }
    }
}

```

```

    return -1;
}

/* 通过添加空字符终止字符串。*/
(*pline_buf)[num_read] = '\0';

return (ssize_t) num_read;
}

#endif

```

第20.5节：fscanf()

假设我们有一个文本文件，想要读取该文件中的所有单词，以便完成一些需求。

file.txt:

这是一个
test文件
供 fscanf() 使用

这是主函数：

```

#include <stdlib.h>
#include <stdio.h>

void printAllWords(FILE *);

int main(void)
{
FILE *fp;

if ((fp = fopen("file.txt", "r")) == NULL) {
    perror("打开文件错误");
    exit(EXIT_FAILURE);
}

printAllWords(fp);

fclose(fp);

return EXIT_SUCCESS;
}

void printAllWords(FILE * fp)
{
char tmp[20];
int i = 1;

while (fscanf(fp, "%19s", tmp) != EOF) {printf("W
ord %d: %s", i, tmp); i++;
}
}

```

输出将会是：

```

    return -1;
}

/* Terminate the string by suffixing NUL. */
(*pline_buf)[num_read] = '\0';

return (ssize_t) num_read;
}

#endif

```

Section 20.5: fscanf()

Let's say we have a text file and we want to read all words in that file, in order to do some requirements.

file.txt:

This is just
a test file
to be used by fscanf()

This is the main function:

```

#include <stdlib.h>
#include <stdio.h>

void printAllWords(FILE *);

int main(void)
{
FILE *fp;

if ((fp = fopen("file.txt", "r")) == NULL) {
    perror("Error opening file");
    exit(EXIT_FAILURE);
}

printAllWords(fp);

fclose(fp);

return EXIT_SUCCESS;
}

void printAllWords(FILE * fp)
{
char tmp[20];
int i = 1;

while (fscanf(fp, "%19s", tmp) != EOF) {
    printf("Word %d: %s\n", i, tmp);
    i++;
}
}

```

The output will be:

单词 1: This
单词 2: is
单词 3: just
单词 4: a
单词 5: test
单词 6: file
单词 7: to
单词 8: be
单词 9: used
单词 10: by
单词 11: fscanf()

Word 1: This
Word 2: is
Word 3: just
Word 4: a
Word 5: test
Word 6: file
Word 7: to
Word 8: be
Word 9: used
Word 10: by
Word 11: fscanf()

第20.6节：从文件读取行

stdio.h 头文件定义了 fgets() 函数。该函数从流中读取一行并存储到指定的字符串中。当读取了 n - 1 个字符、读取到换行符 ('\n') 或到达文件末尾 (EOF) 时，函数停止读取流中的文本。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE_LENGTH 80

int main(int argc, char **argv)
{
    char *path;
    char line[MAX_LINE_LENGTH] = {0};
    unsigned int line_count = 0;

    if (argc < 1)
        return EXIT_FAILURE;
    path = argv[1];

    /* 打开文件 */
    FILE *file = fopen(path, "r");

    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* 读取每一行直到文件结束 */
    while (fgets(line, MAX_LINE_LENGTH, file))
    {
        /* 打印每一行 */
        printf("line[%06d]: %s", ++line_count, line);

        /* 为没有换行符的行添加一个尾随换行符 */
        if (line[strlen(line) - 1] != '\n') printf("\n");
    }

    /* 关闭文件 */
    if (fclose(file))
    {
        return EXIT_FAILURE;
        perror(path);
    }
}
```

Section 20.6: Read lines from a file

The stdio.h header defines the fgets() function. This function reads a line from a stream and stores it in a specified string. The function stops reading text from the stream when either n - 1 characters are read, the newline character ('\n') is read or the end of file (EOF) is reached.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE_LENGTH 80

int main(int argc, char **argv)
{
    char *path;
    char line[MAX_LINE_LENGTH] = {0};
    unsigned int line_count = 0;

    if (argc < 1)
        return EXIT_FAILURE;
    path = argv[1];

    /* Open file */
    FILE *file = fopen(path, "r");

    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Get each line until there are none left */
    while (fgets(line, MAX_LINE_LENGTH, file))
    {
        /* Print each line */
        printf("line[%06d]: %s", ++line_count, line);

        /* Add a trailing newline to lines that don't already have one */
        if (line[strlen(line) - 1] != '\n')
            printf("\n");
    }

    /* Close file */
    if (fclose(file))
    {
        return EXIT_FAILURE;
        perror(path);
    }
}
```

```
}
```

调用程序时传入的参数是一个包含以下文本的文件路径：

```
这是一个文件  
其中包含  
多行文本  
带有各种缩进,  
空白行
```

一行非常长的文本，用来展示如果一行的长度超过了缓冲区的容量，该行将被计为两行，以及行尾的标点符号。

将产生以下输出：

```
line[000001]: 这是一个文件  
line[000002]: 它包含  
line[000003]: 多行文本  
line[000004]: 带有各种缩进,  
line[000005]: 空白行  
line[000006]:  
line[000007]:  
line[000008]:  
line[000009]: 一行非常长的文本，用来展示如果一行的长度超过了缓冲区的容量，该行将被计为两行，  
line[000011]: 行尾的标点符号。  
line[000012]:
```

这个非常简单的例子允许固定的最大行长度，因此较长的行实际上会被计为两行。fgets()函数要求调用代码提供用于存放读取行的内存。

POSIX 提供了 getline() 函数，该函数内部会根据需要分配内存以扩大缓冲区，从而支持任意长度的行（只要有足够的内存）。

第20.7节：打开并写入二进制文件

```
#include <stdlib.h>  
#include <stdio.h>  
  
int main(void)  
{  
    result = EXIT_SUCCESS;  
  
    char file_name[] = "outbut.bin";  
    char str[] = "This is a binary file example";  
    FILE * fp = fopen(file_name, "wb");  
  
    if (fp == NULL) /* 如果文件创建过程中发生错误 */  
    {  
        result = EXIT_FAILURE;  
        fprintf(stderr, "fopen() failed for '%s'\n", file_name);  
    }  
}
```

```
}
```

Calling the program with an argument that is a path to a file containing the following text:

```
This is a file  
which has  
multiple lines  
with various indentation,  
blank lines
```

a really long line to show that the line will be counted as two lines if the length of a line is too long to fit in the buffer it has been given, and punctuation at the end of the lines.

Will result in the following output:

```
line[000001]: This is a file  
line[000002]: which has  
line[000003]: multiple lines  
line[000004]: with various indentation,  
line[000005]: blank lines  
line[000006]:  
line[000007]:  
line[000008]:  
line[000009]: a really long line to show that the line will be counted as two lines if the length of a line is too long to fit in the buffer it has been given,  
line[000010]: and punctuation at the end of the lines.  
line[000011]:  
line[000012]:
```

This very simple example allows a fixed maximum line length, such that longer lines will effectively be counted as two lines. The fgets() function requires that the calling code provide the memory to be used as the destination for the line that is read.

POSIX makes the getline() function available which instead internally allocates memory to enlarge the buffer as necessary for a line of any length (as long as there is sufficient memory).

Section 20.7: Open and write to a binary file

```
#include <stdlib.h>  
#include <stdio.h>  
  
int main(void)  
{  
    result = EXIT_SUCCESS;  
  
    char file_name[] = "outbut.bin";  
    char str[] = "This is a binary file example";  
    FILE * fp = fopen(file_name, "wb");  
  
    if (fp == NULL) /* If an error occurs during the file creation */  
    {  
        result = EXIT_FAILURE;  
        fprintf(stderr, "fopen() failed for '%s'\n", file_name);  
    }  
}
```

```

}
else
{
    size_t element_size = sizeof *str;
    size_t elements_to_write = sizeof str;

    /* 将字符串 (包括NUL终止符) 写入二进制文件。 */
    size_t elements_written = fwrite(str, element_size, elements_to_write, fp);
    if (elements_written != elements_to_write)
    {
        result = EXIT_FAILURE;
        /* 这仅适用于 >=c99, 否则 z 长度修饰符未知。 */
        fprintf(stderr, "fwrite() 失败: 只写入了 %zu 个元素, 共 %zu 个元素。", elements_written, elements_to_write);
        /* 对于 <c99 使用此方法: */
        fprintf(stderr, "fwrite() 失败: 只写入了 %lu 个元素, 共 %lu 个元素。", (unsigned long) elements_written, (unsigned long) elements_to_write);
        /*
         */
    }
    fclose(fp);
}

return result;
}

```

该程序通过 `fwrite` 函数以二进制形式创建并写入文本到文件 `output.bin`。

如果已存在同名文件，其内容将被丢弃，文件被视为新的空文件。

二进制流是一个有序的字符序列，可以透明地记录内部数据。在这种模式下，字节在程序和文件之间写入时不进行任何解释。

为了便携地写入整数，必须知道文件格式是期望大端还是小端格式，以及大小（通常为16、32或64位）。然后可以使用位移和掩码来按正确顺序写出字节。C语言中的整数不保证采用二补数表示（尽管几乎所有实现都是如此）。幸运的是，转换为无符号数是保证使用二补数的。因此，将有符号整数写入二进制文件的代码有些令人惊讶。

```

/* 写入一个16位小端整数 */
int fput16le(int x, FILE *fp)
{
    unsigned int rep = x;
    int e1, e2;

    e1 = fputc(rep & 0xFF, fp);
    e2 = fputc((rep >> 8) & 0xFF, fp);

    if(e1 == EOF || e2 == EOF)
        return EOF;
    return 0;
}

```

其他函数遵循相同的模式，只是在大小和字节顺序上有细微修改。

```

}
else
{
    size_t element_size = sizeof *str;
    size_t elements_to_write = sizeof str;

    /* Writes str (_including_ the NUL-terminator) to the binary file. */
    size_t elements_written = fwrite(str, element_size, elements_to_write, fp);
    if (elements_written != elements_to_write)
    {
        result = EXIT_FAILURE;
        /* This works for >=c99 only, else the z length modifier is unknown. */
        fprintf(stderr, "fwrite() failed: wrote only %zu out of %zu elements.\n",
                elements_written, elements_to_write);
        /* Use this for <c99: */
        fprintf(stderr, "fwrite() failed: wrote only %lu out of %lu elements.\n",
                (unsigned long) elements_written, (unsigned long) elements_to_write);
    }
    fclose(fp);
}

return result;
}

```

This program creates and writes text in the binary form through the `fwrite` function to the file `output.bin`.

If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.

A binary stream is an ordered sequence of characters that can transparently record internal data. In this mode, bytes are written between the program and the file without any interpretation.

To write integers portably, it must be known whether the file format expects them in big or little-endian format, and the size (usually 16, 32 or 64 bits). Bit shifting and masking may then be used to write out the bytes in the correct order. Integers in C are not guaranteed to have two's complement representation (though almost all implementations do). Fortunately a conversion to `unsigned` is guaranteed to use two's complement. The code for writing a signed integer to a binary file is therefore a little surprising.

```

/* write a 16-bit little endian integer */
int fput16le(int x, FILE *fp)
{
    unsigned int rep = x;
    int e1, e2;

    e1 = fputc(rep & 0xFF, fp);
    e2 = fputc((rep >> 8) & 0xFF, fp);

    if(e1 == EOF || e2 == EOF)
        return EOF;
    return 0;
}

```

The other functions follow the same pattern with minor modifications for size and byte order.

第21章：格式化输入/输出

第21.1节：打印的转换说明符

转换说明符	参数类型	描述
i, d	int	打印十进制
u	无符号整数	打印十进制
o	无符号整数	打印八进制
x	无符号整数	打印十六进制，小写
X	无符号整数	打印十六进制，大写
f	double	以默认精度6打印浮点数，若未指定精度（小写用于特殊数字 nan 和 inf 或 infinity）
F	double	以默认精度6打印浮点数，若未指定精度（大写用于特殊数字 NAN 和 INF 或 INFINITY）
e	double	以默认精度6打印浮点数，若未指定精度，使用科学计数法（尾数/指数）；小写指数和特殊数字
E	double	以默认精度6打印浮点数，若未指定精度，使用科学计数法（尾数/指数）；大写指数和特殊数字
g	double	使用 f 或 e [见下文]
G	double	使用F或E[见下文]
a	double	打印十六进制，小写
A	double	打印十六进制，大写
c	char	打印单个字符
s	char*	打印以NUL结尾的字符串，或如果指定了精度，则截断到给定长度
p	void*	打印void指针值；非void指针应显式转换（“强制类型转换”）为void*；仅限指向对象的指针，不是函数指针
%	不适用	打印%字符
n	int *	将到目前为止打印的字节数写入指向int的指针中。

注意，长度修饰符可以应用于%n（例如，%hhn表示后续的 n 转换说明符适用于指向有符号字符参数的指针，根据ISO/IEC 9899:2011§7.21.6.1¶7）。

注意，浮点转换适用于float和double类型，因为默认提升规则——§6.5.2.2 函数调用，¶7函数原型声明中的省略号表示参数类型转换在最后一个声明参数后停止。对尾随参数执行默认参数提升。因此，诸如printf()等函数传递的值总是double类型，即使引用的变量是float类型。

对于g和G格式，选择 e 和 f （或 E 和 F ）表示法的规则在C标准和POSIX规范中对printf()进行了说明：

表示浮点数的双精度参数应根据转换的值和精度，以 f 或 e 样式（对于 G 转换说明符则为 F 或 E 样式）进行转换。

设P等于非零的精度，若省略精度则为6，若精度为零则为1。然后，如果使用 E 样式转换的指数为 X :

- 如果 P > X >= -4，则转换应采用样式 f (或 F) 和精度 P - (X+1)。
- 否则，转换应采用样式 e (或 E) 和精度 P - 1。

Chapter 21: Formatted Input/Output

Section 21.1: Conversion Specifiers for printing

Conversion Specifier	Type of Argument	Description
i, d	int	prints decimal
u	unsigned int	prints decimal
o	unsigned int	prints octal
x	unsigned int	prints hexadecimal, lower-case
X	unsigned int	prints hexadecimal, upper-case
f	double	prints float with a default precision of 6, if no precision is given (lower-case used for special numbers nan and inf or infinity)
F	double	prints float with a default precision of 6, if no precision is given (upper-case used for special numbers NAN and INF or INFINITY)
e	double	prints float with a default precision of 6, if no precision is given, using scientific notation using mantissa/exponent; lower-case exponent and special numbers
E	double	prints float with a default precision of 6, if no precision is given, using scientific notation using mantissa/exponent; upper-case exponent and special numbers
g	double	uses either f or e [see below]
G	double	uses either F or E [see below]
a	double	prints hexadecimal, lower-case
A	double	prints hexadecimal, upper-case
c	char	prints single character
s	char*	prints string of characters up to a NUL terminator, or truncated to length given by precision, if specified
p	void*	prints void-pointer value; a nonvoid-pointer should be explicitly converted ("cast") to void*; pointer to object only, not a function-pointer
%	n/a	prints the % character
n	int *	write the number of bytes printed so far into the int pointed at.

Note that length modifiers can be applied to %n (e.g. %hhn indicates that a following n conversion specifier applies to a pointer to a signed char argument, according to the ISO/IEC 9899:2011 §7.21.6.1 ¶7).

Note that the floating point conversions apply to types float and double because of default promotion rules — §6.5.2.2 Function calls, ¶7 The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.) Thus, functions such as printf() are only ever passed double values, even if the variable referenced is of type float.

With the g and G formats, the choice between e and f (or E and F) notation is documented in the C standard and in the POSIX specification for printf():

The double argument representing a floating-point number shall be converted in the style f or e (or in the style F or E in the case of a G conversion specifier), depending on the value converted and the precision. Let P equal the precision if non-zero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style E would have an exponent of X:

- If P > X >= -4, the conversion shall be with style f (or F) and precision P - (X+1).
- Otherwise, the conversion shall be with style e (or E) and precision P - 1.

最后，除非使用了 '#' 标志，否则应从结果的小数部分去除所有尾随的零，且如果没有剩余的小数部分，则应去除小数点字符。

Finally, unless the '#' flag is used, any trailing zeros shall be removed from the fractional portion of the result and the decimal-point character shall be removed if there is no fractional portion remaining.

第21.2节：printf()函数

通过包含<stdio.h>访问，函数printf()是C语言中用于向控制台打印文本的主要工具。

```
printf("Hello world!");
// Hello world!
```

普通的未格式化字符数组可以直接放在括号内打印。

```
printf("%d 是生命、宇宙以及一切的答案.", 42);
// 42 是生命、宇宙以及一切的答案。
```

```
int x = 3;
char y = 'Z';
char* z = "Example";
printf("Int: %d, Char: %c, String: %s", x, y, z);
// Int: 3, Char: Z, String: Example
```

另外，整数、浮点数、字符等可以使用转义字符%，后跟表示格式的字符或字符序列，即格式说明符，进行打印。

函数printf()的所有附加参数用逗号分隔，这些参数应与格式说明符的顺序一致。多余的参数会被忽略，而参数类型错误或缺少参数会导致错误或未定义行为。每个参数可以是字面值或变量。

成功执行后，返回打印的字符数，类型为int。否则，失败时返回负值。

Section 21.2: The printf() Function

Accessed through including <stdio.h>, the function printf() is the primary tool used for printing text to the console in C.

```
printf("Hello world!");
// Hello world!
```

Normal, unformatted character arrays can be printed by themselves by placing them directly in between the parentheses.

```
printf("%d is the answer to life, the universe, and everything.", 42);
// 42 is the answer to life, the universe, and everything.
```

```
int x = 3;
char y = 'Z';
char* z = "Example";
printf("Int: %d, Char: %c, String: %s", x, y, z);
// Int: 3, Char: Z, String: Example
```

Alternatively, integers, floating-point numbers, characters, and more can be printed using the escape character %, followed by a character or sequence of characters denoting the format, known as the *format specifier*.

All additional arguments to the function printf() are separated by commas, and these arguments should be in the same order as the format specifiers. Additional arguments are ignored, while incorrectly typed arguments or a lack of arguments will cause errors or undefined behavior. Each argument can be either a literal value or a variable.

After successful execution, the number of characters printed is returned with type int. Otherwise, a failure returns a negative value.

第21.3节：打印格式标志

C标准（C11及C99）为printf()定义了以下标志：

标志	转换	含义
-	全部	转换结果应在字段内左对齐。如果未指定此标志，则转换结果右对齐。
+	带符号数字	带符号转换的结果应始终以符号（'+' 或 '-'）开头。如果未指定此标志，则仅在转换负值时以符号开头。
<space>	带符号数字	如果带符号转换的第一个字符不是符号，或者带符号转换结果为空，则应在结果前加一个<space>。这意味着如果同时出现<space>和'+'标志，则忽略<space>标志。
#	全部	指定将值转换为另一种形式。对于 o 转换，只有在必要时才会增加精度，以强制结果的第一个数字为零（如果值和精度均为0，则打印单个0）。对于 x 或 X 转换说明符，非零结果应以 0x（或 0X）为前缀。对于 a、A、e、E、f、F、g 和 G 转换说明符，结果应始终包含小数点，即使小数点后没有数字。没有此标志时，这些转换的结果中只有在小数点后有数字时才会出现小数点。对于 g 和 G 转换说明符，结果中的尾随零不应像通常那样被移除。对于其他转换说明符，行为未定义。

The C standard (C11, and C99 too) defines the following flags for printf():

Flag	Conversions	Meaning
-	all	The result of the conversion shall be left-justified within the field. The conversion is right-justified if this flag is not specified.
+	signed numeric	The result of a signed conversion shall always begin with a sign ('+' or '-'). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.
<space>	signed numeric	If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a <space> shall be prefixed to the result. This means that if the <space> and '+' flags both appear, the <space> flag shall be ignored.
#	all	Specifies that the value is to be converted to an alternative form. For o conversion, it shall increase the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For x or X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A, e, E, f, F, g, and G conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversion specifiers, trailing zeros shall not be removed from the result as they normally are. For other conversion specifiers, the behavior is undefined.

对于 d、i、o、u、x、X、a、A、e、E、f、F、g 和 G 转换说明符，前导零（紧跟任何符号或进制指示之后）用于填充字段宽度，而不是使用空格填充，除非转换的是无穷大或 NaN。如果同时出现 '0' 和 '-' 标志，则忽略 '0' 标志。对于 d、i、o、u、x 和 X 转换说明符，如果指定了精度，则应忽略 '0' 标志。 \bowtie 如果同时出现 '0' 和 <apostrophe> 标志，则分组字符插入在零填充之前。对于其他转换，行为未定义。 \bowtie

这些标志也得到了微软的支持，含义相同。

POSIX 规范中对 printf() 的补充说明：

标志转换

含义

十进制转换结果的整数部分应采用千位分隔符格式

i, d, u, f, F, g, G 分组字符。对于其他转换，行为未定义。使用非货币分组字符。

For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the '0' and '-' flags both appear, the '0' flag is ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag shall be ignored. \bowtie If the '0' and <apostrophe> flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined. \bowtie

These flags are also supported by Microsoft with the same meanings.

The POSIX specification for [printf\(\)](#) adds:

Flag Conversions

Meaning

The integer portion of the result of a decimal conversion shall be formatted with thousands' i, d, u, f, F, g, G grouping characters. For other conversions the behavior is undefined. The non-monetary grouping character is used.

第21.4节：打印指向对象的指针的值

要打印指向对象的指针的值（与函数指针相对），请使用 p 转换说明符。它被定义为仅打印 void 指针，因此要打印非 void 指针的值，需要显式地将其转换（“强制类型转换”）为 void*。

```
#include <stdlib.h> /* 用于 EXIT_SUCCESS */
#include <stdio.h> /* 用于 printf() */

int main(void)
{
    int i;
    int * p = &i;

    printf("i 的地址是 %p.", (void*) p);return EXIT_SUCCESS;
}
```

版本 ≥ C99

使用 <inttypes.h> 和 uintptr_t

在 C99 或更高版本中打印指针的另一种方法是使用 uintptr_t 类型和 <inttypes.h> 中的宏：

```
#include <inttypes.h> /* 用于 uintptr_t 和 PRIXPTR */
#include <stdio.h> /* 用于 printf() */

int main(void)
{
    int i;
    int *p = &i;

    printf("i 的地址是 0x%" PRIXPTR ". ", (uintptr_t)p);return 0;
}
```

理论上，可能不存在能够保存任何转换为整数的指针的整数类型（因此类型 **uintptr_t** 可能不存在）。实际上，它是存在的。函数指针不一定能转换为 **uintptr_t** 类型—尽管它们通常是可转换的。

如果 **uintptr_t** 类型存在，那么 **intptr_t** 类型也存在。不过不清楚为什么你会想把地址当作

Section 21.4: Printing the Value of a Pointer to an Object

To print the value of a pointer to an object (as opposed to a function pointer) use the p conversion specifier. It is defined to print **void**-pointers only, so to print out the value of a non **void**-pointer it needs to be explicitly converted ("casted") to **void***.

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */

int main(void)
{
    int i;
    int * p = &i;

    printf("The address of i is %p.\n", (void*) p);

    return EXIT_SUCCESS;
}
```

Version ≥ C99

Using <inttypes.h> and uintptr_t

Another way to print pointers in C99 or later uses the **uintptr_t** type and the macros from <inttypes.h>:

```
#include <inttypes.h> /* for uintptr_t and PRIXPTR */
#include <stdio.h> /* for printf() */

int main(void)
{
    int i;
    int *p = &i;

    printf("The address of i is 0x%" PRIXPTR ".\n", (uintptr_t)p);

    return 0;
}
```

In theory, there might not be an integer type that can hold any pointer converted to an integer (so the type **uintptr_t** might not exist). In practice, it does exist. Pointers to functions need not be convertible to the **uintptr_t** type — though again they most often are convertible.

If the **uintptr_t** type exists, so does the **intptr_t** type. It is not clear why you'd ever want to treat addresses as

有符号整数来处理。

版本 = K&R 版本 < C89

标准之前的历史：

在C89之前的K&R-C时代，没有void*类型（也没有头文件<stdlib.h>，没有函数原型，因此也没有int main(void)的写法），所以指针被强制转换为long unsigned int，并使用lx长度修饰符/转换说明符打印。

下面的示例仅供参考。如今这段代码是无效的，很可能会引发臭名昭著的未定义行为。

```
#include <stdio.h> /* 在标准之前的C中可选 - 用于printf() */

int main()
{
    int i;
    int *p = &i;

    printf("i 的地址是 0x%lx.", (long unsigned) p);return 0;
}
```

第21.5节：打印两个指向同一对象的指针值之差

两个指向同一对象的指针值相减会得到一个有符号整数*1。因此，至少应使用 d转换说明符来打印该值。

为了确保有足够的类型来存储这种“指针差值”，C99标准在<stddef.h>中定义了类型ptrdiff_t。打印ptrdiff_t类型时应使用 t长度修饰符。

版本 ≥ C99

```
#include <stdlib.h> /* 用于EXIT_SUCCESS */
#include <stdio.h> /* 用于printf() */
#include <stddef.h> /* 用于ptrdiff_t */

int main(void)
{
    int a[2];
    int * p1 = &a[0], * p2 = &a[1];
    ptrdiff_t pd = p2 - p1;

    printf("p1 = %p, (void*) p1);printf("p2
= %p, (void*) p2);printf("p2 - p1 = %t
d", pd);return EXIT_SUCCESS;
}
```

结果可能如下所示：

```
p1 = 0x7fff6679f430
p2 = 0x7fff6679f434
p2 - p1 = 1
```

signed integers, though.

Version = K&R Version < C89

Pre-Standard History:

Prior to C89 during K&R-C times there was no type void* (nor header <stdlib.h>, nor prototypes, and hence no int main(void) notation), so the pointer was cast to long unsigned int and printed using the lx length modifier/conversion specifier.

The example below is just for informational purpose. Nowadays this is invalid code, which very well might provoke the infamous Undefined Behaviour.

```
#include <stdio.h> /* optional in pre-standard C - for printf() */

int main()
{
    int i;
    int *p = &i;

    printf("The address of i is 0x%lx.\n", (long unsigned) p);

    return 0;
}
```

Section 21.5: Printing the Difference of the Values of two Pointers to an Object

Subtracting the values of two pointers to an object results in a signed integer *1. So it would be printed using at least the d conversion specifier.

To make sure there is a type being wide enough to hold such a "pointer-difference", since C99 <stddef.h> defines the type ptrdiff_t. To print a ptrdiff_t use the t length modifier.

Version ≥ C99

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */
#include <stddef.h> /* for ptrdiff_t */

int main(void)
{
    int a[2];
    int * p1 = &a[0], * p2 = &a[1];
    ptrdiff_t pd = p2 - p1;

    printf("p1 = %p\n", (void*) p1);
    printf("p2 = %p\n", (void*) p2);
    printf("p2 - p1 = %td\n", pd);

    return EXIT_SUCCESS;
}
```

The result might look like this:

```
p1 = 0x7fff6679f430
p2 = 0x7fff6679f434
p2 - p1 = 1
```

请注意，差值的结果会根据指针所指向类型的大小进行缩放，这里指的是int类型。此例中int的大小为4。

*1如果要相减的两个指针不指向同一个对象，则行为未定义。

第21.6节：长度修饰符

C99和C11标准为printf()指定了以下长度修饰符；它们的含义是：

修饰符	修饰	适用于
hh	d、i、o、u、x或X	char, signed char或unsigned char
h	d、i、o、u、x或X	short int或unsigned short int
l	d、i、o、u、x或X	long int或unsigned long int
l	a, A, e, E, f, F, g, 或 G double (为兼容 scanf()；在 C90 中未定义)	
ll	d、i、o、u、x或X	长长整型或无符号长长整型
j	d、i、o、u、x或X	intmax_t或uintmax_t
z	d、i、o、u、x或X	size_t或对应的有符号类型 (POSIX中的ssize_t)
t	d、i、o、u、x或X	ptrdiff_t或对应的无符号整型
L	a、A、E、F、g或G长双精度浮点型	

如果长度修饰符与上述指定以外的任何转换说明符一起出现，行为未定义。

微软指定了一些不同的长度修饰符，并明确不支持 hh、j、z或t。

修饰符	修饰	适用范围
I32	d、i、o、x 或 X	__int32
I32	o、u、x 或 X	unsigned __int32
I64	d、i、o、x 或 X	__int64
I64	o、u、x 或 X	unsigned __int64
I	d、i、o、x 或 X	ptrdiff_t (即 32 位平台上的__int32, 64 位平台上的__int64)
I	o、u、x 或 X	size_t (即 32 位平台上的unsigned __int32, 64 位平台上的unsigned __int64)
I 或 L	a、A、E、f、g 或 G long double (在 Visual C++ 中，虽然long double是一个独立类型，但其内部表示与double相同)	
c 或 C	使用printf和wprintf函数的宽字符。(lc、lC、wc或wC类型说明符在printf函数中与C同义，在wprintf函数中与c同义。)	
I 或 w	s、S 或 Z	使用printf和wprintf函数的宽字符字符串。(ls、lS、ws或wS类型说明符在printf函数中与S同义，在wprintf函数中与s同义。)

请注意，C、S 和Z转换说明符以及I、I32、I64 和w长度修饰符是微软的扩展。

将I视为long double的修饰符而非double的修饰符与标准不同，尽管除非long double与double的表示不同，否则你很难察觉差异。

Please note that the resulting value of the difference is scaled by the size of the type the pointers subtracted point to, an int here. The size of an int for this example is 4.

*1If the two pointers to be subtracted do not point to the same object the behaviour is undefined.

Section 21.6: Length modifiers

The C99 and C11 standards specify the following length modifiers for printf(); their meanings are:

Modifier	Modifies	Applies to
hh	d, i, o, u, x, or X	char, signed char or unsigned char
h	d, i, o, u, x, or X	short int or unsigned short int
l	d, i, o, u, x, or X	long int or unsigned long int
l	a, A, e, E, f, F, g, or G double (for compatibility with scanf(); undefined in C90)	
ll	d, i, o, u, x, or X	long long int or unsigned long long int
j	d, i, o, u, x, or X	intmax_t or uintmax_t
z	d, i, o, u, x, or X	size_t or the corresponding signed type (ssize_t in POSIX)
t	d, i, o, u, x, or X	ptrdiff_t or the corresponding unsigned integer type
L	a, A, e, E, f, F, g, or G long double	

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

Microsoft specifies some different length modifiers, and explicitly does not support hh, j, z, or t.

Modifier	Modifies	Applies to
I32	d, i, o, x, or X	__int32
I32	o, u, x, or X	unsigned __int32
I64	d, i, o, x, or X	__int64
I64	o, u, x, or X	unsigned __int64
I	d, i, o, x, or X	ptrdiff_t (that is, __int32 on 32-bit platforms, __int64 on 64-bit platforms)
I	o, u, x, or X	size_t (that is, unsigned __int32 on 32-bit platforms, unsigned __int64 on 64-bit platforms)
I or L	a, A, e, E, f, g, or G long double (In Visual C++, although long double is a distinct type, it has the same internal representation as double.)	long double (In Visual C++, although long double is a distinct type, it has the same internal representation as double.)
I or w	c or C	Wide character with printf and wprintf functions. (An lc, lC, wc or wC type specifier is synonymous with C in printf functions and with c in wprintf functions.)
I or w	s, S, or Z	Wide-character string with printf and wprintf functions. (An ls, lS, ws or wS type specifier is synonymous with S in printf functions and with s in wprintf functions.)

Note that the C, S, and Z conversion specifiers and the I, I32, I64, and w length modifiers are Microsoft extensions. Treating l as a modifier for long double rather than double is different from the standard, though you'll be hard-pressed to spot the difference unless long double has a different representation from double.

第22章：指针

指针是一种变量类型，可以存储另一个对象或函数的地址。

第22.1节：介绍

指针的声明方式与其他变量类似，只是在类型和变量名之间放置一个星号 (*) 以表示它是一个指针。

```
int *pointer; /* 在函数内部, pointer未初始化, 尚未指向任何有效对象 */
```

要在同一声明中声明两个相同类型的指针变量，需要在每个标识符前使用星号符号。例如，

```
int *iptr1, *iptr2;  
int *iptr3, iptr4; /* iptr3是指针变量, 而iptr4命名错误, 是int类型 */
```

取地址或引用运算符用符号& (&) 表示，返回给定变量的地址，该地址可以存放在相应类型的指针中。

```
int value = 1;  
pointer = &value;
```

间接或解引用运算符用星号 (*) 表示，用于获取指针所指向对象的内容。

```
printf("指针指向的整数值: %d", *pointer);  
/* 指向的整数的值: 1 */
```

如果指针指向结构体或联合体类型，则可以对其解引用并使用->操作符直接访问其成员：

```
SomeStruct *s = &someObject;  
s->someMember = 5; /* 等同于 (*s).someMember = 5 */
```

在C语言中，指针是一种独立的值类型，可以重新赋值，并且被视为一个独立的变量。例如，下面的示例打印指针（变量）本身的价值。

```
printf("指针本身的值: %p", (void *)pointer);  
/* 指针本身的值: 0x7ffcd41b06e4 */  
/* 每次程序执行时该地址都会不同 */
```

因为指针是可变的变量，所以它可能不指向有效对象，可能被设置为null

```
pointer = 0; /* 或者 */  
pointer = NULL;
```

或者仅仅包含一个不是有效地址的任意位模式。后者是非常糟糕的情况，因为在指针被解引用之前无法进行测试，只有指针为空的情况可以测试：

```
if (!pointer) exit(EXIT_FAILURE);
```

Chapter 22: Pointers

A pointer is a type of variable which can store the address of another object or a function.

Section 22.1: Introduction

A pointer is declared much like any other variable, except an asterisk (*) is placed between the type and the name of the variable to denote it is a pointer.

```
int *pointer; /* inside a function, pointer is uninitialized and doesn't point to any valid object yet */
```

To declare two pointer variables of the same type, in the same declaration, use the asterisk symbol before each identifier. For example,

```
int *iptr1, *iptr2;  
int *iptr3, iptr4; /* iptr3 is a pointer variable, whereas iptr4 is misnamed and is an int */
```

The address-of or reference operator denoted by an ampersand (&) gives the address of a given variable which can be placed in a pointer of appropriate type.

```
int value = 1;  
pointer = &value;
```

The indirection or dereference operator denoted by an asterisk (*) gets the contents of an object pointed to by a pointer.

```
printf("Value of pointed to integer: %d\n", *pointer);  
/* Value of pointed to integer: 1 */
```

If the pointer points to a structure or union type then you can dereference it and access its members directly using the -> operator:

```
SomeStruct *s = &someObject;  
s->someMember = 5; /* Equivalent to (*s).someMember = 5 */
```

In C, a pointer is a distinct value type which can be reassigned and otherwise is treated as a variable in its own right. For example the following example prints the value of the pointer (variable) itself.

```
printf("Value of the pointer itself: %p\n", (void *)pointer);  
/* Value of the pointer itself: 0x7ffcd41b06e4 */  
/* This address will be different each time the program is executed */
```

Because a pointer is a mutable variable, it is possible for it to not point to a valid object, either by being set to null

```
pointer = 0; /* or alternatively */  
pointer = NULL;
```

or simply by containing an arbitrary bit pattern that isn't a valid address. The latter is a very bad situation, because it cannot be tested before the pointer is being dereferenced, there is only a test for the case a pointer is null:

```
if (!pointer) exit(EXIT_FAILURE);
```

只有当指针指向一个有效对象时，才能对其进行解引用，否则行为是未定义的。许多现代实现可能会通过引发某种错误（如段错误）来帮助你并终止执行，但其他实现可能只是让你的程序处于无效状态。

解引用操作符返回的值是原始变量的可变别名，因此可以更改，从而修改原始变量。

```
*pointer += 1;
printf("Value of pointed to variable after change: %d", *pointer);
/* Value of pointed to variable after change: 2 */
```

指针也是可重新赋值的。这意味着指向一个对象的指针以后可以用来指向同一类型的另一个对象。

```
int value2 = 10;
指针 = &value2;
printf("指针指向的值: %d", *pointer);
/* 指针指向的值: 10 */
```

像其他变量一样，指针也有特定的类型。例如，你不能将short int的地址赋给long int类型的指针。这种行为称为类型混用（type punning），在C语言中是禁止的，尽管有一些例外情况。

虽然指针必须是特定类型，但为每种类型的指针分配的内存大小等于环境用于存储地址的内存大小，而不是所指向类型的大小。

```
#include <stdio.h>

int main(void) {
    printf("int指针的大小: %zu", sizeof(int*)); /* 大小4字节 */
    printf("int变量的大小: %zu", sizeof(int));
    printf("char指针的大小: %zu", sizeof(char*)); /* 大小4字节 */
    printf("char变量的大小: %zu", sizeof(char)); /* 大小1字节 */
    printf("short指针的大小: %zu", sizeof(short*)); /* 大小4字节 */
    printf("short变量的大小: %zu", sizeof(short)); /* 大小2字节 */
    return 0;
}
```

(注意：如果你使用的是不支持C99或C11标准的Microsoft Visual Studio，则必须在上述示例中使用%Iu1代替%zu。)

请注意，上述结果在不同环境中数值可能有所不同，但所有环境中不同类型指针的大小都是相等的。

摘自卡迪夫大学C语言指针入门 ([Cardiff University C Pointers Introduction](#))

指针与数组

指针和数组在C语言中密切相关。C语言中的数组总是存储在内存的连续位置。指针算术运算总是按所指向项的大小进行缩放。因此，如果我们有一个包含三个双精度浮点数的数组，以及一个指向数组首元素的指针，*ptr 指向第一个双精度数，*(ptr + 1) 指向第二个，*(ptr + 2) 指向第三个。更方便的表示法是使用数组符号 []。

```
double point[3] = {0.0, 1.0, 2.0};
double *ptr = point;
```

A pointer may only be dereferenced if it points to a *valid* object, otherwise the behavior is undefined. Many modern implementations may help you by raising some kind of error such as a [segmentation fault](#) and terminate execution, but others may just leave your program in an invalid state.

The value returned by the dereference operator is a mutable alias to the original variable, so it can be changed, modifying the original variable.

```
*pointer += 1;
printf("Value of pointed to variable after change: %d\n", *pointer);
/* Value of pointed to variable after change: 2 */
```

Pointers are also re assignable. This means that a pointer pointing to an object can later be used to point to another object of the same type.

```
int value2 = 10;
pointer = &value2;
printf("Value from pointer: %d\n", *pointer);
/* Value from pointer: 10 */
```

Like any other variable, pointers have a specific type. You can't assign the address of a *short int* to a pointer to a *long int*, for instance. Such behavior is referred to as type punning and is forbidden in C, though there are a few exceptions.

Although pointer must be of a specific type, the memory allocated for each type of pointer is equal to the memory used by the environment to store addresses, rather than the size of the type that is pointed to.

```
#include <stdio.h>

int main(void) {
    printf("Size of int pointer: %zu\n", sizeof(int*)); /* size 4 bytes */
    printf("Size of int variable: %zu\n", sizeof(int)); /* size 4 bytes */
    printf("Size of char pointer: %zu\n", sizeof(char*)); /* size 4 bytes */
    printf("Size of char variable: %zu\n", sizeof(char)); /* size 1 bytes */
    printf("Size of short pointer: %zu\n", sizeof(short*)); /* size 4 bytes */
    printf("Size of short variable: %zu\n", sizeof(short)); /* size 2 bytes */
    return 0;
}
```

(NB: if you are using Microsoft Visual Studio, which does not support the C99 or C11 standards, you must use %Iu1 instead of %zu in the above sample.)

Note that the results above can vary from environment to environment in numbers but all environments would show equal sizes for different types of pointer.

Extract based on information from [Cardiff University C Pointers Introduction](#)

Pointers and Arrays

Pointers and arrays are intimately connected in C. Arrays in C are always held in contiguous locations in memory. Pointer arithmetic is always scaled by the size of the item pointed to. So if we have an array of three doubles, and a pointer to the base, *ptr refers to the first double, *(ptr + 1) to the second, *(ptr + 2) to the third. A more convenient notation is to use array notation [].

```
double point[3] = {0.0, 1.0, 2.0};
double *ptr = point;
```

```
/* 输出 x 0.0, y 1.0 z 2.0 */
printf("x %f y %f z %f", ptr[0], ptr[1], ptr[2]);
```

所以本质上 `ptr` 和数组名是可以互换的。这个规则也意味着数组在传递给子程序时会退化为指针。

```
double point[3] = {0.0, 1.0, 2.0}; printf("length of point is %s", length(point));

/* 获取三维点到原点的距离 */
double length(double *pt)
{
    return sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2]);
}
```

指针可以指向数组中的任意元素，或者指向最后一个元素之后的位置。然而，将指针设置为其他任何值都是错误的，包括指向数组之前的元素。（原因是在分段架构中，第一个元素之前的地址可能跨越段边界，编译器确保这种情况不会发生在最后一个元素加一的位置）。

脚注1：Microsoft格式信息可通过`printf()`和格式说明语法获取。

第22.2节：常见错误

指针使用不当常常是导致漏洞（包括安全漏洞）或程序崩溃的原因，最常见的是由于段错误引起的。

未检查分配失败

内存分配不保证一定成功，可能返回一个NULL指针。使用返回值时，如果不检查分配是否成功，就会导致未定义行为。通常这会导致程序崩溃，但并不保证一定会崩溃，因此依赖崩溃也可能引发问题。

例如，不安全的写法：

```
struct SomeStruct *s = malloc(sizeof *s);
s->someValue = 0; /* 不安全，因为s可能是空指针 */
```

安全的写法：

```
struct SomeStruct *s = malloc(sizeof *s);
if (s)
{
    s->someValue = 0; /* 这是安全的，我们已经检查了s是有效的 */
}
```

请求内存时使用字面数字代替`sizeof`

对于给定的编译器/机器配置，类型具有已知的大小；然而，没有任何标准规定除`char`之外的某个类型的大小在所有编译器/机器配置中都相同。如果代码在内存分配时使用4而不是`sizeof(int)`，代码可能在原始机器上运行正常，但不一定能移植到其他机器或编译器。类型的固定大小应替换为

```
/* prints x 0.0, y 1.0 z 2.0 */
printf("x %f y %f z %f\n", ptr[0], ptr[1], ptr[2]);
```

So essentially `ptr` and the array name are interchangeable. This rule also means that an array decays to a pointer when passed to a subroutine.

```
double point[3] = {0.0, 1.0, 2.0};

printf("length of point is %s\n", length(point));

/* get the distance of a 3D point from the origin */
double length(double *pt)
{
    return sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2]);
}
```

A pointer may point to any element in an array, or to the element beyond the last element. It is however an error to set a pointer to any other value, including the element before the array. (The reason is that on segmented architectures the address before the first element may cross a segment boundary, the compiler ensures that does not happen for the last element plus one).

Footnote 1: Microsoft format information can be found via `printf()` and [format specification syntax](#).

Section 22.2: Common errors

Improper use of pointers are frequently a source of bugs that can include security bugs or program crashes, most often due to segmentation faults.

Not checking for allocation failures

Memory allocation is not guaranteed to succeed, and may instead return a NULL pointer. Using the returned value, without checking if the allocation is successful, invokes undefined behavior. This usually leads to a crash, but there is no guarantee that a crash will happen so relying on that can also lead to problems.

For example, unsafe way:

```
struct SomeStruct *s = malloc(sizeof *s);
s->someValue = 0; /* UNSAFE, because s might be a null pointer */
```

Safe way:

```
struct SomeStruct *s = malloc(sizeof *s);
if (s)
{
    s->someValue = 0; /* This is safe, we have checked that s is valid */
}
```

Using literal numbers instead of `sizeof` when requesting memory

For a given compiler/machine configuration, types have a known size; however, there isn't any standard which defines that the size of a given type (other than `char`) will be the same for all compiler/machine configurations. If the code uses 4 instead of `sizeof(int)` for memory allocation, it may work on the original machine, but the code isn't necessarily portable to other machines or compilers. Fixed sizes for types should be replaced by

`sizeof(that_type)` 或 `sizeof(*var_ptr_to_that_type)`。

不可移植的分配：

```
int *intPtr = malloc(4*1000); /* 为1000个int分配存储空间 */
long *longPtr = malloc(8*1000); /* 为1000个long分配存储空间 */
```

可移植的分配：

```
int *intPtr = malloc(sizeof(int)*1000); /* 为1000个int分配存储空间 */
long *longPtr = malloc(sizeof(long)*1000); /* 为1000个long分配存储空间 */
```

或者，更好的方式：

```
int *intPtr = malloc(sizeof(*intPtr)*1000); /* 为1000个int分配存储空间 */
long *longPtr = malloc(sizeof(*longPtr)*1000); /* 为1000个long分配存储空间 */
```

内存泄漏

未使用 `free` 释放内存会导致不可重用内存的积累，这部分内存不再被程序使用；这称为内存泄漏。内存泄漏会浪费内存资源，并可能导致分配失败。

逻辑错误

所有分配必须遵循相同的模式：

1. 使用 `malloc`（或 `calloc`）进行分配
2. 用于存储数据
3. 使用 `free` 进行释放

未遵守此模式，例如在调用 `free` 后使用内存（悬空指针）或在调用 `malloc` 之前使用内存（野指针），调用 `free` 两次（“双重释放”）等，通常会导致段错误并使程序崩溃。

这些错误可能是暂时性的且难以调试—例如，释放的内存通常不会被操作系统立即回收，因此悬空指针可能会持续一段时间并且看似正常工作。

在支持的系统上，`Valgrind` 是识别内存泄漏及其最初分配位置的宝贵工具。

创建指向栈变量的指针

创建指针并不会延长被指向变量的生命周期。例如：

```
int* myFunction()
{
    int x = 10;
    return &x;
}
```

这里，`x` 具有 自动存储期（通常称为 栈 分配）。因为它是在栈上分配的，所以它的生命周期仅限于 `myFunction` 执行期间；当 `myFunction` 退出后，变量 `x` 被销毁。该函数获取 `x` 的地址（使用 `&x`），并将其返回给调用者，导致调用者得到一个指向不存在变量的指针。尝试访问该变量将引发未定义行为。

`sizeof(that_type)` 或 `sizeof(*var_ptr_to_that_type)`。

Non-portable allocation:

```
int *intPtr = malloc(4*1000); /* allocating storage for 1000 int */
long *longPtr = malloc(8*1000); /* allocating storage for 1000 long */
```

Portable allocation:

```
int *intPtr = malloc(sizeof(int)*1000); /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(long)*1000); /* allocating storage for 1000 long */
```

Or, better still:

```
int *intPtr = malloc(sizeof(*intPtr)*1000); /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(*longPtr)*1000); /* allocating storage for 1000 long */
```

Memory leaks

Failure to de-allocate memory using `free` leads to a buildup of non-reusable memory, which is no longer used by the program; this is called a [memory leak](#). Memory leaks waste memory resources and can lead to allocation failures.

Logical errors

All allocations must follow the same pattern:

1. Allocation using `malloc` (or `calloc`)
2. Usage to store data
3. De-allocation using `free`

Failure to adhere to this pattern, such as using memory after a call to `free` ([dangling pointer](#)) or before a call to `malloc` ([wild pointer](#)), calling `free` twice ("double free"), etc., usually causes a segmentation fault and results in a crash of the program.

These errors can be transient and hard to debug – for example, freed memory is usually not immediately reclaimed by the OS, and thus dangling pointers may persist for a while and appear to work.

On systems where it works, `Valgrind` is an invaluable tool for identifying what memory is leaked and where it was originally allocated.

Creating pointers to stack variables

Creating a pointer does not extend the life of the variable being pointed to. For example:

```
int* myFunction()
{
    int x = 10;
    return &x;
}
```

Here, `x` has *automatic storage duration* (commonly known as *stack allocation*). Because it is allocated on the stack, its lifetime is only as long as `myFunction` is executing; after `myFunction` has exited, the variable `x` is destroyed. This function gets the address of `x` (using `&x`), and returns it to the caller, leaving the caller with a pointer to a non-existent variable. Attempting to access this variable will then invoke undefined behavior.

大多数编译器实际上不会在函数退出后清除栈帧，因此对返回的指针进行解引用通常会得到预期的数据。然而，当调用另一个函数时，所指向的内存可能会被覆盖，导致指针所指向的数据看起来已被破坏。

为了解决这个问题，要么使用 `malloc` 为要返回的变量分配存储空间，并返回指向新分配存储的指针，要么要求传入一个有效指针给函数，而不是返回一个指针，例如：

```
#include <stdlib.h>
#include <stdio.h>

int *solution1(void)
{
    int *x = malloc(sizeof *x);
    if (x == NULL)
    {
        /* 出现错误 */
        return NULL;
    }

    *x = 10;

    return x;
}

void solution2(int **x)
{
    /* 注意：使用无效或空指针调用此函数
     * 会导致未定义行为。 */
    *x = 10;
}

int main(void)
{
    /* 使用 solution1() */

    int *foo = solution1();
    if (foo == NULL)
    {
        /* 出现错误 */
        return 1;
    }

    printf("The value set by solution1() is %i", *foo);
    /* 输出结果："The value set by solution1() is 10" */

    free(foo);    /* 清理 */
}

{
    /* 使用 solution2() */

    int bar;
    solution2(&bar);

    printf("solution2() 设置的值是 %i", bar);
    /* 将输出："solution2() 设置的值是 10" */
}
return 0;
```

Most compilers don't actually clear a stack frame after the function exits, thus dereferencing the returned pointer often gives you the expected data. When another function is called however, the memory being pointed to may be overwritten, and it appears that the data being pointed to has been corrupted.

To resolve this, either `malloc` the storage for the variable to be returned, and return a pointer to the newly created storage, or require that a valid pointer is passed in to the function instead of returning one, for example:

```
#include <stdlib.h>
#include <stdio.h>

int *solution1(void)
{
    int *x = malloc(sizeof *x);
    if (x == NULL)
    {
        /* Something went wrong */
        return NULL;
    }

    *x = 10;

    return x;
}

void solution2(int **x)
{
    /* NB: calling this function with an invalid or null pointer
     * causes undefined behaviour. */

    *x = 10;
}

int main(void)
{
    /* Use solution1() */

    int *foo = solution1();
    if (foo == NULL)
    {
        /* Something went wrong */
        return 1;
    }

    printf("The value set by solution1() is %i\n", *foo);
    /* Will output: "The value set by solution1() is 10" */

    free(foo);    /* Tidy up */
}

{
    /* Use solution2() */

    int bar;
    solution2(&bar);

    printf("The value set by solution2() is %i\n", bar);
    /* Will output: "The value set by solution2() is 10" */
}

return 0;
```

}

递增 / 递减和解引用

如果你写 `*p++` 来递增 `p` 指向的内容，那你是错误的。

后置递增 / 递减是在解引用之前执行的。因此，这个表达式会递增指针 `p` 本身，并返回递增前 `p` 指向的内容，且不改变该内容。

你应该写 `(*p)++` 来递增 `p` 指向的内容。

这个规则同样适用于后置递减：`*p--` 会递减指针 `p` 本身，而不是 `p` 指向的内容。

第22.3节：解引用指针

```
int a = 1;
int *a_pointer = &a;
```

要解引用 `a_pointer` 并改变 `a` 的值，我们使用以下操作

```
*a_pointer = 2;
```

这可以通过以下打印语句进行验证。

```
printf("%d", a); /* 打印 2 */
printf("%d", *a_pointer); /* 也打印 2 */
```

然而，解引用一个NULL或其他无效指针是错误的。

```
int *p1, *p2;

p1 = (int *) 0xbad;
p2 = NULL;

*p1 = 42;
*p2 = *p1 + 1;
```

这通常是未定义行为。`p1` 不能被解引用，因为它指向地址 `0xbad`，该地址可能不是有效地址。谁知道那里是什么？可能是操作系统内存，或者是另一个程序的内存。

这类代码唯一的使用场景是在嵌入式开发中，特定信息存储在硬编码地址。`p2` 不能被解引用，因为它是 `NULL`，属于无效指针。

第22.4节：解引用指向结构体的指针

假设我们有以下结构体：

```
struct MY_STRUCT
{
    int my_int;
    float my_float;
};
```

我们可以定义`MY_STRUCT`来省略`struct`关键字，这样每次使用时就不必输入`struct MY_STRUCT`了。不过，这只是可选的。

}

Incrementing / decrementing and dereferencing

If you write `*p++` to increment what is pointed by `p`, you are wrong.

Post incrementing / decrementing is executed before dereferencing. Therefore, this expression will increment the pointer `p` itself and return what was pointed by `p` before incrementing without changing it.

You should write `(*p)++` to increment what is pointed by `p`.

This rule also applies to post decrementing: `*p--` will decrement the pointer `p` itself, not what is pointed by `p`.

Section 22.3: Dereferencing a Pointer

```
int a = 1;
int *a_pointer = &a;
```

To dereference `a_pointer` and change the value of `a`, we use the following operation

```
*a_pointer = 2;
```

This can be verified using the following print statements.

```
printf("%d\n", a); /* Prints 2 */
printf("%d\n", *a_pointer); /* Also prints 2 */
```

However, one would be mistaken to dereference a NULL or otherwise invalid pointer. This

```
int *p1, *p2;

p1 = (int *) 0xbad;
p2 = NULL;

*p1 = 42;
*p2 = *p1 + 1;
```

is usually undefined behavior. `p1` may not be dereferenced because it points to an address `0xbad` which may not be a valid address. Who knows what's there? It might be operating system memory, or another program's memory. The only time code like this is used, is in embedded development, which stores particular information at hard-coded addresses. `p2` cannot be dereferenced because it is `NULL`, which is invalid.

Section 22.4: Dereferencing a Pointer to a struct

Let's say we have the following structure:

```
struct MY_STRUCT
{
    int my_int;
    float my_float;
};
```

We can define `MY_STRUCT` to omit the `struct` keyword so we don't have to type `struct MY_STRUCT` each time we use it. This, however, is optional.

```
typedef struct MY_STRUCT MY_STRUCT;
```

如果我们有一个指向该结构体实例的指针

```
MY_STRUCT *instance;
```

如果该语句出现在文件作用域，程序启动时`instance`将被初始化为一个空指针。如果该语句出现在函数内部，其值未定义。该变量必须初始化为指向一个有效的`MY_STRUCT`变量，或动态分配的空间，才能被解引用。例如：

```
MY_STRUCT info = { 1, 3.141593F };
MY_STRUCT *instance = &info;
```

当指针有效时，我们可以解引用它，通过两种不同的表示法访问其成员：

```
int a = (*instance).my_int;
float b = instance->my_float;
```

虽然这两种方法都可行，但更好的做法是使用箭头`->`操作符，而不是括号、解引用`*`操作符和点`.`操作符的组合，因为箭头操作符更易于阅读和理解，尤其是在嵌套使用时。

另一个重要的区别如下所示：

```
MY_STRUCT 复制 = *实例;
复制.my_int      = 2;
```

在这种情况下，复制包含了实例内容的副本。修改复制的`my_int`不会改变实例中的值。

```
MY_STRUCT *引用 = 实例;
引用->my_int    = 2;
```

在这种情况下，引用是对实例的引用。通过引用修改`my_int`会改变实例中的值。

通常的做法是在函数中使用指向结构体的指针作为参数，而不是直接使用结构体本身。

如果结构体较大，使用结构体作为函数参数可能会导致栈溢出。使用指向结构体的指针只会占用指针大小的栈空间，但如果函数修改了传入的结构体，可能会产生副作用。

第22.5节：常量指针

单指针

- 指向`int`的指针

指针可以指向不同的整数，并且可以通过指针修改整数的值。此代码示例将指针`p`指向整数`b`，然后将`b`的值改为100。

```
int b;
int* p;
p = &b; /* 正确 */
*p = 100; /* 正确 */
```

```
typedef struct MY_STRUCT MY_STRUCT;
```

If we then have a pointer to an instance of this struct

```
MY_STRUCT *instance;
```

If this statement appears at file scope, `instance` will be initialized with a null pointer when the program starts. If this statement appears inside a function, its value is undefined. The variable must be initialized to point to a valid `MY_STRUCT` variable, or to dynamically allocated space, before it can be dereferenced. For example:

```
MY_STRUCT info = { 1, 3.141593F };
MY_STRUCT *instance = &info;
```

When the pointer is valid, we can dereference it to access its members using one of two different notations:

```
int a = (*instance).my_int;
float b = instance->my_float;
```

While both these methods work, it is better practice to use the arrow `->` operator rather than the combination of parentheses, the dereference `*` operator and the dot `.` operator because it is easier to read and understand, especially with nested uses.

Another important difference is shown below:

```
MY_STRUCT copy = *instance;
copy.my_int     = 2;
```

In this case, `copy` contains a copy of the contents of `instance`. Changing `my_int` of `copy` will not change it in `instance`.

```
MY_STRUCT *ref = instance;
ref->my_int    = 2;
```

In this case, `ref` is a reference to `instance`. Changing `my_int` using the reference will change it in `instance`.

It is common practice to use pointers to structs as parameters in functions, rather than the structs themselves. Using the structs as function parameters could cause the stack to overflow if the struct is large. Using a pointer to a struct only uses enough stack space for the pointer, but can cause side effects if the function changes the struct which is passed into the function.

Section 22.5: Const Pointers

Single Pointers

- Pointer to an `int`

The pointer can point to different integers and the `int`'s can be changed through the pointer. This sample of code assigns `b` to point to `int` `b` then changes `b`'s value to 100.

```
int b;
int* p;
p = &b; /* OK */
*p = 100; /* OK */
```

- 指向常量整数的指针

指针可以指向不同的整数，但不能通过指针修改整数的值。

```
int b;  
const int* p;  
p = &b; /* 正确 */  
*p = 100; /* 编译错误 */
```

- 指向整数的常量指针

指针只能指向一个int，但可以通过指针更改该int的值。

```
int a, b;  
int* const p = &b; /* 作为初始化，允许，无赋值 */  
*p = 100; /* 允许 */  
p = &a; /* 编译错误 */
```

- 指向常量整数的常量指针

该指针只能指向一个int，且不能通过该指针修改该int的值。

```
int a, b;  
const int* const p = &b; /* 作为初始化，允许，无赋值 */  
p = &a; /* 编译错误 */  
*p = 100; /* 编译错误 */
```

指向指针的指针

- 指向int指针的指针

该代码将p1的地址赋值给双重指针p（该指针随后指向int* p1（该指针指向int））。

然后将p1指向int类型的a。然后将a的值改为100。

```
void f1(void)  
{  
    int a, b;  
    int *p1;  
    int **p;  
    p1 = &b; /* OK */  
    p = &p1; /* OK */  
    *p = &a; /* OK */  
    **p = 100; /* OK */  
}
```

- 指向常量int的指针的指针

```
void f2(void)  
{  
    int b;  
    const int *p1;  
    const int **p;
```

- Pointer to a `const int`

The pointer can point to different integers but the `int`'s value can't be changed through the pointer.

```
int b;  
const int* p;  
p = &b; /* OK */  
*p = 100; /* Compiler Error */
```

- `const` pointer to `int`

The pointer can only point to one `int` but the `int`'s value can be changed through the pointer.

```
int a, b;  
int* const p = &b; /* OK as initialisation, no assignment */  
*p = 100; /* OK */  
p = &a; /* Compiler Error */
```

- `const` pointer to `const int`

The pointer can only point to one `int` and the `int` can not be changed through the pointer.

```
int a, b;  
const int* const p = &b; /* OK as initialisation, no assignment */  
p = &a; /* Compiler Error */  
*p = 100; /* Compiler Error */
```

Pointer to Pointer

- Pointer to a pointer to an `int`

This code assigns the address of p1 to the double pointer p (which then points to `int*` p1 (which points to `int`)).

Then changes p1 to point to `int` a. Then changes the value of a to be 100.

```
void f1(void)  
{  
    int a, b;  
    int *p1;  
    int **p;  
    p1 = &b; /* OK */  
    p = &p1; /* OK */  
    *p = &a; /* OK */  
    **p = 100; /* OK */  
}
```

- Pointer to pointer to a `const int`

```
void f2(void)  
{  
    int b;  
    const int *p1;  
    const int **p;
```

```

p = &p1; /* OK */
*p = &b; /* OK */
**p = 100; /* 错误：赋值给只读位置 '**p' */
}

```

- 指向 const 指针的 int

```

void f3(void)
{
    int b;
    int *p1;
    int * const *p;
p = &p1; /* OK */
*p = &b; /* 错误：赋值给只读位置 'p' */
**p = 100; /* 正确 */
}

```

- 指向 const 指针的指针到 int

```

void f4(void)
{
    int b;
    int *p1;
    int ** const p = &p1; /* 作为初始化正确，非赋值 */
p = &p1; /* 错误：赋值给只读变量 'p' */
*p = &b; /* 正确 */
**p = 100; /* 正确 */
}

```

- 指向 const 指针的 const int

```

void f5(void)
{
    int b;
    const int *p1;
    const int * const *p;
p = &p1; /* OK */
*p = &b; /* 错误：对只读位置 'p' 的赋值 */
**p = 100; /* 错误：对只读位置 '**p' 的赋值 */
}

```

- 指向常量整数的常量指针的指针

```

void f6(void)
{
    int b;
    const int *p1;
    const int ** const p = &p1; /* 作为初始化是可以的，不是赋值 */
p = &p1; /* 错误：对只读变量 'p' 的赋值 */
*p = &b; /* OK */
**p = 100; /* 错误：赋值给只读位置 '**p' */
}

```

- 指向常量指针的常量指针，指向整数

```

void f7(void)
{
}

```

```

p = &p1; /* OK */
*p = &b; /* OK */
**p = 100; /* error: assignment of read-only location '**p' */
}

```

- Pointer to const pointer to an int

```

void f3(void)
{
    int b;
    int *p1;
    int * const *p;
p = &p1; /* OK */
*p = &b; /* error: assignment of read-only location 'p' */
**p = 100; /* OK */
}

```

- const pointer to pointer to int

```

void f4(void)
{
    int b;
    int *p1;
    int ** const p = &p1; /* OK as initialisation, not assignment */
p = &p1; /* error: assignment of read-only variable 'p' */
*p = &b; /* OK */
**p = 100; /* OK */
}

```

- Pointer to const pointer to const int

```

void f5(void)
{
    int b;
    const int *p1;
    const int * const *p;
p = &p1; /* OK */
*p = &b; /* error: assignment of read-only location 'p' */
**p = 100; /* error: assignment of read-only location '**p' */
}

```

- const pointer to pointer to const int

```

void f6(void)
{
    int b;
    const int *p1;
    const int ** const p = &p1; /* OK as initialisation, not assignment */
p = &p1; /* error: assignment of read-only variable 'p' */
*p = &b; /* OK */
**p = 100; /* error: assignment of read-only location '**p' */
}

```

- const pointer to const pointer to int

```

void f7(void)
{
}

```

```

int b;
int *p1;
int * const * const p = &p1; /* 作为初始化是可以的，不是赋值 */
p = &p1; /* 错误：对只读变量 'p' 的赋值 */
*p = &b; /* 错误：赋值给只读位置 '*p' */
**p = 100; /* 正确 */
}

```

```

int b;
int *p1;
int * const * const p = &p1; /* OK as initialisation, not assignment */
p = &p1; /* error: assignment of read-only variable 'p' */
*p = &b; /* error: assignment of read-only location '*p' */
**p = 100; /* OK */
}

```

第22.6节：函数指针

指针也可以用来指向函数。

让我们来看一个基本函数：

```

int my_function(int a, int b)
{
    return 2 * a + 3 * b;
}

```

现在，让我们定义一个该函数类型的指针：

```
int (*my_pointer)(int, int);
```

要创建一个，只需使用以下模板：

```
return_type_of_func (*my_func_pointer)(type_arg1, type_arg2, ...)
```

然后我们必须将该指针赋值给函数：

```
my_pointer = &my_function;
```

现在可以使用该指针来调用函数：

```

/* 调用指针指向的函数 */
int result = (*my_pointer)(4, 2);

...

/* 将函数指针作为参数传递给另一个函数 */
void another_function(int (*another_pointer)(int, int))
{
    int a = 4;
    int b = 2;
    int result = (*another_pointer)(a, b);printf("%d
", result);
}

```

虽然这种语法看起来更自然且与基本类型一致，但赋值和解引用函数指针时并不需要使用`&`和`*`操作符。因此，以下代码片段同样有效：

```

/* 赋值时不使用 & 操作符 */
my_pointer = my_function;

/* 不使用 * 运算符进行解引用 */
int result = my_pointer(4, 2);

```

Section 22.6: Function pointers

Pointers can also be used to point at functions.

Let's take a basic function:

```

int my_function(int a, int b)
{
    return 2 * a + 3 * b;
}

```

Now, let's define a pointer of that function's type:

```
int (*my_pointer)(int, int);
```

To create one, just use this template:

```
return_type_of_func (*my_func_pointer)(type_arg1, type_arg2, ...)
```

We then must assign this pointer to the function:

```
my_pointer = &my_function;
```

This pointer can now be used to call the function:

```

/* Calling the pointed function */
int result = (*my_pointer)(4, 2);

...

/* Using the function pointer as an argument to another function */
void another_function(int (*another_pointer)(int, int))
{
    int a = 4;
    int b = 2;
    int result = (*another_pointer)(a, b);

    printf("%d\n", result);
}

```

Although this syntax seems more natural and coherent with basic types, attributing and dereferencing function pointers don't require the usage of `&` and `*` operators. So the following snippet is equally valid:

```

/* Attribution without the & operator */
my_pointer = my_function;

/* Dereferencing without the * operator */
int result = my_pointer(4, 2);

```

为了提高函数指针的可读性，可以使用 `typedef`。

```
typedef void (*Callback)(int a);

void some_function(Callback callback)
{
    int a = 4;
    callback(a);
}
```

另一个提高可读性的技巧是，C 标准允许在参数中简化函数指针（如上所示）（但不允许在变量声明中），使其看起来像函数原型；因此，以下内容可以等效地用于函数定义和声明：

```
void some_function(void callback(int))
{
    int a = 4;
    callback(a);
}
```

另见

函数指针

第 22.7 节：使用 `void` 指针实现多态行为

标准库函数 `qsort()` 是一个很好的例子，展示了如何使用 `void` 指针让单个函数操作多种不同类型的数据。

```
void qsort (
    void *base,           /* 待排序的数组 */
    size_t num,            /* 数组中元素的数量 */
    size_t size,           /* 每个元素的字节大小 */
    int (*compar)(const void *, const void *)); /* 用于比较两个元素的函数 */
```

待排序的数组作为 `void` 指针传入，因此可以操作任何类型的元素数组。接下来的两个参数告诉 `qsort()` 数组中应有多少元素，以及每个元素的字节大小。

最后一个参数是指向比较函数的函数指针，该函数本身接受两个 `void` 指针。通过让调用者提供此函数，`qsort()` 可以有效地对任何类型的元素进行排序。

下面是一个比较浮点数的比较函数示例。请注意，传递给 `qsort()` 的任何比较函数都需要具有此类型签名。它之所以能实现多态，是通过将 `void` 指针参数转换为我们希望比较的元素类型的指针来实现的。

```
int compare_floats(const void *a, const void *b)
{
    float fa = *((float *)a);
    float fb = *((float *)b);
    if (fa < fb)
        return -1;
    if (fa > fb)
        return 1;
    return 0;
}
```

To increase the readability of function pointers, `typedefs` may be used.

```
typedef void (*Callback)(int a);

void some_function(Callback callback)
{
    int a = 4;
    callback(a);
}
```

Another readability trick is that the C standard allows one to simplify a function pointer in arguments like above (but not in variable declaration) to something that looks like a function prototype; thus the following can be equivalently used for function definitions and declarations:

```
void some_function(void callback(int))
{
    int a = 4;
    callback(a);
}
```

See also

Function Pointers

Section 22.7: Polymorphic behaviour with `void` pointers

The `qsort()` standard library function is a good example of how one can use `void` pointers to make a single function operate on a large variety of different types.

```
void qsort (
    void *base,           /* Array to be sorted */
    size_t num,            /* Number of elements in array */
    size_t size,           /* Size in bytes of each element */
    int (*compar)(const void *, const void *)); /* Comparison function for two elements */
```

The array to be sorted is passed as a `void` pointer, so an array of any type of element can be operated on. The next two arguments tell `qsort()` how many elements it should expect in the array, and how large, in bytes, each element is.

The last argument is a function pointer to a comparison function which itself takes two `void` pointers. By making the caller provide this function, `qsort()` can effectively sort elements of any type.

Here's an example of such a comparison function, for comparing floats. Note that any comparison function passed to `qsort()` needs to have this type signature. The way it is made polymorphic is by casting the `void` pointer arguments to pointers of the type of element we wish to compare.

```
int compare_floats(const void *a, const void *b)
{
    float fa = *((float *)a);
    float fb = *((float *)b);
    if (fa < fb)
        return -1;
    if (fa > fb)
        return 1;
    return 0;
}
```

由于我们知道 `qsort` 会使用此函数来比较浮点数，因此在解引用之前，我们将 `void` 指针参数转换回 `float` 指针。

现在，使用多态函数 `qsort` 对一个名为 "array"、长度为 "len" 的数组进行排序非常简单：

```
qsort(array, len, sizeof(array[0]), compare_floats);
```

第22.8节：取地址运算符（&）

对于任何对象（即变量、数组、联合体、结构体、指针或函数），一元取地址运算符可用于访问该对象的地址。

假设

```
int i = 1;
int *p = NULL;
```

那么语句 `p = &i;` 会将变量 `i` 的地址复制给指针 `p`。

这表示 `p` 指向 `i`。

`printf("%d", *p);` 会打印1，即 `i` 的值。

第22.9节：指针初始化

指针初始化是避免野指针的好方法。初始化很简单，与变量的初始化没有区别。

```
#include <stddef.h>

int main()
{
    int *p1 = NULL;
    char *p2 = NULL;
    float *p3 = NULL;

    /* NULL 是在 stddef.h、stdio.h、stdlib.h 和 string.h 中定义的宏 */
    ...
}
```

在大多数操作系统中，无意中使用已初始化为NULL的指针通常会导致程序立即崩溃，从而很容易识别问题的原因。使用未初始化的指针则常常会导致难以诊断的错误。

注意：

解引用NULL指针的结果是未定义的，因此即使这是程序运行的操作系统的自然行为，它不一定会导致崩溃。编译器优化可能会掩盖崩溃，导致崩溃发生在源代码中空指针解引用之前或之后，或者导致包含空指针解引用的代码部分意外地从程序中被移除。调试版本通常不会表现出这些行为，但语言标准并不保证这一点。其他意外和/或不希望出现的行为也是允许的。

因为NULL从不指向变量、已分配的内存或函数，所以用作保护值是安全的。

Since we know that `qsort` will use this function to compare floats, we cast the void pointer arguments back to float pointers before dereferencing them.

Now, the usage of the polymorphic function `qsort` on an array "array" with length "len" is very simple:

```
qsort(array, len, sizeof(array[0]), compare_floats);
```

Section 22.8: Address-of Operator (&)

For any object (i.e, variable, array, union, struct, pointer or function) the unary address operator can be used to access the address of that object.

Suppose that

```
int i = 1;
int *p = NULL;
```

So then a statement `p = &i;`, copies the address of the variable `i` to the pointer `p`.

It's expressed as `p points to i`.

`printf("%d\n", *p);` prints 1, which is the value of `i`.

Section 22.9: Initializing Pointers

Pointer initialization is a good way to avoid wild pointers. The initialization is simple and is no different from initialization of a variable.

```
#include <stddef.h>

int main()
{
    int *p1 = NULL;
    char *p2 = NULL;
    float *p3 = NULL;

    /* NULL is a macro defined in stddef.h, stdio.h, stdlib.h, and string.h */
    ...
}
```

In most operating systems, inadvertently using a pointer that has been initialized to NULL will often result in the program crashing immediately, making it easy to identify the cause of the problem. Using an uninitialized pointer can often cause hard-to-diagnose bugs.

Caution:

The result of dereferencing a NULL pointer is undefined, so it *will not necessarily cause a crash* even if that is the natural behaviour of the operating system the program is running on. Compiler optimizations may mask the crash, cause the crash to occur before or after the point in the source code at which the null pointer dereference occurred, or cause parts of the code that contains the null pointer dereference to be unexpectedly removed from the program. Debug builds will not usually exhibit these behaviours, but this is not guaranteed by the language standard. Other unexpected and/or undesirable behaviour is also allowed.

Because NULL never points to a variable, to allocated memory, or to a function, it is safe to use as a guard value.

注意：

通常NULL被定义为(void *)0。但这并不意味着被赋予的内存地址是0x0。更多说明请参见C-faq 关于 NULL 指针的内容

注意，你也可以将指针初始化为除 NULL 以外的其他值。

```
int i1;

int main()
{
    int *p1 = &i1;
    const char *p2 = "指向常量字符串的指针";
    float *p3 = malloc(10 * sizeof(float));
}
```

第22.10节：指向指针的指针

在C语言中，指针可以指向另一个指针。

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int A = 42;
    int* pA = &A;
    int** ppA = &pA;
    int*** pppA = &ppA;

    printf("%d", ***pppA); /* 输出42 */

    return EXIT_SUCCESS;
}
```

但是，不允许直接引用引用。

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int A = 42;
    int* pA = &A;
    int** ppA = &&A; /* 这里编译错误！ */
    int*** pppA = &&&A; /* 这里编译错误！ */

    ...
}
```

第22.11节：void* 指针作为标准函数的参数和返回值

版本 > K&R

void* 是指向对象类型指针的通用类型。一个使用示例是 malloc 函数，其声明为

```
void* malloc(size_t);
```

Caution:

Usually NULL is defined as (void *)0. But this does not imply that the assigned memory address is 0x0. For more clarification refer to [C-faq for NULL pointers](#)

Note that you can also initialize pointers to contain values other than NULL.

```
int i1;

int main()
{
    int *p1 = &i1;
    const char *p2 = "A constant string to point to";
    float *p3 = malloc(10 * sizeof(float));
}
```

Section 22.10: Pointer to Pointer

In C, a pointer can refer to another pointer.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int A = 42;
    int* pA = &A;
    int** ppA = &pA;
    int*** pppA = &ppA;

    printf("%d", ***pppA); /* prints 42 */

    return EXIT_SUCCESS;
}
```

But, reference-and-reference directly is not allowed.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int A = 42;
    int* pA = &A;
    int** ppA = &&A; /* Compilation error here! */
    int*** pppA = &&&A; /* Compilation error here! */

    ...
}
```

Section 22.11: void* pointers as arguments and return values to standard functions

Version > K&R

void* is a catch all type for pointers to object types. An example of this in use is with the malloc function, which is declared as

```
void* malloc(size_t);
```

指向 void 的指针返回类型意味着可以将 malloc 的返回值赋给指向任何其他类型对象的指针：

```
int* vector = malloc(10 * sizeof *vector);
```

通常认为不应显式地将值转换为 void 指针或从 void 指针转换出来是一种良好做法。在 malloc() 的特定情况下，这是因为如果显式转换，编译器可能会假设但不会警告 malloc() 返回类型错误，尤其是当你忘记包含 stdlib.h 时。这也是利用 void 指针的正确行为更好地遵守 DRY（不要重复自己）原则的一个例子；与下面代码相比，后者包含多个不必要的额外位置，可能因打字错误而导致问题：

```
int* vector = (int*)malloc(10 * sizeof int*);
```

类似地，诸如

```
void* memcpy(void *restrict target, void const *restrict source, size_t size);
```

的函数参数被指定为 void * 是因为可以传入任何对象的地址，无论其类型如何。在这里，调用时也不应使用强制转换

```
unsigned char buffer[sizeof(int)];  
int b = 67;  
memcpy(buffer, &b, sizeof buffer);
```

第22.12节：相同的星号，不同的含义

前提

围绕C和C++中指针语法最令人困惑的事情是，当指针符号星号 (*) 与变量一起使用时，实际上有两种不同的含义。

示例

首先，你使用*来声明一个指针变量。

```
int i = 5;  
/* 'p' 是一个指向整数的指针，初始化为NULL */  
int *p = NULL;  
/* '&i' 计算为 'i' 的地址，然后赋值给 'p' */  
p = &i;  
/* 'p' 现在保存着 'i' 的地址 */
```

当你不是在声明（或乘法）时，* 用于解引用一个指针变量：

```
*p = 123;  
/* 'p' 指向 'i'，所以这会将 'i' 的值改为 123 */
```

当你想让一个已有的指针变量保存另一个变量的地址时，不要使用*，而是这样做：

```
p = &another_variable;
```

C 语言初学者常常在同时声明和初始化指针变量时产生混淆。

The pointer-to-void return type means that it is possible to assign the return value from malloc to a pointer to any other type of object:

```
int* vector = malloc(10 * sizeof *vector);
```

It is generally considered good practice to *not* explicitly cast the values into and out of void pointers. In specific case of malloc() this is because with an explicit cast, the compiler may otherwise assume, but not warn about, an incorrect return type for malloc(), if you forget to include stdlib.h. It is also a case of using the correct behavior of void pointers to better conform to the DRY (don't repeat yourself) principle; compare the above to the following, wherein the following code contains several needless additional places where a typo could cause issues:

```
int* vector = (int*)malloc(10 * sizeof int*);
```

Similarly, functions such as

```
void* memcpy(void *restrict target, void const *restrict source, size_t size);
```

have their arguments specified as void * because the address of any object, regardless of the type, can be passed in. Here also, a call should not use a cast

```
unsigned char buffer[sizeof(int)];  
int b = 67;  
memcpy(buffer, &b, sizeof buffer);
```

Section 22.12: Same Asterisk, Different Meanings

Premise

The most confusing thing surrounding pointer syntax in C and C++ is that there are actually two different meanings that apply when the pointer symbol, the asterisk (*), is used with a variable.

Example

Firstly, you use * to **declare** a pointer variable.

```
int i = 5;  
/* 'p' is a pointer to an integer, initialized as NULL */  
int *p = NULL;  
/* '&i' evaluates into address of 'i', which then assigned to 'p' */  
p = &i;  
/* 'p' is now holding the address of 'i' */
```

When you're not declaring (or multiplying), * is used to **dereference** a pointer variable:

```
*p = 123;  
/* 'p' was pointing to 'i', so this changes value of 'i' to 123 */
```

When you want an existing pointer variable to hold address of other variable, you **don't** use *, but do it like this:

```
p = &another_variable;
```

A common confusion among C-programming newbies arises when they declare and initialize a pointer variable at the same time.

```
int *p = &i;
```

由于 `int i = 5;` 和 `int i; i = 5;` 结果相同，有些人可能会认为 `int *p = &i;` 和 `int *p; *p = &i;` 也会得到相同结果。事实并非如此，`int *p; *p = &i;` 会尝试解引用一个未初始化的指针，这将导致未定义行为。不要在非声明或非解引用指针时使用*。

结论

星号 (*) 在C语言中与指针相关时有两种不同的含义，取决于它的使用位置。当用于变量声明中时，等号右侧的值应为指向内存中某个地址的指针值。当用于已经声明的变量时，星号将解引用该指针值，跟随它到内存中指向的位置，从而允许对存储在那里值的赋值或读取。

要点

在处理指针时，所谓“注意你的P和Q”非常重要。要留意你何时使用星号，以及在那个位置使用时它的含义。忽视这个细节可能导致程序出现错误和/或未定义行为，而这些是你绝对不想面对的。

```
int *p = &i;
```

Since `int i = 5;` and `int i; i = 5;` give the same result, some of them might thought `int *p = &i;` and `int *p; *p = &i;` give the same result too. The fact is, no, `int *p; *p = &i;` will attempt to deference an **uninitialized** pointer which will result in UB. Never use * when you're not declaring nor dereferencing a pointer.

Conclusion

The asterisk (*) has two distinct meanings within C in relation to pointers, depending on where it's used. When used within a **variable declaration**, the value on the right hand side of the equals side should be a **pointer value** to an **address** in memory. When used with an already **declared variable**, the asterisk will **dereference** the pointer value, following it to the pointed-to place in memory, and allowing the value stored there to be assigned or retrieved.

Takeaway

It is important to mind your P's and Q's, so to speak, when dealing with pointers. Be mindful of when you're using the asterisk, and what it means when you use it there. Overlooking this tiny detail could result in buggy and/or undefined behavior that you really don't want to have to deal with.

第23章：序列点

第23.1节：无序表达式

版本 ≥ C11

以下表达式是无序的：

```
a + b;  
a - b;  
a * b;  
a / b;  
a % b;  
a & b;  
a | b;
```

在上述示例中，表达式a可能在表达式b之前或之后被求值，b可能在 a之前被求值，或者如果它们对应多条指令，求值顺序甚至可能交错进行。

函数调用也遵循类似规则：

```
f(a, b);
```

这里不仅 a 和 b 是无序的（即函数调用中的操作符不产生序列点），而且 f，即决定调用哪个函数的表达式也是无序的。

副作用可能在求值后立即应用，也可能延迟到稍后的某个时间点。

类似于

```
x++ & x++;  
f(x++, x++); /* 函数调用中的 ',' *不是*与逗号操作符相同 */  
x++ * x++;  
a[i] = i++;
```

或

```
x++ & x;  
f(x++, x);  
x++ * x;  
a[i++] = i;
```

将导致未定义行为，因为

- 对一个对象的修改和对该对象的任何其他访问必须是有序的
- 求值顺序和副作用¹的应用顺序未被指定。

1 执行环境状态的任何变化。

第23.2节：序列表达式

以下表达式是有序的：

```
a && b  
a || b
```

Chapter 23: Sequence points

Section 23.1: Unsequenced expressions

Version ≥ C11

The following expressions are *unsequenced*:

```
a + b;  
a - b;  
a * b;  
a / b;  
a % b;  
a & b;  
a | b;
```

In the above examples, the expression a may be evaluated before or after the expression b, b may be evaluated before a, or they may even be intermixed if they correspond to several instructions.

A similar rule holds for function calls:

```
f(a, b);
```

Here not only a and b are unsequenced (i.e. the , operator in a function call does not produce a sequence point) but also f, the expression that determines the function that is to be called.

Side effects may be applied immediately after evaluation or deferred until a later point.

Expressions like

```
x++ & x++;  
f(x++, x++); /* the ',' in a function call is *not* the same as the comma operator */  
x++ * x++;  
a[i] = i++;
```

or

```
x++ & x;  
f(x++, x);  
x++ * x;  
a[i++] = i;
```

will yield *undefined behavior* because

- a modification of an object and any other access to it must be sequenced
- the order of evaluation and the order in which *side effects*¹ are applied is not specified.

1 Any changes in the state of the execution environment.

Section 23.2: Sequenced expressions

The following expressions are *sequenced*:

```
a && b  
a || b
```

```
a , b  
a ? b : c  
for ( a ; b ; c ) { ... }
```

在所有情况下，表达式 a 会被完全求值，且所有副作用都会被应用，然后才会对 b 或 c 进行求值。在第四种情况下，只有 b 或 c 中的一个会被求值。在最后一种情况下，b 会被完全求值，且所有副作用都会被应用，然后才对 c 进行求值。

在所有情况下，表达式 a 的求值先于 b 或 c 的求值（换言之，b 和 c 的求值后于 a 的求值）。

因此，像下面这样的表达式

```
x++ && x++  
x++ ? x++ : y++  
(x = f()) && x != 0  
for ( x = 0; x < 10; x++ ) { ... }  
y = (x++, x++);
```

具有良好定义的行为。

第23.3节：不确定顺序的表达式

函数调用如 f(a) 总是意味着在参数和指定符（此处为 f 和 a）的求值与实际调用之间存在一个序列点。如果两个这样的调用是无序的，则这两个函数调用是不确定顺序的，也就是说，一个在另一个之前执行，但顺序未指定。

```
unsigned counter = 0;  
  
unsigned account(void) {  
    return counter++;  
}  
  
int main(void) {  
    printf("the order is %u %u", account(), account());
```

在评估 printf 参数时对 counter 的这种隐式双重修改是有效的，我们只是不知道哪个调用先发生。由于顺序未指定，可能会变化且不能依赖。因此输出可能是：

顺序是 0 1

或

顺序是 1 0

与上述类似的语句，但没有中间函数调用

```
printf("顺序是 %u %u", counter++, counter++); // 未定义行为
```

由于对counter的两次修改之间没有序列点，存在未定义行为。

```
a , b  
a ? b : c  
for ( a ; b ; c ) { ... }
```

In all cases, the expression a is fully evaluated and *all side effects are applied* before either b or c are evaluated. In the fourth case, only one of b or c will be evaluated. In the last case, b is fully evaluated and all side effects are applied before c is evaluated.

In all cases, the evaluation of expression a is *sequenced before* the evaluations of b or c (alternately, the evaluations of b and c are *sequenced after* the evaluation of a).

Thus, expressions like

```
x++ && x++  
x++ ? x++ : y++  
(x = f()) && x != 0  
for ( x = 0; x < 10; x++ ) { ... }  
y = (x++, x++);
```

have well defined behavior.

Section 23.3: Indeterminately sequenced expressions

Function calls as f(a) always imply a sequence point between the evaluation of the arguments and the designator (here f and a) and the actual call. If two such calls are unsequenced, the two function calls are indeterminately sequenced, that is, one is executed before the other, and order is unspecified.

```
unsigned counter = 0;  
  
unsigned account(void) {  
    return counter++;  
}  
  
int main(void) {  
    printf("the order is %u %u\n", account(), account());
```

This implicit twofold modification of counter during the evaluation of the printf arguments is valid, we just don't know which of the calls comes first. As the order is unspecified, it may vary and cannot be depended on. So the printout could be:

the order is 0 1

or

the order is 1 0

The analogous statement to the above without intermediate function call

```
printf("the order is %u %u\n", counter++, counter++); // undefined behavior
```

has undefined behavior because there is no sequence point between the two modifications of counter.

第24章：函数指针

函数指针是指向函数而非数据类型的指针。它们可以用于在运行时允许调用的函数具有可变性。

第24.1节：介绍

就像char和int一样，函数是C语言的基本特性。因此，你可以声明一个指向函数的指针：这意味着你可以将调用哪个函数传递给另一个函数以帮助其完成任务。例如，如果你有一个显示图形的graph()函数，你可以将用于绘图的函数传递给graph()。

```
// 为了使示例更清晰，定义几个外部变量
extern unsigned int screenWidth;
extern void plotXY(double x, double y);

// graph()函数。
// 传入边界：应绘制的最小和最大X及Y值。
// 还要传入实际绘图的函数。
void graph(double minX, double minY,
           double maxX, double maxY,
           ??? *fn) {           // 语法见下文

    double stepX = (maxX - minX) / screenWidth;
    for (double x=minX; x<maxX; x+=stepX) {

        double y = fn(x);           // 通过调用传入的 fn() 获取该 x 对应的 y 值

        if (minY<=y && y<maxY) {
            plotXY(x, y);          // 绘制计算出的点
        } // if
    } // for
} // graph(minX, minY, maxX, maxY, fn)
```

用法

所以上述代码将绘制你传入的任意函数的图形——只要该函数满足某些条件：

即你传入一个double类型的参数，并返回一个double类型的结果。有许多函数符合这个条件——如 sin()、cos()、tan()、exp() 等——但也有许多不符合，比如 graph() 本身！

语法

那么你如何指定可以传入 graph() 的函数，以及哪些不能传入呢？传统的方法是使用一种可能不易阅读或理解的语法：

```
double (*fn)(double); // fn 是一个指向函数的指针，该函数接受一个 double 类型参数并返回一个 double 类型值
```

上述问题在于同时试图定义两件事：函数的结构，以及它是一个指针的事实。所以，将这两个定义分开！但通过使用type def，可以实现更好的语法（更易读且易于理解）。

第24.2节：从函数返回函数指针

```
#include <stdio.h>

enum 操作符
{
```

Chapter 24: Function Pointers

Function pointers are pointers that point to functions instead of data types. They can be used to allow variability in the function that is to be called, at run-time.

Section 24.1: Introduction

Just like `char` and `int`, a function is a fundamental feature of C. As such, you can declare a pointer to one: which means that you can pass *which function to call* to another function to help it do its job. For example, if you had a `graph()` function that displayed a graph, you could pass *which function to graph* into `graph()`.

```
// A couple of external definitions to make the example clearer
extern unsigned int screenWidth;
extern void plotXY(double x, double y);

// The graph() function.
// Pass in the bounds: the minimum and maximum X and Y that should be plotted.
// Also pass in the actual function to plot.
void graph(double minX, double minY,
           double maxX, double maxY,
           ??? *fn) {           // See below for syntax

    double stepX = (maxX - minX) / screenWidth;
    for (double x=minX; x<maxX; x+=stepX) {

        double y = fn(x);           // Get y for this x by calling passed-in fn()

        if (minY<=y && y<maxY) {
            plotXY(x, y);          // Plot calculated point
        } // if
    } // for
} // graph(minX, minY, maxX, maxY, fn)
```

Usage

So the above code will graph whatever function you passed into it - as long as that function meets certain criteria: namely, that you pass a `double` in and get a `double` out. There are many functions like that - `sin()`, `cos()`, `tan()`, `exp()` etc. - but there are many that aren't, such as `graph()` itself!

Syntax

So how do you specify which functions you can pass into `graph()` and which ones you can't? The conventional way is by using a syntax that may not be easy to read or understand:

```
double (*fn)(double); // fn is a pointer-to-function that takes a double and returns one
```

The problem above is that there are two things trying to be defined at the same time: the structure of the function, and the fact that it's a pointer. So, split the two definitions! But by using `typedef`, a better syntax (easier to read & understand) can be achieved.

Section 24.2: Returning Function Pointers from a Function

```
#include <stdio.h>

enum 操作符
{
```

```

加法 = '+',
减法 = '-',
};

/* add : 将a和b相加, 返回结果 */
int add(int a, int b)
{
    return a + b;
}

/* sub : 用a减去b, 返回结果 */
int sub(int a, int b)
{
    return a - b;
}

/* getmath : 返回相应的数学函数 */
int (*getmath(enum Op op))(int,int)
{
    switch (op)
    {
        case ADD:
            return &add;
        case SUB:
            return &sub;
        default:
            return NULL;
    }
}

int main(void)
{
    int a, b, c;
    int (*fp)(int,int);

    fp = getmath(ADD);

    a = 1, b = 2;
    c = (*fp)(a, b);printf("%d
+ %d = %d", a, b, c);return 0;
}

```

```

ADD = '+',
SUB = '-';
};

/* add: add a and b, return result */
int add(int a, int b)
{
    return a + b;
}

/* sub: subtract b from a, return result */
int sub(int a, int b)
{
    return a - b;
}

/* getmath: return the appropriate math function */
int (*getmath(enum Op op))(int,int)
{
    switch (op)
    {
        case ADD:
            return &add;
        case SUB:
            return &sub;
        default:
            return NULL;
    }
}

int main(void)
{
    int a, b, c;
    int (*fp)(int,int);

    fp = getmath(ADD);

    a = 1, b = 2;
    c = (*fp)(a, b);
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}

```

第24.3节：最佳实践

使用typedef

使用typedef来代替每次手动声明函数指针可能会很方便。

声明函数指针的typedef的语法是：

```
typedef 返回类型 (*名称)(参数);
```

示例：

假设我们有一个函数 sort，期望传入一个指向函数 compare的函数指针，该函数满足以下条件：

compare - 一个用于比较两个元素的函数，将传递给排序函数。

Section 24.3: Best Practices

Using typedef

It might be handy to use a **typedef** instead of declaring the function pointer each time by hand.

The syntax for declaring a **typedef** for a function pointer is:

```
typedef returnType (*name)(parameters);
```

Example:

Posit that we have a function, sort, that expects a function pointer to a function compare such that:

compare - A compare function for two elements which is to be supplied to a sort function.

"compare"函数预期在两个元素被认为相等时返回0，如果第一个传入的元素在某种意义上“较大”，则返回正值，否则返回负值（表示第一个元素“较小”）。

如果没有typedef，我们将以如下方式将函数指针作为参数传递给函数：

```
void sort(int (*compare)(const void *elem1, const void *elem2)) {  
    /* 在此代码块内，函数名为 "compare" */  
}
```

使用typedef，我们可以这样写：

```
typedef int (*compare_func)(const void *, const void *);
```

然后我们可以将 sort的函数签名改为：

```
void sort(compare_func func) {  
    /* 在此代码块中，函数名为 "func" */  
}
```

这两种 sort的定义都可以接受任何形式的函数

```
int compare(const void *arg1, const void *arg2) {  
    /* 注意变量名不必是 "elem1" 和 "elem2" */  
}
```

函数指针是唯一需要包含类型指针属性的地方，例如不要尝试定义类似typedef struct something_struct *something_type的类型。即使是结构体中包含不应由API调用者直接访问的成员，也适用此规则，例如stdio.h中的FILE类型（你现在会注意到它不是指针）。

获取上下文指针。

函数指针几乎总是应该接受用户提供的 void * 作为上下文指针。

示例

```
/* 函数最小化器，细节不重要 */  
double findminimum( double (*fptr)(double x, double y, void *ctx), void *ctx)  
{  
    ...  
  
    /* 反复调用类似这样的函数 */  
    temp = (*fptr)(testx, testy, ctx);  
}  
  
/* 我们要最小化的函数，两个三次多项式之和 */  
double *cubics(double x, double y, void *ctx)  
{  
    double *coeffsx = ctx;  
    double *coeffsy = coeffx + 4;  
  
    return coeffsx[0] * x * x * x + coeffsx[1] * x * x + coeffsx[2] * x + coeffsx[3] +  
           coeffsy[0] * y * y * y + coeffsy[1] * y * y + coeffsy[2] * y + coeffsy[3];  
}  
  
void caller()
```

"compare" is expected to return 0 if the two elements are deemed equal, a positive value if the first element passed is "larger" in some sense than the latter element and otherwise the function returns a negative value (meaning that the first element is "lesser" than the latter).

Without a **typedef** we would pass a function pointer as an argument to a function in the following manner:

```
void sort(int (*compare)(const void *elem1, const void *elem2)) {  
    /* inside of this block, the function is named "compare" */  
}
```

With a **typedef**, we'd write:

```
typedef int (*compare_func)(const void *, const void *);
```

and then we could change the function signature of sort to:

```
void sort(compare_func func) {  
    /* In this block the function is named "func" */  
}
```

both definitions of sort would accept any function of the form

```
int compare(const void *arg1, const void *arg2) {  
    /* Note that the variable names do not have to be "elem1" and "elem2" */  
}
```

Function pointers are the only place where you should include the pointer property of the type, e.g. do not try to define types like **typedef struct** something_struct *something_type. This applies even for a structure with members which are not supposed to accessed directly by API callers, for example the stdio.h FILE type (which as you now will notice is not a pointer).

Taking context pointers.

A function pointer should almost always take a user-supplied void * as a context pointer.

Example

```
/* function minimiser, details unimportant */  
double findminimum( double (*fptr)(double x, double y, void *ctx), void *ctx)  
{  
    ...  
  
    /* repeatedly make calls like this */  
    temp = (*fptr)(testx, testy, ctx);  
}  
  
/* the function we are minimising, sums two cubics */  
double *cubics(double x, double y, void *ctx)  
{  
    double *coeffsx = ctx;  
    double *coeffsy = coeffx + 4;  
  
    return coeffsx[0] * x * x * x + coeffsx[1] * x * x + coeffsx[2] * x + coeffsx[3] +  
           coeffsy[0] * y * y * y + coeffsy[1] * y * y + coeffsy[2] * y + coeffsy[3];  
}  
  
void caller()
```

```

{
    /* 上下文，三次方程的系数 */
    double coeffs[8] = {1, 2, 3, 4, 5, 6, 7, 8};
    double min;

    min = findminimum(cubics, coeffs);
}

```

使用上下文指针意味着额外的参数不需要硬编码到被指向的函数中，也不需要使用全局变量。

库函数 `qsort()` 不遵循此规则，对于简单的比较函数通常可以不使用上下文指针。但对于更复杂的情况，上下文指针变得必不可少。

另见

函数指针

第24.4节：赋值函数指针

```

#include <stdio.h>

/* increment：接收一个数字，将其加一后返回 */
int increment(int i)
{
    printf("increment %d by 1", i); return i +
    1;
}

/* decrement：取数字，将其减一，并返回 */
int decrement(int i)
{
    printf("decrement %d by 1", i); return i -
    1;
}

int main(void)
{
    int num = 0; /* 声明用于递增的数字 */
    int (*fp)(int); /* 声明一个函数指针 */

    fp = &increment; /* 将函数指针指向递增函数 */
    num = (*fp)(num); /* 递增 num */
    num = (*fp)(num); /* 第二次递增 num */

    fp = &decrement; /* 将函数指针指向递减函数 */
    num = (*fp)(num); /* 递减 num */
    printf("num is now: %d", num); return
    0;
}

```

第24.5节：编写函数指针的助记符

所有C函数实际上都是指向程序内存中某段代码位置的指针。函数指针的主要用途是为其他函数提供“回调”（或模拟类和对象）。

函数的语法定义如下（在本页稍后部分有详细说明）：

```

{
    /* context, the coefficients of the cubics */
    double coeffs[8] = {1, 2, 3, 4, 5, 6, 7, 8};
    double min;

    min = findminimum(cubics, coeffs);
}

```

Using the context pointer means that the extra parameters do not need to be hard-coded into the function pointed to, or require the use of globals.

The library function `qsort()` does not follow this rule, and one can often get away without context for trivial comparison functions. But for anything more complicated, the context pointer becomes essential.

See also

Functions pointers

Section 24.4: Assigning a Function Pointer

```

#include <stdio.h>

/* increment: take number, increment it by one, and return it */
int increment(int i)
{
    printf("increment %d by 1\n", i);
    return i + 1;
}

/* decrement: take number, decrement it by one, and return it */
int decrement(int i)
{
    printf("decrement %d by 1\n", i);
    return i - 1;
}

int main(void)
{
    int num = 0; /* declare number to increment */
    int (*fp)(int); /* declare a function pointer */

    fp = &increment; /* set function pointer to increment function */
    num = (*fp)(num); /* increment num */
    num = (*fp)(num); /* increment num a second time */

    fp = &decrement; /* set function pointer to decrement function */
    num = (*fp)(num); /* decrement num */
    printf("num is now: %d\n", num);
    return 0;
}

```

Section 24.5: Mnemonic for writing function pointers

All C functions are actually pointers to a spot in the program memory where some code exists. The main use of a function pointer is to provide a "callback" to other functions (or to simulate classes and objects).

The syntax of a function, as defined further down on this page is:

```
returnType (*name)(parameters)
```

编写函数指针定义的助记法如下步骤：

1. 首先写一个普通的函数声明：返回类型 名称(参数)
2. 用指针语法包裹函数名：返回类型 (*名称)(参数)

第24.6节：基础知识

就像你可以有指向int、char、float、数组/字符串、结构体等的指针一样，你也可以有指向函数的指针。

声明指针需要函数的返回值类型、函数的名称以及它接收的参数类型。

假设你有如下声明并定义的函数：

```
int addInt(int n, int m){  
    return n+m;  
}
```

你可以声明并初始化一个指向该函数的指针：

```
int (*functionPtrAdd)(int, int) = addInt; // 或 &addInt - &是可选的
```

如果你有一个无返回值的函数，它可能看起来像这样：

```
void Print(void){  
    printf("看妈——没有手，只有指针！");}
```

然后声明指向它的指针将是：

```
void (*functionPtrPrint)(void) = Print;
```

访问函数本身需要对指针进行解引用：

```
sum = (*functionPtrAdd)(2, 3); //将5赋值给sum  
(*functionPtrPrint)(); //将打印Print函数中的文本
```

如本文档中更高级的示例所示，如果函数传递的参数超过几个，声明指向函数的指针可能会变得混乱。如果你有几个指向具有相同“结构”（相同的返回值类型和相同类型参数）的函数的指针，最好使用typedef命令来减少输入量，并使代码更清晰：

```
typedef int (*ptrInt)(int, int);  
  
int Add(int i, int j){  
    return i+j;  
}  
  
int Multiply(int i, int j){  
    return i*j;  
}  
  
int main()
```

```
returnType (*name)(parameters)
```

A mnemonic for writing a function pointer definition is the following procedure:

1. Begin by writing a normal function declaration: returnType name(parameters)
2. Wrap the function name with pointer syntax: returnType (*name)(parameters)

Section 24.6: Basics

Just like you can have a pointer to an **int, char, float, array/string, struct**, etc. - you can have a pointer to a function.

Declaring the pointer takes the *return value of the function*, the *name of the function*, and the *type of arguments/parameters it receives*.

Say you have the following function declared and initialized:

```
int addInt(int n, int m){  
    return n+m;  
}
```

You can declare and initialize a pointer to this function:

```
int (*functionPtrAdd)(int, int) = addInt; // or &addInt - the & is optional
```

If you have a void function it could look like this:

```
void Print(void){  
    printf("look ma' - no hands, only pointers!\n");
```

Then declaring the pointer to it would be:

```
void (*functionPtrPrint)(void) = Print;
```

Accessing the function itself would require dereferencing the pointer:

```
sum = (*functionPtrAdd)(2, 3); //will assign 5 to sum  
(*functionPtrPrint)(); //will print the text in Print function
```

As seen in more advanced examples in this document, declaring a pointer to a function could get messy if the function is passed more than a few parameters. If you have a few pointers to functions that have identical "structure" (same type of return value, and same type of parameters) it's best to use the **typedef** command to save you some typing, and to make the code more clear:

```
typedef int (*ptrInt)(int, int);  
  
int Add(int i, int j){  
    return i+j;  
}  
  
int Multiply(int i, int j){  
    return i*j;  
}  
  
int main()
```

```
{  
ptrInt ptr1 = Add;  
ptrInt ptr2 = Multiply;  
  
printf("%d", (*ptr1)(2,3)); //将打印 5printf("%d", (*ptr2  
(2,3)); //将打印 6return 0;  
}
```

你也可以创建一个函数指针数组。如果所有指针具有相同的“结构”：

```
int (*array[2])(int x, int y); // 可以存放 2 个函数指针  
array[0] = Add;  
array[1] = Multiply;
```

你可以在这里和这里了解更多。[here](#) [here](#)

也可以定义不同类型的函数指针数组，不过每次访问特定函数时都需要进行类型转换。你可以在这里了解更多。

```
{  
ptrInt ptr1 = Add;  
ptrInt ptr2 = Multiply;  
  
printf("%d\n", (*ptr1)(2,3)); //will print 5  
printf("%d\n", (*ptr2)(2,3)); //will print 6  
return 0;  
}
```

You can also create an **Array of function-pointers**. If all the pointers are of the same "structure":

```
int (*array[2])(int x, int y); // can hold 2 function pointers  
array[0] = Add;  
array[1] = Multiply;
```

You can learn more [here](#) and [here](#).

It is also possible to define an array of function-pointers of different types, though that would require casting whenever you want to access the specific function. You can learn more [here](#).

第25章：函数参数

第25.1节：参数按值传递

在C语言中，所有函数参数都是按值传递的，因此在被调用函数中修改传入的参数不会影响调用函数的局部变量。

```
#include <stdio.h>

void modify(int v) {
    printf("modify 1: %d", v); /* 打印0 */    v = 42;
    printf("modify 2: %d", v); /* 打印42 */

int main(void) {
    int v = 0;
    printf("main 1: %d", v); /* 打印0 */    modify(v);
    printf("main 2: %d", v); /* 打印0, 不是42 */return 0;
}
```

你可以使用指针让被调用函数修改调用函数的局部变量。注意，这不是按引用传递，而是传递指向局部变量的指针值。

```
#include <stdio.h>

void modify(int* v) {
    printf("modify 1: %d", *v); /* 打印0 */*v = 42;
    printf("修改 2: %d", *v); /* 打印 42 */

int main(void) {
    int v = 0;
    printf("主函数 1: %d", v); /* 打印 0 */    修改(&v);
    printf("主函数 2: %d", v); /* 打印 42 */return 0;
}
```

但是返回局部变量的地址给被调用者会导致未定义行为。参见超出生命周期的变量指针解引用。

第 25.2 节：向函数传递数组

```
int 获取好友列表(size_t 大小, int 好友索引[]) {
    size_t i = 0;
    for (; i < 大小; i++) {
        好友索引[i] = i;
    }
}

/* 类型 "void" 和变长数组 ("int friend_indexes[static size]") 至少需要 C99 标准。
在 C11 中, 变长数组是可选的。 */
void 获取好友列表(size_t 大小, int 好友索引[static 大小]) {
```

版本 ≥ C99 版本 < C11

/* Type "void" and VLAs ("int friend_indexes[static size]") require C99 at least.
In C11 VLAs are optional. */
void getListOfFriends(size_t size, int friend_indexes[static size]) {

Chapter 25: Function Parameters

Section 25.1: Parameters are passed by value

In C, all function parameters are passed by value, so modifying what is passed in callee functions won't affect caller functions' local variables.

```
#include <stdio.h>

void modify(int v) {
    printf("modify 1: %d\n", v); /* 0 is printed */
    v = 42;
    printf("modify 2: %d\n", v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(v);
    printf("main 2: %d\n", v); /* 0 is printed, not 42 */
    return 0;
}
```

You can use pointers to let callee functions modify caller functions' local variables. Note that this is not *pass by reference* but the pointer *values* pointing at the local variables are passed.

```
#include <stdio.h>

void modify(int* v) {
    printf("modify 1: %d\n", *v); /* 0 is printed */
    *v = 42;
    printf("modify 2: %d\n", *v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(&v);
    printf("main 2: %d\n", v); /* 42 is printed */
    return 0;
}
```

However returning the address of a local variable to the callee results in undefined behaviour. See Dereferencing a pointer to variable beyond its lifetime.

Section 25.2: Passing in Arrays to Functions

```
int getListOfFriends(size_t size, int friend_indexes[]) {
    size_t i = 0;
    for (; i < size; i++) {
        friend_indexes[i] = i;
    }
}
```

Version ≥ C99 Version < C11

/* Type "void" and VLAs ("int friend_indexes[static size]") require C99 at least.
In C11 VLAs are optional. */
void getListOfFriends(size_t size, int friend_indexes[static size]) {

```

size_t i = 0;
for (; i < size; i++) {
    friend_indexes[i] = 1;
}

```

这里函数参数的 [] 中的 static 要求传入的数组参数必须至少包含指定数量的元素（即 size 个元素）。为了能够使用该特性，我们必须确保 size 参数在参数列表中位于数组参数之前。

像这样使用 getListOfFriends()：

```

#define LIST_SIZE (50)

int main(void) {
    size_t size_of_list = LIST_SIZE;
    int friends_indexes[size_of_list];

getListOfFriends(size_of_list, friend_indexes); /* friend_indexes 会退化为指向其第一个元素地址的指针：
                                                &friend_indexes[0] */

/* 这里 friend_indexes 包含：{0, 1, 2, ..., 49}; */

    return 0;
}

```

另见

将多维数组传递给函数

第25.3节：函数参数执行顺序

C语言中参数的执行顺序是不确定的。这里可能从左到右执行，也可能从右到左执行。具体顺序取决于实现。

```

#include <stdio.h>

void function(int a, int b)
{
    printf("%d %d", a, b);
}

int main(void)
{
    int a = 1;
    function(a++, ++a);
    return 0;
}

```

第25.4节：使用指针参数返回多个值

C语言中一个常见的模式，是使用指针来轻松模拟从函数返回多个值。

```
#include <stdio.h>
```

```

size_t i = 0;
for (; i < size; i++) {
    friend_indexes[i] = 1;
}

```

Here the `static` inside the [] of the function parameter, request that the argument array must have at least as many elements as are specified (i.e. size elements). To be able to use that feature we have to ensure that the size parameter comes before the array parameter in the list.

Use `getListOfFriends()` like this:

```

#define LIST_SIZE (50)

int main(void) {
    size_t size_of_list = LIST_SIZE;
    int friends_indexes[size_of_list];

getListOfFriends(size_of_list, friend_indexes); /* friend_indexes decays to a pointer to the
                                                address of its 1st element:
                                                &friend_indexes[0] */

/* Here friend_indexes carries: {0, 1, 2, ..., 49}; */

    return 0;
}

```

See also

Passing multidimensional arrays to a function

Section 25.3: Order of function parameter execution

The order of execution of parameters is undefined in C programming. Here it may execute from left to right or from right to left. The order depends on the implementation.

```

#include <stdio.h>

void function(int a, int b)
{
    printf("%d %d\n", a, b);
}

int main(void)
{
    int a = 1;
    function(a++, ++a);
    return 0;
}

```

Section 25.4: Using pointer parameters to return multiple values

A common pattern in C, to easily imitate returning multiple values from a function, is to use pointers.

```
#include <stdio.h>
```

```

void Get( int* c , double* d )
{
    *c = 72;
    *d = 175.0;
}

int main(void)
{
    int a = 0;
    double b = 0.0;

Get( &a , &b );

printf("a: %d, b: %f", a , b );return 0;

}

```

第25.5节：返回包含错误代码值的结构体的函数示例

大多数返回值的函数示例涉及将指针作为参数之一传入，以允许函数修改指针指向的值，类似如下。函数的实际返回值通常是某种类型，如int，用于指示结果的状态，即是否成功。

```

int func (int *pIvalue)
{
    int iRetStatus = 0;          /* 默认状态为无变化 */

    if (*pIvalue > 10) {
        *pIvalue = *pIvalue * 45; /* 修改指针指向的值 */
        iRetStatus = 1;           /* 表示值已被修改 */
    }

    return iRetStatus;          /* 返回错误代码 */
}

```

但是你也可以使用struct作为返回值，这样可以同时返回错误状态和其他值。例如。

```

typedef struct {
    int    iStat;      /* 返回状态 */
    int    iValue;     /* 返回值 */
} RetValue;

RetValue func (int iValue)
{
    RetValue iRetStatus = {0, iValue};

    if (iValue > 10) {
        iRetStatus.iValue = iValue * 45;
        iRetStatus.iStat = 1;
    }

    return iRetStatus;
}

```

该函数随后可以像以下示例那样使用。

```

void Get( int* c , double* d )
{
    *c = 72;
    *d = 175.0;
}

int main(void)
{
    int a = 0;
    double b = 0.0;

    Get( &a , &b );

    printf("a: %d, b: %f\n", a , b );

    return 0;
}

```

Section 25.5: Example of function returning struct containing values with error codes

Most examples of a function returning a value involve providing a pointer as one of the arguments to allow the function to modify the value pointed to, similar to the following. The actual return value of the function is usually some type such as an `int` to indicate the status of the result, whether it worked or not.

```

int func (int *pIvalue)
{
    int iRetStatus = 0;          /* Default status is no change */

    if (*pIvalue > 10) {
        *pIvalue = *pIvalue * 45; /* Modify the value pointed to */
        iRetStatus = 1;           /* indicate value was changed */
    }

    return iRetStatus;          /* Return an error code */
}

```

However you can also use a `struct` as a return value which allows you to return both an error status along with other values as well. For instance.

```

typedef struct {
    int    iStat;      /* Return status */
    int    iValue;     /* Return value */
} RetValue;

RetValue func (int iValue)
{
    RetValue iRetStatus = {0, iValue};

    if (iValue > 10) {
        iRetStatus.iValue = iValue * 45;
        iRetStatus.iStat = 1;
    }

    return iRetStatus;
}

```

This function could then be used like the following sample.

```
int usingFunc (int iValue)
{
    RetValue iRet = func (iValue);

    if (iRet.iStat == 1) {
        /* 使用返回值 iRet.iValue 进行操作 */
    }
    return 0;
}
```

或者它可以像以下这样使用。

```
int usingFunc (int iValue)
{
    RetValue iRet;

    if ( (iRet = func (iValue)).iStat == 1 ) {
        /* 使用返回值 iRet.iValue 进行操作 */
    }
    return 0;
}
```

```
int usingFunc (int iValue)
{
    RetValue iRet = func (iValue);

    if (iRet.iStat == 1) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Or it could be used like the following.

```
int usingFunc (int iValue)
{
    RetValue iRet;

    if ( (iRet = func (iValue)).iStat == 1 ) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

第26章：向函数传递二维数组

第26.1节：向函数传递二维数组

向函数传递二维数组看起来简单且显而易见，我们愉快地写道：

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int **, int, int);

int main()
{
    int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
    int n = ROWS;
    int m = COLS;

    fun1(array_2D, n, m);

    return EXIT_SUCCESS;
}

void fun1(int **a, int n, int m)
{
    int i, j;
    for (i = 0; i < n; i++) {for (j =
        0; j < m; j++) {printf("array[%d][%d]=%d", i, j, a[i][j]);}
    }
}
```

但是这里的编译器，GCC 4.9.4 版本，并不太支持。

```
$ gcc-4.9 -O3 -g3 -W -Wall -Wextra -std=c11 passarr.c -o passarr
passarr.c: 在函数 ‘main’ 中：
passarr.c:16:8: 警告：传递给 ‘fun1’ 的第1个参数类型不兼容
fun1(array_2D, n, m);
^
passarr.c:8:6: 注意：期望类型为 ‘int **’ 但参数类型为 ‘int (*)[2]’
void fun1(int **, int, int);
```

原因有两个：主要问题是数组不是指针，第二个不便是所谓的指针衰减。将数组传递给函数时，数组会衰减为指向数组第一个元素的指针——对于二维数组来说，它会衰减为指向第一行的指针，因为在C语言中数组是按行优先排序的。

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int, int);
```

Chapter 26: Pass 2D-arrays to functions

Section 26.1: Pass a 2D-array to a function

Passing a 2d array to a functions seems simple and obvious and we happily write:

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int **, int, int);

int main()
{
    int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
    int n = ROWS;
    int m = COLS;

    fun1(array_2D, n, m);

    return EXIT_SUCCESS;
}

void fun1(int **a, int n, int m)
{
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf("array[%d][%d]=%d\n", i, j, a[i][j]);
        }
    }
}
```

But the compiler, here GCC in version 4.9.4 , does not appreciate it well.

```
$ gcc-4.9 -O3 -g3 -W -Wall -Wextra -std=c11 passarr.c -o passarr
passarr.c: In function ‘main’:
passarr.c:16:8: warning: passing argument 1 of ‘fun1’ from incompatible pointer type
fun1(array_2D, n, m);
^
passarr.c:8:6: note: expected ‘int **’ but argument is of type ‘int (*)[2]’
void fun1(int **, int, int);
```

The reasons for this are twofold: the main problem is that arrays are not pointers and the second inconvenience is the so called *pointer decay*. Passing an array to a function will decay the array to a pointer to the first element of the array—in the case of a 2d array it decays to a pointer to the first row because in C arrays are sorted row-first.

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int, int);
```

```

int main()
{
    int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
    int n = ROWS;
    int m = COLS;

fun1(array_2D, n, m);

    return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int n, int m)
{
    int i, j;
    for (i = 0; i < n; i++) {for (j =
        0; j < m; j++) {printf("array[%d][%d]=%d", i, j, a[i][j]);}
    }
}

```

必须传递行数，行数无法计算。

```

#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int);

int main()
{
    int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
    int rows = ROWS;

/* 这里可行，因为 array_2d 仍在作用域内且仍是数组 */
printf("MAIN: %zu", sizeof(array_2D)/sizeof(array_2D[0]));

fun1(array_2D, rows);

    return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int rows)
{
    int i, j;
    int n, m;

    n = rows;
    /* 可行，因为该信息作为"COLS"传递了。
这也是多余的，因为该值在编译时（在"COLS"中）已知。*/
    m = (int) (sizeof(a[0])/sizeof(a[0][0]));

/* 这里不起作用，因为“指针衰减”中的“衰减”是字面意思——信息丢失。*/
printf("FUN1: %zu", sizeof(a)/sizeof(a[0]));for (i = 0; i <
n; i++) {
    for (j = 0; j < m; j++) {printf("array[%d][%d]=%d", i, j, a[i][j]);
}
}

```

```

int main()
{
    int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
    int n = ROWS;
    int m = COLS;

fun1(array_2D, n, m);

    return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int n, int m)
{
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf("array[%d][%d]=%d\n", i, j, a[i][j]);
        }
    }
}

```

It is necessary to pass the number of rows, they cannot be computed.

```

#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int);

int main()
{
    int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
    int rows = ROWS;

/* works here because array_2d is still in scope and still an array */
printf("MAIN: %zu\n", sizeof(array_2D)/sizeof(array_2D[0]));

fun1(array_2D, rows);

    return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int rows)
{
    int i, j;
    int n, m;

    n = rows;
    /* Works, because that information is passed (as "COLS").
It is also redundant because that value is known at compile time (in "COLS"). */
    m = (int) (sizeof(a[0])/sizeof(a[0][0]));

/* Does not work here because the "decay" in "pointer decay" is meant
literally--information is lost. */
printf("FUN1: %zu\n", sizeof(a)/sizeof(a[0]));

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf("array[%d][%d]=%d\n", i, j, a[i][j]);
        }
    }
}

```

```
    }
}
```

版本 = C99

列数是预定义的，因此在编译时固定，但当前C标准的前身（即ISO/IEC 9899:1999，当前为ISO/IEC 9899:2011）实现了可变长度数组（VLA）（TODO: 链接），虽然当前标准将其设为可选，但几乎所有现代C编译器都支持它（TODO: 检查MS Visual Studio现在是否支持）。

```
#include <stdio.h>
#include <stdlib.h>

/* 所有检查均省略！ */

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
    int 行数, 列数, i, j;

    if(argc != 3){
        fprintf(stderr,"用法: %s 行数 列数", argv[0]);exit(EXIT_FAILURE);
    }

    rows = atoi(argv[1]);
    cols = atoi(argv[2]);

    int array_2D[rows][cols];

    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            array_2D[i][j] = (i + 1) * (j + 1);printf("array[%d][%d]=%d", i, j, array_2D[i][j]);
        }
    }

    fun1(array_2D, rows, cols);

    exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
    int i, j;
    int n, m;

    n = rows;
    /* 不再起作用，因为不再指定大小
    m = (int) (sizeof(a[0])/sizeof(a[0][0])); */
    m = cols;

    for (i = 0; i < n; i++) {for (j =
        0; j < m; j++) {printf("array[%d][%d]=%d", i, j, a[i][j]);}
    }
}
```

```
    }
}
}
```

Version = C99

The number of columns is predefined and hence fixed at compile time, but the predecessor to the current C-standard (that was ISO/IEC 9899:1999, current is ISO/IEC 9899:2011) implemented VLAs (TODO: link it) and although the current standard made it optional, almost all modern C-compilers support it (TODO: check if MS Visual Studio supports it now).

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMITTED! */

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
    int rows, cols, i, j;

    if(argc != 3){
        fprintf(stderr,"Usage: %s rows cols\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    rows = atoi(argv[1]);
    cols = atoi(argv[2]);

    int array_2D[rows][cols];

    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            array_2D[i][j] = (i + 1) * (j + 1);
            printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
        }
    }

    fun1(array_2D, rows, cols);

    exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
    int i, j;
    int n, m;

    n = rows;
    /* Does not work anymore, no sizes are specified anymore
    m = (int) (sizeof(a[0])/sizeof(a[0][0])); */
    m = cols;

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf("array[%d][%d]=%d\n", i, j, a[i][j]);
        }
    }
}
```

这不起作用，编译器报错：

```
$ gcc-4.9 -O3 -g3 -W -Wall -Wextra -std=c99 passarr.c -o passarr
passarr.c: 在函数 'fun1' 中:
passarr.c:168:7: 错误：对未指定边界的数组的无效使用printf("array[%d][%d]=%d", i, j, a
[i][j]);
```

如果我们故意在函数调用中出错，将声明改为

`void fun1(int **a, int rows, int cols)`，情况会更清楚一些。这会导致编译器以不同但同样模糊的方式报错

```
$ gcc-4.9 -O3 -g3 -W -Wall -Wextra -std=c99 passarr.c -o passarr
passarr.c: 在函数 'main' 中:
passarr.c:208:8: 警告：传递给 'fun1' 的第 1 个参数的指针类型不兼容
    fun1(array_2D, rows, cols);
           ^
passarr.c:185:6: 注释：预期为 'int **' 但参数类型为 'int (*)[(sizetype)(cols)]'
  void fun1(int **, int rows, int cols);
```

我们可以有几种反应方式，其中一种是忽略所有内容，进行一些难以理解的指针操作：

```
#include <stdio.h>
#include <stdlib.h>

/* 所有检查均省略 ! */

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
    int 行数, 列数, i, j;

    if(argc != 3){
        fprintf(stderr,"用法: %s 行数 列数", argv[0]);exit(EXIT_FAILURE);
    }
}

rows = atoi(argv[1]);
cols = atoi(argv[2]);

int array_2D[rows][cols];
printf("创建一个有 %d 行 %d 列的数组", rows, cols);for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
        array_2D[i][j] = i * cols + j;printf("array[%d][%d]=%d", i, j, array_2D[i][j]);
    }
}

fun1(array_2D, rows, cols);

exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
    int i, j;
    int n, m;
```

This does not work, the compiler complains:

```
$ gcc-4.9 -O3 -g3 -W -Wall -Wextra -std=c99 passarr.c -o passarr
passarr.c: In function 'fun1':
passarr.c:168:7: error: invalid use of array with unspecified bounds
printf("array[%d][%d]=%d\n", i, j, a[i][j]);
```

It becomes a bit clearer if we intentionally make an error in the call of the function by changing the declaration to `void fun1(int **a, int rows, int cols)`. That causes the compiler to complain in a different, but equally nebulous way

```
$ gcc-4.9 -O3 -g3 -W -Wall -Wextra -std=c99 passarr.c -o passarr
passarr.c: In function 'main':
passarr.c:208:8: warning: passing argument 1 of 'fun1' from incompatible pointer type
    fun1(array_2D, rows, cols);
           ^
passarr.c:185:6: note: expected 'int **' but argument is of type 'int (*)[(sizetype)(cols)]'
  void fun1(int **, int rows, int cols);
```

We can react in several ways, one of it is to ignore all of it and do some illegible pointer juggling:

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMITTED! */

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
    int rows, cols, i, j;

    if(argc != 3){
        fprintf(stderr,"Usage: %s rows cols\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    rows = atoi(argv[1]);
    cols = atoi(argv[2]);

    int array_2D[rows][cols];
    printf("Make array with %d rows and %d columns\n", rows, cols);
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            array_2D[i][j] = i * cols + j;
            printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
        }
    }

    fun1(array_2D, rows, cols);

    exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
    int i, j;
    int n, m;
```

```

n = rows;
m = cols;

printf("打印一个有 %d 行 %d 列的数组", rows, cols);for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        printf("array[%d][%d]=%d", i, j, *( (*a) + (i * cols + j)));
    }
}

```

或者我们正确地做，将所需信息传递给fun1。为此，我们需要重新排列传递给fun1的参数：列的大小必须在数组声明之前。为了保持代码更易读，保存行数的变量位置也做了调整，现在放在了最前面。

```

#include <stdio.h>
#include <stdlib.h>

/* 所有检查均省略 */

void fun1(int 行数, int 列数, int (*a)[列数]);

int main(int 参数个数, char **参数数组)
{
    int 行数, 列数, i, j;

    if(argc != 3){
        fprintf(stderr,"用法: %s 行数 列数",argv[0]);exit(EXIT_FAILURE);
    }

    rows = atoi(argv[1]);
    cols = atoi(argv[2]);

    int array_2D[rows][cols];
    printf("创建一个有 %d 行 %d 列的数组", rows, cols);for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            array_2D[i][j] = i * cols + j;printf("array[%d][%d]=%d", i, j, array_2D[i][j]);
        }
    }

    fun1(行数, 列数, 二维数组);

    exit(EXIT_SUCCESS);
}

void fun1(int 行数, int 列数, int (*a)[列数])
{
    int i, j;
    int n, m;

    n = rows;
    m = cols;

    printf("打印一个有 %d 行 %d 列的数组", rows, cols);for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {printf("array[%d][%d]=%d", i, j, a[i][j]);}
    }
}

```

```

n = rows;
m = cols;

printf("\nPrint array with %d rows and %d columns\n", rows, cols);
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        printf("array[%d][%d]=%d\n", i, j, *( (*a) + (i * cols + j)));
    }
}

```

Or we do it right and pass the needed information to fun1. To do so we need to rearrange the arguments to fun1: the size of the column must come before the declaration of the array. To keep it more readable the variable holding the number of rows has changed its place, too, and is first now.

```

#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMITTED!*/

void fun1(int rows, int cols, int (*a)[cols]);

int main(int argc, char **argv)
{
    int rows, cols, i, j;

    if(argc != 3){
        fprintf(stderr,"Usage: %s rows cols\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    rows = atoi(argv[1]);
    cols = atoi(argv[2]);

    int array_2D[rows][cols];
    printf("Make array with %d rows and %d columns\n", rows, cols);
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            array_2D[i][j] = i * cols + j;
            printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
        }
    }

    fun1(rows, cols, array_2D);

    exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int (*a)[cols])
{
    int i, j;
    int n, m;

    n = rows;
    m = cols;

    printf("\nPrint array with %d rows and %d columns\n", rows, cols);
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf("array[%d][%d]=%d\n", i, j, a[i][j]);
        }
    }
}

```

```
}
```

这对某些人来说看起来很别扭，他们认为变量的顺序不应该重要。这并不是什么大问题，只需声明一个指针并让它指向数组即可。

```
#include <stdio.h>
#include <stdlib.h>

/* 所有检查均省略 */

void fun1(int 行数, int 列数, int **);

int main(int 参数个数, char **参数数组)
{
    int 行数, 列数, i, j;

    if(argc != 3){
        fprintf(stderr,"用法: %s 行数 列数",argv[0]);exit(EXIT_FAILURE
E);
    }

rows = atoi(argv[1]);
cols = atoi(argv[2]);

int array_2D[rows][cols];
printf("创建一个有 %d 行 %d 列的数组", rows, cols);for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
array_2D[i][j] = i * cols + j;printf("array[%d]
[%d]=%d", i, j, array_2D[i][j]);
    }
}

// 一个包含“行数”个指向“int”的指针。再次是一个变长数组 (VLA)
int *a[行数];
// 初始化它们以指向各个行
for (i = 0; i < 行数; i++) {
a[i] = 二维数组[i];
}

fun1(rows, cols, a);

exit(EXIT_SUCCESS);
}

void fun1(int 行数, int 列数, int **数组)
{
    int i, j;
    int n, m;

n = rows;
m = cols;

printf("打印一个有 %d 行 %d 列的数组", rows, cols);for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {printf(
        "array[%d][%d]=%d", i, j, a[i][j]);
    }
}
```

```
}
```

This looks awkward to some people, who hold the opinion that the order of variables should not matter. That is not much of a problem, just declare a pointer and let it point to the array.

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMITTED! */

void fun1(int rows, int cols, int **);

int main(int argc, char **argv)
{
    int rows, cols, i, j;

    if(argc != 3){
        fprintf(stderr,"Usage: %s rows cols\n", argv[0]);
        exit(EXIT_FAILURE);
    }

rows = atoi(argv[1]);
cols = atoi(argv[2]);

int array_2D[rows][cols];
printf("Make array with %d rows and %d columns\n", rows, cols);
for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
array_2D[i][j] = i * cols + j;
printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
}

// a "rows" number of pointers to "int". Again a VLA
int *a[rows];
// initialize them to point to the individual rows
for (i = 0; i < rows; i++) {
    a[i] = array_2D[i];
}

fun1(rows, cols, a);

exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int **a)
{
    int i, j;
    int n, m;

n = rows;
m = cols;

printf("\nPrint array with %d rows and %d columns\n", rows, cols);
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
}
}
```

第26.2节：将一维数组用作二维数组

通常最简单的解决方案是将二维及更高维数组作为一维内存传递。

```
/* 创建运行时确定维度的二维数组 */
double *矩阵 = malloc(宽度 * 高度 * sizeof(double));

/* 初始化它 (为了说明, 我们希望对角线上的值为1.0) */
int x, y;
for (y = 0; y < 高度; y++)
{
    for (x = 0; x < 宽度; x++)
    {
        if (x == y)
            matrix[y * width + x] = 1.0;
        else
            matrix[y * width + x] = 0.0;
    }
}

/* 传递给子程序 */
manipulate_matrix(matrix, width, height);

/* 对矩阵进行操作, 例如乘以2 */
void manipulate_matrix(double *matrix, int width, int height)
{
    int x, y;

    for (y = 0; y < 高度; y++)
    {
        for (x = 0; x < 宽度; x++)
        {
            matrix[y * width + x] *= 2.0;
        }
    }
}
```

Section 26.2: Using flat arrays as 2D arrays

Often the easiest solution is simply to pass 2D and higher arrays around as flat memory.

```
/* create 2D array with dimensions determined at runtime */
double *matrix = malloc(width * height * sizeof(double));

/* initialise it (for the sake of illustration we want 1.0 on the diagonal) */
int x, y;
for (y = 0; y < height; y++)
{
    for (x = 0; x < width; x++)
    {
        if (x == y)
            matrix[y * width + x] = 1.0;
        else
            matrix[y * width + x] = 0.0;
    }
}

/* pass it to a subroutine */
manipulate_matrix(matrix, width, height);

/* do something with the matrix, e.g. scale by 2 */
void manipulate_matrix(double *matrix, int width, int height)
{
    int x, y;

    for (y = 0; y < height; y++)
    {
        for (x = 0; x < width; x++)
        {
            matrix[y * width + x] *= 2.0;
        }
    }
}
```

第27章：错误处理

第27.1节：errno

当标准库函数失败时，通常会将errno设置为相应的错误代码。C标准要求errno至少设置3个值：

值	含义
EDOM	域错误
ERANGE	范围错误
EILSEQ	非法多字节字符序列

第27.2节：strerror

如果perror不够灵活，您可以通过调用

<string.h> 中的 strerror 来获得用户可读的错误描述。

```
int main(int argc, char *argv[])
{
FILE *fout;
int last_error = 0;

if ((fout = fopen(argv[1], "w")) == NULL) {
    last_error = errno;
    /* 重置 errno 并继续 */
errno = 0;
}

/* 进行一些处理并尝试以不同方式打开文件，然后 */

if (last_error) {
    fprintf(stderr, "fopen: 无法打开 %s 进行写入: %s",
            argv[1], strerror(last_error));
    fputs("祈祷好运并继续", stderr);
}

/* 进行其他处理 */

return EXIT_SUCCESS;
}
```

第27.3节：perror

要向stderr打印用户可读的错误信息，请调用来自<stdio.h>的perror函数。

```
int main(int argc, char *argv[])
{
FILE *fout;

if ((fout = fopen(argv[1], "w")) == NULL) {
    perror("fopen: 无法打开文件进行写入");
    return EXIT_FAILURE;
}
return EXIT_SUCCESS;
}
```

Chapter 27: Error handling

Section 27.1: errno

When a standard library function fails, it often sets errno to the appropriate error code. The C standard requires at least 3 values for errno be set:

Value	Meaning
EDOM	Domain error
ERANGE	Range error
EILSEQ	Illegal multi-byte character sequence

Section 27.2: strerror

If perror is not flexible enough, you may obtain a user-readable error description by calling `strerror` from `<string.h>`.

```
int main(int argc, char *argv[])
{
FILE *fout;
int last_error = 0;

if ((fout = fopen(argv[1], "w")) == NULL) {
    last_error = errno;
    /* reset errno and continue */
    errno = 0;
}

/* do some processing and try opening the file differently, then */

if (last_error) {
    fprintf(stderr, "fopen: Could not open %s for writing: %s",
            argv[1], strerror(last_error));
    fputs("Cross fingers and continue", stderr);
}

/* do some other processing */

return EXIT_SUCCESS;
}
```

Section 27.3: perror

To print a user-readable error message to stderr, call perror from <stdio.h>.

```
int main(int argc, char *argv[])
{
FILE *fout;

if ((fout = fopen(argv[1], "w")) == NULL) {
    perror("fopen: Could not open file for writing");
    return EXIT_FAILURE;
}
return EXIT_SUCCESS;
}
```

这将打印与当前errno值相关的错误信息。

This will print an error message concerning the current value of errno.

第28章：未定义行为

在C语言中，某些表达式会产生未定义行为。标准明确选择不定义编译器在遇到此类表达式时应如何表现。因此，编译器可以自由决定其行为，可能产生有用的结果、意外的结果，甚至崩溃。

调用未定义行为的代码可能在特定系统和特定编译器上按预期工作，但很可能在另一个系统、不同的编译器、编译器版本或编译器设置下无法正常工作。

第28.1节：解引用超出其

生命周期的变量指针

```
int* foo(int bar)
{
    int baz = 6;
    baz += bar;
    return &baz; /* (&baz) 被复制到foo外的新内存位置。 */
} /* (1) baz和bar的生命周期在此结束，因为它们具有自动存储
   * 持续时间（局部变量），因此返回的指针无效！ */

int main (void)
{
    int* p;

    p = foo(5); /* (2) 此表达式的行为是未定义的 */
    *p = *p - 6; /* (3) 此处为未定义行为 */

    return 0;
}
```

一些编译器会友好地指出这一点。例如，gcc 会发出警告：

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

并且clang会发出如下警告：

```
警告：返回了与局部变量 'baz' 关联的栈内存地址
[-Wreturn-stack-address]
```

针对上述代码。但编译器可能无法在复杂代码中提供帮助。

- (1) 返回声明为static的变量的引用是定义良好的行为，因为该变量在离开当前作用域后不会被销毁。
- (2) 根据 ISO/IEC 9899:2011 6.2.4 §2，“当指针所指向的对象生命周期结束时，指针的值变为不确定。”
- (3) 解引用函数foo返回的指针是未定义行为，因为它所引用的内存包含不确定的值。

第28.2节：复制重叠的内存

各种标准库函数的效果之一是将字节序列从一个内存

Chapter 28: Undefined behavior

In C, some expressions yield *undefined behavior*. The standard explicitly chooses to not define how a compiler should behave if it encounters such an expression. As a result, a compiler is free to do whatever it sees fit and may produce useful results, unexpected results, or even crash.

Code that invokes UB may work as intended on a specific system with a specific compiler, but will likely not work on another system, or with a different compiler, compiler version or compiler settings.

Section 28.1: Dereferencing a pointer to variable beyond its lifetime

```
int* foo(int bar)
{
    int baz = 6;
    baz += bar;
    return &baz; /* (&baz) copied to new memory location outside of foo. */
} /* (1) The lifetime of baz and bar end here as they have automatic storage
   * duration (local variables), thus the returned pointer is not valid! */

int main (void)
{
    int* p;

    p = foo(5); /* (2) this expression's behaviour is undefined */
    *p = *p - 6; /* (3) Undefined behaviour here */

    return 0;
}
```

Some compilers helpfully point this out. For example, gcc warns with:

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

and clang warns with:

```
warning: address of stack memory associated with local variable 'baz' returned
[-Wreturn-stack-address]
```

for the above code. But compilers may not be able to help in complex code.

- (1) Returning reference to variable declared `static` is defined behaviour, as the variable is not destroyed after leaving current scope.
- (2) According to ISO/IEC 9899:2011 6.2.4 §2, "The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime."
- (3) Dereferencing the pointer returned by the function `foo` is undefined behaviour as the memory it references holds an indeterminate value.

Section 28.2: Copying overlapping memory

A wide variety of standard library functions have among their effects copying byte sequences from one memory

区域复制到另一个区域。当源和目标区域重叠时，这些函数中的大多数行为是未定义的。

例如，以下代码...

```
#include <string.h> /* 用于 memcpy() */  
  
char str[19] = "This is an example";  
memcpy(str + 7, str, 10);
```

尝试复制10个字节，但源内存区域和目标内存区域有3个字节重叠。为了便于理解：



由于重叠，结果行为是未定义的。

具有此类限制的标准库函数包括memcpy()、strcpy()、strcat()、sprintf()和sscanf()。标准对这些及其他几个函数的说明是：

如果复制发生在重叠的对象之间，行为是未定义的。

函数memmove()是该规则的主要例外。其定义指定该函数的行为就像先将源数据复制到临时缓冲区，然后再写入目标地址。对于源和目标区域重叠没有例外，也不需要例外，因此memmove()在这种情况下具有明确定义的行为。

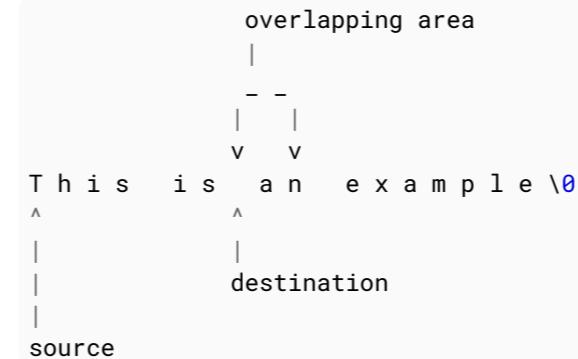
这种区别反映了效率vs通用性的权衡。此类函数执行的复制通常发生在不相交的内存区域之间，并且通常可以在开发时知道特定的内存复制实例是否属于该类别。假设不重叠可以实现相对更高效的实现，但当假设不成立时，这些实现不能可靠地产生正确结果。大多数C库函数允许采用更高效的实现，而memmove()则填补了空白，处理源和目标可能或确重叠的情况。然而，为了在所有情况下产生正确效果，它必须执行额外的测试和/或采用相对效率较低的实现。

region to another. Most of these functions have undefined behavior when the source and destination regions overlap.

For example, this ...

```
#include <string.h> /* for memcpy() */  
  
char str[19] = "This is an example";  
memcpy(str + 7, str, 10);
```

... attempts to copy 10 bytes where the source and destination memory areas overlap by three bytes. To visualize:



Because of the overlap, the resulting behavior is undefined.

Among the standard library functions with a limitation of this kind are `memcpy()`, `strcpy()`, `strcat()`, `sprintf()`, and `sscanf()`. The standard says of these and several other functions:

If copying takes place between objects that overlap, the behavior is undefined.

The `memmove()` function is the principal exception to this rule. Its definition specifies that the function behaves as if the source data were first copied into a temporary buffer and then written to the destination address. There is no exception for overlapping source and destination regions, nor any need for one, so `memmove()` has well-defined behavior in such cases.

The distinction reflects an efficiency vs. generality tradeoff. Copying such as these functions perform usually occurs between disjoint regions of memory, and often it is possible to know at development time whether a particular instance of memory copying will be in that category. Assuming non-overlap affords comparatively more efficient implementations that do not reliably produce correct results when the assumption does not hold. Most C library functions are allowed the more efficient implementations, and `memmove()` fills in the gaps, serving the cases where the source and destination may or do overlap. To produce the correct effect in all cases, however, it must perform additional tests and / or employ a comparatively less efficient implementation.

Section 28.3: Signed integer overflow

根据C99和C11的第6.5/5段，表达式的求值如果结果不是该表达式类型的可表示值，则产生未定义行为。对于算术类型，这称为“溢出”。无符号整数算术不会溢出，因为第6.2.5/9段适用，导致任何本应超出范围的无符号结果被缩减为范围内值。然而，对于有符号整数类型没有类似规定；它们可能会溢出，产生未定义行为。例如，

```
#include <limits.h>      /* 获取INT_MAX */
```

Per paragraph 6.5/5 of both C99 and C11, evaluation of an expression produces undefined behavior if the result is not a representable value of the expression's type. For arithmetic types, that's called an *overflow*. Unsigned integer arithmetic does not overflow because paragraph 6.2.5/9 applies, causing any unsigned result that otherwise would be out of range to be reduced to an in-range value. There is no analogous provision for *signed* integer types, however; these can and do overflow, producing undefined behavior. For example,

```
#include <limits.h>      /* to get INT_MAX */
```

```
int main(void) {
    int i = INT_MAX + 1; /* 这里发生溢出 */
    return 0;
}
```

此类未定义行为的大多数实例更难以识别或预测。原则上，溢出可能发生在任何对有符号整数的加法、减法或乘法操作中（受通常的算术转换约束），只要没有有效的边界或操作数之间的关系来防止它。例如，以下函数：

```
int square(int x) {
    return x * x; /* 对某些x值会溢出 */
}
```

这是合理的，对于足够小的参数值它能正确工作，但对于较大的参数值其行为是未定义的。仅凭该函数无法判断调用它的程序是否表现出未定义行为。行为因此而产生。这取决于他们传递给它的参数。

另一方面，考虑这个溢出安全的有符号整数算术的简单示例：

```
int zero(int x) {
    return x - x; /* 不会溢出 */
}
```

减法运算符的操作数之间的关系确保减法永远不会溢出。或者考虑这个稍微实用一些的示例：

```
int sizeDelta(FILE *f1, FILE *f2) {
    int count1 = 0;
    int count2 = 0;
    while (fgetc(f1) != EOF) count1++; /* 可能溢出 */
    while (fgetc(f2) != EOF) count2++; /* 可能溢出 */

    return count1 - count2; /* 只要到此无未定义行为，不会溢出 */
}
```

只要计数器各自不溢出，最终减法的操作数都将是非负的。任意两个此类值之间的差都可以用int表示。

第28.4节：未初始化变量的使用

```
int a;
printf("%d", a);
```

变量 a 是一个具有自动存储期的 int 类型。上面的示例代码试图打印一个未初始化的变量（a 从未被初始化）。未初始化的自动变量具有不确定的值；访问这些变量可能导致未定义行为。

注意：具有静态或线程局部存储的变量，包括没有使用 static 关键字的 全局变量，都会被初始化为零或其初始化值。因此，以下代码是合法的。

```
static int b;
printf("%d", b);
```

一个非常常见的错误是没有将用作计数器的变量初始化为0。你向它们添加值，但

```
int main(void) {
    int i = INT_MAX + 1; /* Overflow happens here */
    return 0;
}
```

Most instances of this type of undefined behavior are more difficult to recognize or predict. Overflow can in principle arise from any addition, subtraction, or multiplication operation on signed integers (subject to the usual arithmetic conversions) where there are not effective bounds on or a relationship between the operands to prevent it. For example, this function:

```
int square(int x) {
    return x * x; /* overflows for some values of x */
}
```

is reasonable, and it does the right thing for small enough argument values, but its behavior is undefined for larger argument values. You cannot judge from the function alone whether programs that call it exhibit undefined behavior as a result. It depends on what arguments they pass to it.

On the other hand, consider this trivial example of overflow-safe signed integer arithmetic:

```
int zero(int x) {
    return x - x; /* Cannot overflow */
}
```

The relationship between the operands of the subtraction operator ensures that the subtraction never overflows. Or consider this somewhat more practical example:

```
int sizeDelta(FILE *f1, FILE *f2) {
    int count1 = 0;
    int count2 = 0;
    while (fgetc(f1) != EOF) count1++; /* might overflow */
    while (fgetc(f2) != EOF) count2++; /* might overflow */

    return count1 - count2; /* provided no UB to this point, will not overflow */
}
```

As long as that the counters do not overflow individually, the operands of the final subtraction will both be non-negative. All differences between any two such values are representable as int.

Section 28.4: Use of an uninitialized variable

```
int a;
printf("%d", a);
```

The variable a is an int with automatic storage duration. The example code above is trying to print the value of an uninitialized variable (a was never initialized). Automatic variables which are not initialized have indeterminate values; accessing these can lead to undefined behavior.

Note: Variables with static or thread local storage, including global variables without the static keyword, are initialized to either zero, or their initialized value. Hence the following is legal.

```
static int b;
printf("%d", b);
```

A very common mistake is to not initialize the variables that serve as counters to 0. You add values to them, but

由于初始值是垃圾值，你将触发 未定义行为，例如在问题 终端编译时出现指针警告和奇怪符号 中所示。

示例：

```
#include <stdio.h>intmain(void) {int i, ccounter;for(i = 0; i < 10; ++i)    counter+= i;printf("%d", counter);return 0;}
```

输出：

```
C02QT2UBFVH6-lm:~ gsamaras$ gcc main.c -Wall -o mainmain.c:6:9: warning: 变量 'counter' 在此处使用时未初始化 [-Wuninitialized]    counter += i;          ^~~~~~main.c:4:19: 注释: 初始化变量 'counter' 以消除此警告    int i, counter;          ^= 01生成警告。C02QT2UBFVH6-lm:~ gsamaras$ ./main32812
```

上述规则同样适用于指针。例如，以下代码会导致未定义行为

```
int main(void){    int *p;p++; // 试图递增一个未初始化的指针。}
```

请注意，上述代码本身可能不会导致错误或段错误，但稍后尝试解引用该指针会导致未定义行为。

第28.5节：数据竞争

版本 ≥ C11

C11引入了对多线程执行的支持，这带来了数据竞争的可能性。如果程序中有一个对象被两个不同线程访问¹，且至少有一次访问是非原子的，至少有一次修改了该对象，并且程序语义未能保证这两次访问在时间上不会重叠，则该程序包含数据竞争。² 请注意，实际的并发访问并不是数据竞争的必要条件；数据竞争涵盖了由于不同线程对内存的（允许的）不一致视图而产生的更广泛问题。

考虑以下示例：

```
#include <threads.h>int a = 0;
```

since the initial value is garbage, you will invoke **Undefined Behavior**, such as in the question [Compilation on terminal gives off pointer warning and strange symbols.](#)

Example:

```
#include <stdio.h>int main(void) {    int i, counter;    for(i = 0; i < 10; ++i)        counter += i;    printf("%d\n", counter);    return 0;}
```

Output:

```
C02QT2UBFVH6-lm:~ gsamaras$ gcc main.c -Wall -o mainmain.c:6:9: warning: 变量 'counter' 是未初始化的，当在此处使用时 [-Wuninitialized]    counter += i;          ^~~~~~main.c:4:19: 注意: 为消除此警告，请初始化变量 'counter'    int i, counter;          ^= 01 warning generated.C02QT2UBFVH6-lm:~ gsamaras$ ./main32812
```

The above rules are applicable for pointers as well. For example, the following results in undefined behavior

```
int main(void){    int *p;p++; // Trying to increment an uninitialized pointer.}
```

Note that the above code on its own might not cause an error or segmentation fault, but trying to dereference this pointer later would cause the undefined behavior.

Section 28.5: Data race

Version ≥ C11

C11 introduced support for multiple threads of execution, which affords the possibility of data races. A program contains a data race if an object in it is accessed¹ by two different threads, where at least one of the accesses is non-atomic, at least one modifies the object, and program semantics fail to ensure that the two accesses cannot overlap temporally.² Note well that actual concurrency of the accesses involved is not a condition for a data race; data races cover a broader class of issues arising from (allowed) inconsistencies in different threads' views of memory.

Consider this example:

```
#include <threads.h>int a = 0;
```

```

int Function( void* ignore )
{
    a = 1;

    return 0;
}

int main( void )
{
    thrd_t id;
    thrd_create( &id , Function , NULL );

    int b = a;

    thrd_join( id , NULL );
}

```

主线程调用thrd_create来启动一个新线程运行函数Function。第二个线程修改了 a，主线程读取了 a。两者的访问都不是原子的，且两个线程无论单独还是共同都没有采取措施确保它们不会重叠，因此存在数据竞争。

该程序避免数据竞争的方法包括

- 主线程可以在启动另一个线程之前读取 a；主线程可以在通过 thrd_join 确认另
- 一个线程已终止后读取 a；线程可以通过互斥锁同步访问，每个线程在访问 a 之前锁定该互斥锁，访问后再
- 解锁。

正如互斥选项所示，避免数据竞争并不需要确保特定的操作顺序，例如子线程在主线程读取之前修改 a；只需确保在给定的执行中，一个访问发生在另一个之前即可（以避免数据竞争）。

1 修改或读取一个对象。

2 (引用自 ISO:IEC 9889:201x, 第 5.1.2.4 节“多线程执行和数据竞争”) 如果程序的执行包含两个不同线程中的冲突操作，且至少有一个操作不是原子的，且两者之间没有先后顺序，则该执行包含数据竞争。任何此类数据竞争都会导致未定义行为。

第 28.6 节：读取已释放指针的值

即使只是读取已释放指针的值（即不尝试解引用指针）也是未定义行为（UB），例如：

```

char *p = malloc(5);
free(p);
if (p == NULL) /* 注意：即使不解引用，这也可能是未定义行为 */
{
}

```

引用ISO/IEC 9899:2011, 第 6.2.4 节 §2：

[...] 当指针所指向的对象（或紧接其后的对象）生命周期结束时，该指针的值变为不确定。

```

int Function( void* ignore )
{
    a = 1;

    return 0;
}

int main( void )
{
    thrd_t id;
    thrd_create( &id , Function , NULL );

    int b = a;

    thrd_join( id , NULL );
}

```

The main thread calls thrd_create to start a new thread running function Function. The second thread modifies a, and the main thread reads a. Neither of those access is atomic, and the two threads do nothing either individually or jointly to ensure that they do not overlap, so there is a data race.

Among the ways this program could avoid the data race are

- the main thread could perform its read of a before starting the other thread;
- the main thread could perform its read of a after ensuring via thrd_join that the other has terminated;
- the threads could synchronize their accesses via a mutex, each one locking that mutex before accessing a and unlocking it afterward.

As the mutex option demonstrates, avoiding a data race does not require ensuring a specific order of operations, such as the child thread modifying a before the main thread reads it; it is sufficient (for avoiding a data race) to ensure that for a given execution, one access will happen before the other.

1 Modifying or reading an object.

2 (Quoted from ISO:IEC 9889:201x, section 5.1.2.4 "Multi-threaded executions and data races")
The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

Section 28.6: Read value of pointer that was freed

Even just **reading** the value of a pointer that was freed (i.e. without trying to dereference the pointer) is undefined behavior(UB), e.g.

```

char *p = malloc(5);
free(p);
if (p == NULL) /* NOTE: even without dereferencing, this may have UB */
{
}

```

Quoting ISO/IEC 9899:2011, section 6.2.4 §2:

[...] The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

使用不确定内存的任何操作，包括看似无害的比较或算术运算，如果该值可能是该类型的陷阱表示，则可能导致未定义行为。

第28.7节：在printf中使用错误的格式说明符

在printf的第一个参数中使用错误的格式说明符会导致未定义行为。例如，下面的代码会导致未定义行为：

```
long z = 'B';  
printf("%c", z);
```

这里是另一个例子

```
printf("%f")
```

上述代码行是未定义行为。%f期望的是double类型，但0是int类型。

请注意，如果在编译时开启了适当的选项（如clang和gcc中的-Wformat），编译器通常可以帮助你避免这类情况。以下是一个例子的提示信息：

```
warning: format specifies type 'double' but the argument has type  
      'int' [-Wformat]  
      printf("%f", 0);  
      ~~ ^%d
```

第28.8节：修改字符串字面量

在此代码示例中，字符指针p被初始化为指向字符串字面量的地址。尝试修改字符串字面量会导致未定义行为。

```
char *p = "hello world";  
p[0] = 'H'; // 未定义行为
```

然而，直接修改可变的char数组，或通过指针修改，哪怕其初始化器是字面字符串，通常并不是未定义行为。以下代码是可以的：

```
char a[] = "hello, world";  
char *p = a;  
  
a[0] = 'H';  
p[7] = 'W';
```

这是因为字符串字面量在每次数组初始化时都会被有效地复制到数组中（对于静态存储期变量复制一次，对于自动或线程存储期变量每次创建数组时复制一次—对于动态分配存储期变量则不初始化），修改数组内容是允许的。

第28.9节：向printf的%s转换传递空指针

printf的%s转换要求对应的参数是指向字符类型数组首元素的指针。空指针不指向任何字符类型数组的首元素，因此以下行为是未定义的：

```
char *foo = NULL;
```

The use of indeterminate memory for anything, including apparently harmless comparison or arithmetic, can have undefined behavior if the value can be a trap representation for the type.

Section 28.7: Using incorrect format specifier in printf

Using an incorrect format specifier in the first argument to `printf` invokes undefined behavior. For example, the code below invokes undefined behavior:

```
long z = 'B';  
printf("%c\n", z);
```

Here is another example

```
printf("%f\n", 0);
```

Above line of code is undefined behavior. %f expects double. However 0 is of type int.

Note that your compiler usually can help you avoid cases like these, if you turn on the proper flags during compiling (-Wformat in clang and gcc). From the last example:

```
warning: format specifies type 'double' but the argument has type  
      'int' [-Wformat]  
      printf("%f\n", 0);  
      ~~ ^  
      %d
```

Section 28.8: Modify string literal

In this code example, the char pointer p is initialized to the address of a string literal. Attempting to modify the string literal has undefined behavior.

```
char *p = "hello world";  
p[0] = 'H'; // Undefined behavior
```

However, modifying a mutable array of char directly, or through a pointer is naturally not undefined behavior, even if its initializer is a literal string. The following is fine:

```
char a[] = "hello, world";  
char *p = a;  
  
a[0] = 'H';  
p[7] = 'W';
```

That's because the string literal is effectively copied to the array each time the array is initialized (once for variables with static duration, each time the array is created for variables with automatic or thread duration — variables with allocated duration aren't initialized), and it is fine to modify array contents.

Section 28.9: Passing a null pointer to printf %s conversion

The %s conversion of `printf` states that the corresponding argument *a pointer to the initial element of an array of character type*. A null pointer does not point to the initial element of any array of character type, and thus the behavior of the following is undefined:

```
char *foo = NULL;
```

```
printf("%s", foo); /* 未定义行为 */
```

然而，未定义行为并不总意味着程序崩溃—一些系统会采取措施避免空指针解引用时通常发生的崩溃。例如，Glibc已知会打印

(null)

对于上述代码。然而，只需在格式字符串中添加一个换行符，你就会遇到崩溃：

```
char *foo = 0;
printf("%s", foo); /* 未定义行为 */
```

在这种情况下，原因是GCC有一个优化，会将printf("%s", argument);转换为对puts的调用puts(argument)，而Glibc中的puts不处理空指针。所有这些行为都是符合标准的。

注意，null指针不同于空字符串。因此，以下代码是有效的且没有未定义行为。它只会打印一个换行符：

```
char *foo = "";pri
ntf("%s", foo);
```

第28.10节：在两个序列点之间多次修改同一对象

```
int i = 42;
i = i++; /* 赋值改变变量，后置递增也改变变量 */
int a = i++ + i--;
```

像这样的代码常常引发关于 i “结果值”的猜测。然而，C 标准并未指定结果，而是规定评估此类表达式会产生未定义行为。在 C2011 之前，标准通过所谓的序列点来形式化这些规则：

在前一个和后一个序列点之间，标量对象的存储值最多只能被表达式的求值修改一次。此外，先前的值只能被读取以确定要存储的值。

(C99 标准，第6.5节，第2段)

该方案被证明有些过于粗糙，导致某些表达式在C99标准下表现出未定义行为，而这些行为本不应如此。C2011保留了序列点，但引入了一种基于排序和称为“先于 (sequenced before) ”关系的更细致的方法：

如果对同一标量对象的副作用与该标量对象的另一个不同副作用或使用该标量对象值的值计算之间没有排序关系，则行为是未定义的。
如果表达式的子表达式存在多种允许的求值顺序，且在任何一种顺序中发生了此类无序副作用，则行为是未定义的。

(C2011 标准，第6.5节，第2段)

“先于”关系的完整细节过于冗长，无法在此描述，但它们是对序列点的补充而非替代，因此它们的效果是为某些之前行为未定义的求值定义行为。

```
printf("%s", foo); /* undefined behavior */
```

However, the undefined behavior does not always mean that the program crashes — some systems take steps to avoid the crash that normally happens when a null pointer is dereferenced. For example Glibc is known to print

(null)

for the code above. However, add (just) a newline to the format string and you will get a crash:

```
char *foo = 0;
printf("%s\n", foo); /* undefined behavior */
```

In this case, it happens because GCC has an optimization that turns printf("%s\n", argument); into a call to puts with puts(argument), and puts in Glibc does not handle null pointers. All this behavior is standard conforming.

Note that *null pointer* is different from an *empty string*. So, the following is valid and has no undefined behaviour. It'll just print a *newline*:

```
char *foo = "";
printf("%s\n", foo);
```

Section 28.10: Modifying any object more than once between two sequence points

```
int i = 42;
i = i++; /* Assignment changes variable, post-increment as well */
int a = i++ + i--;
```

Code like this often leads to speculations about the "resulting value" of i. Rather than specifying an outcome, however, the C standards specify that evaluating such an expression produces *undefined behavior*. Prior to C2011, the standard formalized these rules in terms of so-called *sequence points*:

Between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.

(C99 standard, section 6.5, paragraph 2)

That scheme proved to be a little too coarse, resulting in some expressions exhibiting undefined behavior with respect to C99 that plausibly should not do. C2011 retains sequence points, but introduces a more nuanced approach to this area based on *sequencing* and a relationship it calls "sequenced before":

If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.

(C2011 standard, section 6.5, paragraph 2)

The full details of the "sequenced before" relation are too long to describe here, but they supplement sequence points rather than supplanting them, so they have the effect of defining behavior for some evaluations whose

特别地，如果两个求值之间存在序列点，则序列点之前的求值“先于”序列点之后的求值。

以下示例具有明确定义的行为：

```
int i = 42;
i = (i++, i+42); /* 逗号运算符创建了一个序列点 */
```

以下示例具有未定义行为：

```
int i = 42;
printf("%d %d", i++, i++); /* 逗号作为函数参数分隔符，而非逗号运算符 */
```

与任何形式的未定义行为一样，观察违反顺序规则的表达式求值的实际行为并无参考价值，除非是事后回顾。语言标准不支持期望此类观察能预测同一程序未来的行为。

第28.11节：重复释放内存

重复释放内存是未定义行为，例如

```
int * x = malloc(sizeof(int));
*x = 9;
free(x);
free(x);
```

摘自标准（C99第7.20.3.2节 free函数）：

否则，如果参数不是先前由calloc、malloc或realloc函数返回的指针，或者该空间已被free或realloc调用释放，则行为未定义。

第28.12节：使用负数或超出类型宽度的位移计数进行位移操作

如果shift count值为负值，则左移和右移操作均未定义¹：

```
int x = 5 << -3; /* 未定义 */
int x = 5 >> -3; /* 未定义 */
```

如果对负值执行左移操作，则结果是未定义的：

```
int x = -5 << 3; /* 未定义 */
```

如果对正值执行左移操作，且数学结果无法用该类型表示，则结果是未定义的¹：

```
/* 假设int是32位宽，值'5 * 2^72'无法
 * 存储在int中。因此，这是未定义的。 */
```

```
int x = 5 << 72;
```

注意，对负值执行右移（例如-5 >> 3）不是未定义，而是实现定义的。

behavior previously was undefined. In particular, if there is a sequence point between two evaluations, then the one before the sequence point is "sequenced before" the one after.

The following example has well-defined behaviour:

```
int i = 42;
i = (i++, i+42); /* The comma-operator creates a sequence point */
```

The following example has undefined behaviour:

```
int i = 42;
printf("%d %d\n", i++, i++); /* commas as separator of function arguments are not comma-operators */
```

As with any form of undefined behavior, observing the actual behavior of evaluating expressions that violate the sequencing rules is not informative, except in a retrospective sense. The language standard provides no basis for expecting such observations to be predictive even of the future behavior of the same program.

Section 28.11: Freeing memory twice

Freeing memory twice is undefined behavior, e.g.

```
int * x = malloc(sizeof(int));
*x = 9;
free(x);
free(x);
```

Quote from standard(7.20.3.2. The free function of C99):

Otherwise, if the argument does not match a pointer earlier returned by the calloc, malloc, or realloc function, or if the space has been deallocated by a call to free or realloc, the behavior is undefined.

Section 28.12: Bit shifting using negative counts or beyond the width of the type

If the shift count value is a **negative value** then both *left shift* and *right shift* operations are undefined¹:

```
int x = 5 << -3; /* undefined */
int x = 5 >> -3; /* undefined */
```

If *left shift* is performed on a **negative value**, it's undefined:

```
int x = -5 << 3; /* undefined */
```

If *left shift* is performed on a **positive value** and result of the mathematical value is **not** representable in the type, it's undefined¹:

```
/* Assuming an int is 32-bits wide, the value '5 * 2^72' doesn't fit
 * in an int. So, this is undefined. */
```

```
int x = 5 << 72;
```

Note that *right shift* on a **negative value** (e.g -5 >> 3) is *not* undefined but *implementation-defined*.

如果右操作数的值为负数或大于等于提升后的左操作数的宽度，则行为是未定义的。

第28.13节：从声明为`_Noreturn`或`noreturn`函数说明符的函数返回

版本 ≥ C11

函数说明符_Noreturn是在C11中引入的。头文件<stdnoreturn.h>提供了一个宏noreturn，它展开为_Noreturn。因此，使用_Noreturn或来自<stdnoreturn.h>的noreturn都是可以的且等效的。

用_Noreturn（或noreturn）声明的函数不允许返回给调用者。如果这样的函数确实返回给调用者，则行为是未定义的。

在下面的示例中，func()被声明为带有noreturn说明符，但它却返回给了调用者。

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void func(void);

void func(void)
{
    printf("In func()..."); /* 由于func()返回，行为未定义 */
}

int main(void)
{
    func();
    return 0;
}
```

gcc和clang会对上述程序产生警告：

```
$ gcc test.c
test.c: 在函数'func'中：
test.c:9:1: 警告：'noreturn'函数确实返回了
}
^

$ clang test.c
test.c:9:1: 警告：声明为 'noreturn' 的函数不应返回 [-Winvalid-noreturn]
}
^
```

一个使用 noreturn 并具有明确定义行为的示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void my_exit(void);
```

If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

Section 28.13: Returning from a function that's declared with `_Noreturn` or `noreturn` function specifier

Version ≥ C11

The function specifier _Noreturn was introduced in C11. The header <stdnoreturn.h> provides a macro noreturn which expands to _Noreturn. So using _Noreturn or noreturn from <stdnoreturn.h> is fine and equivalent.

A function that's declared with _Noreturn (or noreturn) is not allowed to return to its caller. If such a function does return to its caller, the behavior is undefined.

In the following example, func() is declared with noreturn specifier but it returns to its caller.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void func(void);

void func(void)
{
    printf("In func()...\\n");
} /* Undefined behavior as func() returns */

int main(void)
{
    func();
    return 0;
}
```

gcc and clang produce warnings for the above program:

```
$ gcc test.c
test.c: In function 'func':
test.c:9:1: warning: 'noreturn' function does return
}
^

$ clang test.c
test.c:9:1: warning: function declared 'noreturn' should not return [-Winvalid-noreturn]
}
^
```

An example using noreturn that has well-defined behavior:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void my_exit(void);
```

```
/* 调用 exit() 并且不返回调用者。*/
void my_exit(void)
{
    printf("正在退出...");exit(0);
}

int main(void)
{
    my_exit();
    return 0;
}
```

第28.14节：访问超出分配块的内存

指向包含 n 个元素的内存块的指针 a 只有在其指向的地址位于 memory 和 $\text{memory} + (n - 1)$ 范围内时才能被解引用。解引用超出该范围的指针会导致未定义行为。例如，考虑以下代码：

```
int array[3];
int *超出数组 = 数组 + 3;
*超出数组 = 0; /* 访问未分配的内存。*/
```

第三行访问了只有3个元素的数组中的第4个元素，导致未定义行为。
类似地，下面代码片段中第二行的行为也不是很好定义：

```
int 数组[3];
数组[3] = 0;
```

注意，指向数组最后一个元素之后的位置不是未定义行为（ $\text{*超出数组} = \text{数组} + 3$ 在这里是良定义的），但解引用它是未定义行为（ *超出数组 是未定义行为）。该规则同样适用于动态分配的内存（例如通过 `malloc` 创建的缓冲区）。

第28.15节：使用指针修改const变量

```
int main (void)
{
    const int foo_READONLY = 10;
    int *foo_ptr;

    foo_ptr = (int *)&foo_READONLY; /* (1) 这会去除const限定符 */
    *foo_ptr = 20; /* 这是未定义行为 */

    return 0;
}
```

引用 ISO/IEC 9899:201x，第6.7.3节 §2：

如果试图通过使用非const限定类型的左值来修改用const限定类型定义的对象，则行为是未定义的。[...]

(1) 在 GCC 中，这可能会抛出以下警告：warning: 赋值丢弃了指针目标类型中的'const'限定符 [-Wdiscarded-qualifiers]

```
/* calls exit() and doesn't return to its caller. */
void my_exit(void)
{
    printf("Exiting...\\n");
    exit(0);
}

int main(void)
{
    my_exit();
    return 0;
}
```

Section 28.14: Accessing memory beyond allocated chunk

A pointer to a piece of memory containing n elements may only be dereferenced if it is in the range memory and $\text{memory} + (n - 1)$. Dereferencing a pointer outside of that range results in undefined behavior. As an example, consider the following code:

```
int array[3];
int *beyond_array = array + 3;
*beyond_array = 0; /* Accesses memory that has not been allocated. */
```

The third line accesses the 4th element in an array that is only 3 elements long, leading to undefined behavior.
Similarly, the behavior of the second line in the following code fragment is also not well defined:

```
int array[3];
array[3] = 0;
```

Note that pointing past the last element of an array is not undefined behavior ($\text{beyond_array} = \text{array} + 3$ is well defined here), but dereferencing it is (*beyond_array is undefined behavior). This rule also holds for dynamically allocated memory (such as buffers created through `malloc`).

Section 28.15: Modifying a const variable using a pointer

```
int main (void)
{
    const int foo_READONLY = 10;
    int *foo_ptr;

    foo_ptr = (int *)&foo_READONLY; /* (1) This casts away the const qualifier */
    *foo_ptr = 20; /* This is undefined behavior */

    return 0;
}
```

Quoting ISO/IEC 9899:201x, section 6.7.3 §2:

If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. [...]

(1) In GCC this can throw the following warning: warning: assignment discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]

第28.16节：读取未初始化且未被内存支持的对象

版本 ≥ C11

读取对象将导致未定义行为，如果该对象满足以下条件之一：

- 未初始化
- 定义为自动存储期
- 且其地址从未被获取

下面示例中的变量 a 满足所有这些条件：

```
void Function( void )
{
    int a;
    int b = a;
}
```

1 (引用自：ISO:IEC 9899:201X 6.3.2.1 左值、数组和函数指定符 2) 如果左值指定了一个自动存储期的对象，该对象本可以声明为寄存器存储类（其地址从未被获取），且该对象未初始化（未用初始化器声明且在使用前未进行赋值），则行为是未定义的。

第28.17节：指针的加减未正确限定范围

以下代码存在未定义行为：

```
char buffer[6] = "hello";
char *ptr1 = buffer - 1; /* 未定义行为 */
char *ptr2 = buffer + 5; /* 正确，指向数组内的'\0' */
char *ptr3 = buffer + 6; /* 正确，指向数组末尾之后 */
char *ptr4 = buffer + 7; /* 未定义行为 */
```

根据C11标准，如果对指向数组对象内部或刚好超出数组的指针与整数类型进行加法或减法运算，结果指针不指向同一数组对象的内部或刚好超出该数组，则行为未定义（6.5.6）。

此外，解引用指向数组末尾之后的指针自然也是未定义行为：

```
char buffer[6] = "hello";
char *ptr3 = buffer + 6; /* 正确，指向数组末尾之后 */
char value = *ptr3; /* 未定义行为 */
```

第28.18节：解引用空指针

这是一个解引用NULL指针导致未定义行为的示例。

```
int * pointer = NULL;
int value = *pointer; /* 这里发生了解引用 */
```

C标准保证空指针（NULL指针）与任何指向有效对象的指针比较时不相等，且对其解引用会导致未定义行为。

Section 28.16: Reading an uninitialized object that is not backed by memory

Version ≥ C11

Reading an object will cause undefined behavior, if the object is1:

- uninitialized
- defined with automatic storage duration
- its address is never taken

The variable a in the below example satisfies all those conditions:

```
void Function( void )
{
    int a;
    int b = a;
}
```

1 (Quoted from: ISO:IEC 9899:201X 6.3.2.1 Lvalues, arrays, and function designators 2)
If the lvalue designates an object of automatic storage duration that could have been declared with the register storage class (never had its address taken), and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

Section 28.17: Addition or subtraction of pointer not properly bounded

The following code has undefined behavior:

```
char buffer[6] = "hello";
char *ptr1 = buffer - 1; /* undefined behavior */
char *ptr2 = buffer + 5; /* OK, pointing to the '\0' inside the array */
char *ptr3 = buffer + 6; /* OK, pointing to just beyond */
char *ptr4 = buffer + 7; /* undefined behavior */
```

According to C11, if addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object, the behavior is undefined (6.5.6).

Additionally it is naturally undefined behavior to *dereference* a pointer that points to just beyond the array:

```
char buffer[6] = "hello";
char *ptr3 = buffer + 6; /* OK, pointing to just beyond */
char value = *ptr3; /* undefined behavior */
```

Section 28.18: Dereferencing a null pointer

This is an example of dereferencing a NULL pointer, causing undefined behavior.

```
int * pointer = NULL;
int value = *pointer; /* Dereferencing happens here */
```

A NULL pointer is guaranteed by the C standard to compare unequal to any pointer to a valid object, and dereferencing it invokes undefined behavior.

第28.19节：在输入流上使用 fflush

POSIX和C标准明确指出，在输入流上使用 fflush 是未定义行为。fflush 仅定义用于输出流。

```
#include <stdio.h>

int main()
{
    int i;
    char input[4096];

    scanf("%i", &i);
    fflush(stdin); // <- 未定义行为
    gets(input);

    return 0;
}
```

没有标准方法可以丢弃输入流中未读的字符。另一方面，一些实现使用 fflush 来清除 stdin 缓冲区。微软定义了 fflush 在输入流上的行为：如果流是以输入模式打开，fflush 会清除缓冲区内容。根据 POSIX.1-2008，除非输入文件是可定位的，否则 fflush 的行为是未定义的。

详见 [“使用 fflush\(stdin\)”](#) 了解更多细节。

第28.20节：标识符链接不一致

```
extern int var;
static int var; /* 未定义行为 */
```

C11, §6.2.2, 7 规定：

如果在同一翻译单元内，同一标识符同时具有内部链接和外部链接，则行为未定义。

注意，如果标识符的先前声明是可见的，则它将具有先前声明的链接类型。 C11, §6.2.2, 4

允许这样做：

对于在某个作用域中用存储类说明符 extern 声明的标识符，且该标识符的先前声明是可见的，31) 如果先前声明指定了内部或外部链接，则后续声明的标识符的链接与先前声明指定的链接相同。如果没有可见的先前声明，或者先前声明未指定链接，则该标识符具有外部链接。

```
/* 1. 这不是未定义行为 */
static int var;
extern int var;
```

```
/* 2. 这不是未定义行为 */
static int var;
static int var;
```

Section 28.19: Using fflush on an input stream

The POSIX and C standards explicitly state that using `fflush` on an input stream is undefined behavior. The `fflush` is defined only for output streams.

```
#include <stdio.h>

int main()
{
    int i;
    char input[4096];

    scanf("%i", &i);
    fflush(stdin); // <- undefined behavior
    gets(input);

    return 0;
}
```

There is no standard way to discard unread characters from an input stream. On the other hand, some implementations uses `fflush` to clear `stdin` buffer. Microsoft defines the behavior of `fflush` on an input stream: If the stream is open for input, `fflush` clears the contents of the buffer. According to POSIX.1-2008, the behavior of `fflush` is undefined unless the input file is seekable.

See [Using fflush\(stdin\)](#) for many more details.

Section 28.20: Inconsistent linkage of identifiers

```
extern int var;
static int var; /* Undefined behaviour */
```

C11, §6.2.2, 7 says:

If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

Note that if an prior declaration of an identifier is visible then it'll have the prior declaration's linkage. C11, §6.2.2, 4 allows it:

For an identifier declared with the storage-class specifier `extern` in a scope in which a prior declaration of that identifier is visible,31) if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

```
/* 1. This is NOT undefined */
static int var;
extern int var;
```

```
/* 2. This is NOT undefined */
static int var;
static int var;
```

```
/* 3. 这不是未定义 */
extern int var;
extern int var;
```

第28.21节：返回值函数中缺少返回语句

```
int foo(void) {
    /* 执行操作 */
    /* 此处无返回 */
}

int main(void) {
    /* 尝试使用未返回的值会导致未定义行为 */
    int value = foo();
    return 0;
}
```

当函数声明为返回值时，必须在所有可能的代码路径中返回值。一旦调用者（期望返回值）尝试使用返回值，就会发生未定义行为。

注意，未定义行为仅在调用者尝试使用/访问函数返回值时发生。

例如，

```
int foo(void) {
    /* 执行操作 */
    /* 此处无返回 */
}

int main(void) {
    /* foo()未返回的值未被使用。因此，该程序
     * 不会导致*未定义行为*。 */
    foo();
    return 0;
}
```

版本 ≥ C99

主函数（main()）是该规则的一个例外，因为它可以在没有return语句的情况下终止，此时会自动假定返回值为0。

1 (ISO/IEC 9899:201x, 6.9.1/12)

如果执行到终止函数的}，且调用者使用了该函数调用的返回值，则行为是未定义的。

2 (ISO/IEC 9899:201x, 5.1.2.2.3/1)

执行到终止main函数的}时，返回值为0。

第28.22节：除以零

```
int x = 0;
```

```
/* 3. This is NOT undefined */
extern int var;
extern int var;
```

Section 28.21: Missing return statement in value returning function

```
int foo(void) {
    /* do stuff */
    /* no return here */
}

int main(void) {
    /* Trying to use the (not) returned value causes UB */
    int value = foo();
    return 0;
}
```

When a function is declared to return a value then it has to do so on every possible code path through it. Undefined behavior occurs as soon as the caller (which is expecting a return value) tries to use the return value¹.

Note that the undefined behaviour happens *only if* the caller attempts to use/access the value from the function. For example,

```
int foo(void) {
    /* do stuff */
    /* no return here */
}

int main(void) {
    /* The value (not) returned from foo() is unused. So, this program
     * doesn't cause *undefined behaviour*. */
    foo();
    return 0;
}
```

Version ≥ C99

The main() function is an exception to this rule in that it is possible for it to be terminated without a return statement because an assumed return value of 0 will automatically be used in this case².

1 (ISO/IEC 9899:201x, 6.9.1/12)

If the } that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.

2 (ISO/IEC 9899:201x, 5.1.2.2.3/1)

reaching the } that terminates the main function returns a value of 0.

Section 28.22: Division by zero

```
int x = 0;
```

```
int y = 5 / x; /* 整数除法 */
```

或

```
double x = 0.0;  
double y = 5.0 / x; /* 浮点数除法 */
```

或

```
int x = 0;  
int y = 5 % x; /* 取模运算 */
```

对于每个示例中的第二行，当第二个操作数 (x) 的值为零时，行为是未定义的。

请注意，大多数浮点数学的实现会遵循一个标准（例如 IEEE 754），在这种情况下像除以零这样的操作将有一致的结果（例如，INFINITY），尽管 C 标准说该操作是未定义的。

第28.23节：指针类型之间的转换产生未正确对齐的结果

以下情况可能由于指针未正确对齐而具有未定义行为：

```
char *memory_block = calloc(sizeof(uint32_t) + 1, 1);  
uint32_t *intptr = (uint32_t*)(memory_block + 1); /* 可能的未定义行为 */  
uint32_t mvalue = *intptr;
```

未定义行为发生在指针转换时。根据 C11，如果两种指针类型之间的转换产生了未正确对齐的结果（6.3.2.3），则行为是未定义的。这里 uint32_t 可能需要对齐为 2 或 4。

另一方面，calloc 被要求返回一个适合任何对象类型的指针；因此 memory_block 在其初始部分正确对齐以包含一个 uint32_t。然后，在 uint32_t 需要对齐为 2 或 4 的系统上，memory_block + 1 将是一个奇数地址，因此未正确对齐。

请注意，C 标准要求即使是强制类型转换操作本身也是未定义的。这是因为在地址分段的平台上，字节地址 memory_block + 1 甚至可能没有作为整数指针的正确表示。

将 char * 强制转换为其他类型的指针而不考虑对齐要求，有时会被错误地用于解码打包结构，如文件头或网络数据包。

您可以通过使用 memcpy 来避免因指针未对齐转换而产生的未定义行为：

```
memcpy(&mvalue, memory_block + 1, sizeof mvalue);
```

这里没有发生指针转换为 uint32_t*，字节是逐个复制的。

对于我们的示例，这种复制操作只会导致 mvalue 的有效值，因为：

- 我们使用了 calloc，因此字节被正确初始化。在我们的例子中，所有字节的值都是 0，但任何其他适当的初始化也可以。
- uint32_t 是一个精确宽度类型，没有填充位
- 任何任意的位模式对于任何无符号类型都是有效的表示。

```
int y = 5 / x; /* integer division */
```

或

```
double x = 0.0;  
double y = 5.0 / x; /* floating point division */
```

或

```
int x = 0;  
int y = 5 % x; /* modulo operation */
```

For the second line in each example, where the value of the second operand (x) is zero, the behaviour is undefined.

Note that most implementations of floating point math will follow a standard (e.g. IEEE 754), in which case operations like divide-by-zero will have consistent results (e.g., INFINITY) even though the C standard says the operation is undefined.

Section 28.23: Conversion between pointer types produces incorrectly aligned result

The following might have undefined behavior due to incorrect pointer alignment:

```
char *memory_block = calloc(sizeof(uint32_t) + 1, 1);  
uint32_t *intptr = (uint32_t*)(memory_block + 1); /* possible undefined behavior */  
uint32_t mvalue = *intptr;
```

The undefined behavior happens as the pointer is converted. According to C11, if a conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3), the behavior is undefined. Here an uint32_t could require alignment of 2 or 4 for example.

calloc on the other hand is required to return a pointer that is suitably aligned for any object type; thus memory_block is properly aligned to contain an uint32_t in its initial part. Then, on a system where uint32_t has required alignment of 2 or 4, memory_block + 1 will be an odd address and thus not properly aligned.

Observe that the C standard requests that already the cast operation is undefined. This is imposed because on platforms where addresses are segmented, the byte address memory_block + 1 may not even have a proper representation as an integer pointer.

Casting char * to pointers to other types without any concern to alignment requirements is sometimes incorrectly used for decoding packed structures such as file headers or network packets.

You can avoid the undefined behavior arising from misaligned pointer conversion by using memcpy:

```
memcpy(&mvalue, memory_block + 1, sizeof mvalue);
```

Here no pointer conversion to uint32_t* takes place and the bytes are copied one by one.

This copy operation for our example only leads to valid value of mvalue because:

- We used calloc, so the bytes are properly initialized. In our case all bytes have value 0, but any other proper initialization would do.
- uint32_t is an exact width type and has no padding bits
- Any arbitrary bit pattern is a valid representation for any unsigned type.

第28.24节：修改由getenv、strerror和setlocale函数返回的字符串

修改由标准函数`getenv()`、`strerror()`和`setlocale()`返回的字符串是未定义的。因此，实现可能会为这些字符串使用静态存储。

`getenv()`函数，C11, §7.22.4.7, 第4款，规定：

`getenv`函数返回一个指向与匹配列表成员相关联的字符串的指针。程序不得修改该字符串，但后续对`getenv`函数的调用可能会覆盖该字符串。

`strerror()` 函数，C11, §7.23.6.3, 4 规定：

`strerror` 函数返回一个指向字符串的指针，该字符串的内容是特定于区域设置的。指向的数组不得被程序修改，但可能会被后续对 `strerror` 函数的调用覆盖。

`setlocale()` 函数，C11, § 7.11.1.1, 8 规定：

`setlocale` 函数返回的字符串指针是这样的：使用该字符串值及其相关类别进行后续调用时，将恢复程序的该部分区域设置。指向的字符串不得被程序修改，但可能会被后续对 `setlocale` 函数的调用覆盖。

同样，`localeconv()` 函数返回一个指向 `struct lconv` 的指针，该指针指向的内容不得被修改。

`localeconv()` 函数，C11, § 7.11.2.1, 8 规定：

`localeconv` 函数返回一个指向已填充对象的指针。返回值指向的结构体不得被程序修改，但可能会被后续对 `localeconv` 函数的调用覆盖。

Section 28.24: Modifying the string returned by `getenv`, `strerror`, and `setlocale` functions

Modifying the strings returned by the standard functions `getenv()`, `strerror()` and `setlocale()` is undefined. So, implementations may use static storage for these strings.

The `getenv()` function, C11, §7.22.4.7, 4, says:

The `getenv` function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `getenv` function.

The `strerror()` function, C11, §7.23.6.3, 4 says:

The `strerror` function returns a pointer to the string, the contents of which are locale-specific. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `strerror` function.

The `setlocale()` function, C11, §7.11.1.1, 8 says:

The pointer to string returned by the `setlocale` function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `setlocale` function.

Similarly the `localeconv()` function returns a pointer to `struct lconv` which shall not be modified.

The `localeconv()` function, C11, §7.11.2.1, 8 says:

The `localeconv` function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the `localeconv` function.

第29章：随机数生成

第29.1节：基本随机数生成

函数 `rand()` 可用于生成一个介于 0 和 `RAND_MAX`（包括 0 和 `RAND_MAX`）之间的伪随机整数值。

`srand(int)` 用于为伪随机数生成器设定种子。每次使用相同的种子对 `rand()` 进行初始化时，必须产生相同的数值序列。应仅在调用 `rand()` 之前设定一次种子。不应重复设定种子，或每次想生成一批新的伪随机数时重新设定种子。

标准做法是使用 `time(NULL)` 的结果作为种子。如果你的随机数生成器需要有确定性的序列，可以在每次程序启动时用相同的值来给生成器设定种子。通常发布版本的代码不需要这样做，但在调试运行时这样做有助于使错误可复现。

建议始终给生成器设定种子，如果没有设定种子，它的行为就像被设定为 `srand(1)` 一样。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    int i;
    srand(time(NULL));
    i = rand();

    printf("随机值范围在 [0, %d]: %d", RAND_MAX, i);
    return 0;
}
```

可能的输出：

```
随机值范围在 [0, 2147483647]: 823321433
```

注意事项：

C 标准并不保证生成的随机序列的质量。过去，一些 `rand()` 的实现存在分布和随机性方面的严重问题。对于严肃的随机数生成需求，比如密码学，不推荐使用 `rand()`。

第29.2节：置换同余生成器

这是一个独立的随机数生成器，不依赖于 `rand()` 或类似的库函数。

你为什么会想要这样的东西？也许你不信任你平台内置的随机数生成器，或者你想要一个独立于任何特定库实现的可复现的随机源。

这段代码是来自pcg-random.org的PCG32，一种现代、快速、通用的随机数生成器，具有出色的统计特性。它不是密码学安全的，所以不要用于密码学。

```
#include <stdint.h>

/* *真正* 极简的PCG32代码 / (c) 2014 M.E. O'Neill / pcg-random.org
 * 依据Apache许可证2.0授权 (无保证等, 详见网站) */
```

Chapter 29: Random Number Generation

Section 29.1: Basic Random Number Generation

The function `rand()` can be used to generate a pseudo-random integer value between 0 and `RAND_MAX` (0 and `RAND_MAX` included).

`srand(int)` is used to seed the pseudo-random number generator. Each time `rand()` is seeded with the same seed, it must produce the same sequence of values. It should only be seeded once before calling `rand()`. It should not be repeatedly seeded, or reseeded every time you wish to generate a new batch of pseudo-random numbers.

Standard practice is to use the result of `time(NULL)` as a seed. If your random number generator requires to have a deterministic sequence, you can seed the generator with the same value on each program start. This is generally not required for release code, but is useful in debug runs to make bugs reproducible.

It is advised to always seed the generator, if not seeded, it behaves as if it was seeded with `srand(1)`.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    int i;
    srand(time(NULL));
    i = rand();

    printf("Random value between [0, %d]: %d\n", RAND_MAX, i);
    return 0;
}
```

Possible output:

```
Random value between [0, 2147483647]: 823321433
```

Notes:

The C Standard does not guarantee the quality of the random sequence produced. In the past, some implementations of `rand()` had serious issues in distribution and randomness of the generated numbers. **The usage of `rand()` is not recommended for serious random number generation needs, like cryptography.**

Section 29.2: Permuted Congruential Generator

Here's a standalone random number generator that doesn't rely on `rand()` or similar library functions.

Why would you want such a thing? Maybe you don't trust your platform's builtin random number generator, or maybe you want a reproducible source of randomness independent of any particular library implementation.

This code is PCG32 from pcg-random.org, a modern, fast, general-purpose RNG with excellent statistical properties. It's not cryptographically secure, so don't use it for cryptography.

```
#include <stdint.h>

/* *Really* minimal PCG32 code / (c) 2014 M.E. O'Neill / pcg-random.org
 * Licensed under Apache License 2.0 (NO WARRANTY, etc. see website) */
```

```

typedef struct { uint64_t state; uint64_t inc; } pcg32_random_t;

uint32_t pcg32_random_r(pcg32_random_t* rng) {
    uint64_t oldstate = rng->state;
    /* 推进内部状态 */
    rng->state = oldstate * 6364136223846793005ULL + (rng->inc | 1);
    /* 计算输出函数 (XSH RR), 使用旧状态以最大化指令级并行 */
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
}

void pcg32_srandom_r(pcg32_random_t* rng, uint64_t initstate, uint64_t initseq) {
    rng->state = 0U;
    rng->inc = (initseq << 1u) | 1u;
    pcg32_random_r(rng);
    rng->state += initstate;
    pcg32_random_r(rng);
}

```

下面是调用方法：

```

#include <stdio.h>
int main(void) {
pcg32_random_t rng; /* RNG 状态 */
    int i;

    /* 初始化随机数生成器种子 */
    pcg32_srandom_r(&rng, 42u, 54u);

    /* 打印一些随机的32位整数 */
    for (i = 0; i < 6; i++) printf("0x%08x", pcg32_random_r(&rng));
    return 0;
}

```

第29.3节：Xorshift生成

一种对有缺陷的 `rand()` 过程的良好且简单的替代方法是 xorshift，这是一类由 乔治·马萨利亚 (George Marsaglia) 发现的伪随机数生成器。xorshift生成器是最快的非加密安全随机数生成器之一。更多信息和其他示例实现可在 xorshift 维基百科页面找到。

示例实现

```

#include <stdint.h>

/* 这些状态变量必须初始化，确保它们不全为零。*/
uint32_t w, x, y, z;

uint32_t xorshift128(void)
{
    uint32_t t = x;
    t ^= t << 11U;
    t ^= t >> 8U;
    x = y; y = z; z = w;
    w ^= w >> 19U;
    w ^= t;
}

```

```

typedef struct { uint64_t state; uint64_t inc; } pcg32_random_t;

uint32_t pcg32_random_r(pcg32_random_t* rng) {
    uint64_t oldstate = rng->state;
    /* Advance internal state */
    rng->state = oldstate * 6364136223846793005ULL + (rng->inc | 1);
    /* Calculate output function (XSH RR), uses old state for max ILP */
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
}

void pcg32_srandom_r(pcg32_random_t* rng, uint64_t initstate, uint64_t initseq) {
    rng->state = 0U;
    rng->inc = (initseq << 1u) | 1u;
    pcg32_random_r(rng);
    rng->state += initstate;
    pcg32_random_r(rng);
}

```

And here's how to call it:

```

#include <stdio.h>
int main(void) {
    pcg32_random_t rng; /* RNG state */
    int i;

    /* Seed the RNG */
    pcg32_srandom_r(&rng, 42u, 54u);

    /* Print some random 32-bit integers */
    for (i = 0; i < 6; i++)
        printf("0x%08x\n", pcg32_random_r(&rng));

    return 0;
}

```

Section 29.3: Xorshift Generation

A good and easy alternative to the flawed `rand()` procedures, is *xorshift*, a class of pseudo-random number generators discovered by [George Marsaglia](#). The xorshift generator is among the fastest non-cryptographically-secure random number generators. More information and other example implementations are available on the [xorshift Wikipedia page](#)

Example implementation

```

#include <stdint.h>

/* These state variables must be initialised so that they are not all zero. */
uint32_t w, x, y, z;

uint32_t xorshift128(void)
{
    uint32_t t = x;
    t ^= t << 11U;
    t ^= t >> 8U;
    x = y; y = z; z = w;
    w ^= w >> 19U;
    w ^= t;
}

```

```
    return w;  
}
```

第29.4节：限制生成到给定范围内

通常在生成随机数时，生成一个范围内的整数或介于0.0和1.0之间的p值是有用的。虽然可以使用取模运算将种子缩减到较小的整数，但这会使用低位，而低位通常经过一个短周期，如果N相对于RAND_MAX较大，会导致分布略微偏斜。

宏

```
#define uniform() (rand() / (RAND_MAX + 1.0))
```

生成一个在0.0到1.0 - epsilon之间的p值，因此

```
i = (int)(uniform() * N)
```

将把i设置为范围在0到N - 1之间的均匀随机数。

不幸的是，这里存在一个技术缺陷，即RAND_MAX允许大于类型为double能够准确表示的数值。这意味着RAND_MAX + 1.0的计算结果等于RAND_MAX，函数偶尔会返回1。不过这种情况不太可能发生。

```
    return w;  
}
```

Section 29.4: Restrict generation to a given range

Usually when generating random numbers it is useful to generate integers within a range, or a p value between 0.0 and 1.0. Whilst modulus operation can be used to reduce the seed to a low integer this uses the low bits, which often go through a short cycle, resulting in a slight skewing of distribution if N is large in proportion to RAND_MAX.

The macro

```
#define uniform() (rand() / (RAND_MAX + 1.0))
```

produces a p value on 0.0 to 1.0 - epsilon, so

```
i = (int)(uniform() * N)
```

will set i to a uniform random number within the range 0 to N - 1.

Unfortunately there is a technical flaw, in that RAND_MAX is permitted to be larger than a variable of type `double` can accurately represent. This means that `RAND_MAX + 1.0` evaluates to RAND_MAX and the function occasionally returns unity. This is unlikely however.

第30章：预处理器和宏

所有预处理命令都以井号 (#) 符号开头。C宏只是使用#define预处理指令定义的预处理命令。在预处理阶段，C预处理器（C编译器的一部分）会简单地将宏的主体替换到其名称出现的地方。

第30.1节：头文件包含保护

几乎每个头文件都应该遵循include guard惯用法：

my-header-file.h

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H

// 头文件的代码主体

#endif
```

这确保了当你在多个地方#include "my-header-file.h"时，不会出现函数、变量等的重复声明。
想象以下文件层次结构：

header-1.h

```
typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);
```

header-2.h

```
#include "header-1.h"

int myFunction2(MyStruct *value);
```

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // 做一些事情
}
```

这段代码有一个严重的问题：MyStruct 的详细内容被定义了两次，这是不允许的。这会导致一个编译错误，且由于一个头文件包含另一个头文件，错误可能难以追踪。如果你改用头文件保护宏：

header-1.h

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
```

Chapter 30: Preprocessor and Macros

All preprocessor commands begins with the hash (pound) symbol #. A C macro is just a preprocessor command that is defined using the #define preprocessor directive. During the preprocessing stage, the C preprocessor (a part of the C compiler) simply substitutes the body of the macro wherever its name appears.

Section 30.1: Header Include Guards

Pretty much every header file should follow the [include guard](#) idiom:

my-header-file.h

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H

// Code body for header file

#endif
```

This ensures that when you #include "my-header-file.h" in multiple places, you don't get duplicate declarations of functions, variables, etc. Imagine the following hierarchy of files:

header-1.h

```
typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);
```

header-2.h

```
#include "header-1.h"

int myFunction2(MyStruct *value);
```

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

This code has a serious problem: the detailed contents of MyStruct is defined twice, which is not allowed. This would result in a compilation error that can be difficult to track down, since one header file includes another. If you instead did it with header guards:

header-1.h

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
```

```

...
} MyStruct;

int myFunction(MyStruct *value);

#endif

```

header-2.h

```

#ifndef HEADER_2_H
#define HEADER_2_H

#include "header-1.h"

int myFunction2(MyStruct *value);

#endif

```

main.c

```

#include "header-1.h"
#include "header-2.h"

int main() {
    // 做一些事情
}

```

这将展开为：

```

#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);

#endif

#ifndef HEADER_2_H
#define HEADER_2_H

#ifndef HEADER_1_H // 安全，因为 HEADER_1_H 已经被 #define 了。
#define HEADER_1_H

typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);

#endif

int myFunction2(MyStruct *value);

#endif

int main() {
    // 做一些事情
}

```

```

...
} MyStruct;

int myFunction(MyStruct *value);

#endif

```

header-2.h

```

#ifndef HEADER_2_H
#define HEADER_2_H

#include "header-1.h"

int myFunction2(MyStruct *value);

#endif

```

main.c

```

#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}

```

This would then expand to:

```

#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);

#endif

#ifndef HEADER_2_H
#define HEADER_2_H

#ifndef HEADER_1_H // Safe, since HEADER_1_H was #define'd before.
#define HEADER_1_H

typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);

#endif

int myFunction2(MyStruct *value);

#endif

int main() {
    // do something
}

```

```
}
```

当编译器遇到第二次包含 **header-1.h** 时，**HEADER_1_H** 已经被之前的包含定义了。因此，归结为以下内容：

```
#define HEADER_1_H

typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);

#define HEADER_2_H

int myFunction2(MyStruct *value);

int main() {
    // 做一些事情
}
```

因此不会有编译错误。

注意：命名头文件保护符有多种不同的约定。有些人喜欢命名为 **HEADER_2_H**，有些会包含项目名称，如**MY_PROJECT_HEADER_2_H**。重要的是确保你遵循的约定能使项目中的每个文件都有唯一的头文件保护符。

如果结构体细节未包含在头文件中，声明的类型将是不完整的或称为“不透明类型”。这种类型很有用，可以隐藏函数用户的实现细节。对于许多用途，标准C库中的FILE类型可以被视为不透明类型（尽管它通常不是不透明的，以便标准I/O函数的宏实现可以利用结构体的内部）。在这种情况下，头文件-1.h 可以包含：

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct MyStruct MyStruct;

int myFunction(MyStruct *value);

#endif
```

请注意，结构体必须有一个标签名（这里是**MyStruct**——它属于标签命名空间，与**typedef**名**MyStruct**的普通标识符命名空间是分开的），且省略了{ ... }。这表示“存在一个结构体类型**struct MyStruct**，并且有一个别名**MyStruct**”。

在实现文件中，可以定义结构体的详细内容以使类型完整：

```
struct MyStruct {
    ...
};
```

如果你使用的是C11，可以重复声明**typedef struct MyStruct MyStruct;**而不会导致编译错误，但早期版本的C会报错。因此，最好还是使用包含保护

```
}
```

When the compiler reaches the second inclusion of **header-1.h**, **HEADER_1_H** was already defined by the previous inclusion. Ergo, it boils down to the following:

```
#define HEADER_1_H

typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);

#define HEADER_2_H

int myFunction2(MyStruct *value);

int main() {
    // do something
}
```

And thus there is no compilation error.

Note: There are multiple different conventions for naming the header guards. Some people like to name it **HEADER_2_H**, some include the project name like **MY_PROJECT_HEADER_2_H**. The important thing is to ensure that the convention you follow makes it so that each file in your project has a unique header guard.

If the structure details were not included in the header, the type declared would be incomplete or an [opaque type](#). Such types can be useful, hiding implementation details from users of the functions. For many purposes, the FILE type in the standard C library can be regarded as an opaque type (though it usually isn't opaque so that macro implementations of the standard I/O functions can make use of the internals of the structure). In that case, the **header-1.h** could contain:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct MyStruct MyStruct;

int myFunction(MyStruct *value);

#endif
```

Note that the structure must have a tag name (here **MyStruct** — that's in the tags namespace, separate from the ordinary identifiers namespace of the **typedef** name **MyStruct**), and that the { ... } is omitted. This says "there is a structure type **struct MyStruct** and there is an alias for it **MyStruct**".

In the implementation file, the details of the structure can be defined to make the type complete:

```
struct MyStruct {
    ...
};
```

If you are using C11, you could repeat the **typedef struct MyStruct MyStruct;** declaration without causing a compilation error, but earlier versions of C would complain. Consequently, it is still best to use the include guard

惯用法，尽管在本例中，如果代码仅用支持C11的编译器编译，则该保护是可选的。

许多编译器支持`#pragma once`指令，其效果相同：

my-header-file.h

```
#pragma once  
// 头文件代码
```

但是，`#pragma once`不是C标准的一部分，因此使用它会降低代码的可移植性。

有些头文件没有使用包含保护惯用法。一个具体的例子是标准`<assert.h>`头文件。它可能在单个翻译单元中被多次包含，且这样做的效果取决于宏是否存在。每次包含该头文件时都会定义`NDEBUG`。你偶尔可能有类似的需求；这种情况会很少见。通常，你的头文件应该使用包含保护惯用法来保护。

第30.2节：使用`#if 0`来屏蔽代码段

如果有些代码段你考虑删除或想暂时禁用，可以用块注释将其注释掉。

```
/* 用块注释包围整个函数，防止其被使用。  
* 这个函数的目的到底是什么？  
int myUnusedFunction(void)  
{  
    int i = 5;  
    return i;  
}
```

但是，如果你用块注释包围的源代码中本身包含块注释，现有块注释的结束符`*/`可能会导致你新加的块注释无效，从而引发编译问题。

```
/* 用块注释包围整个函数，防止其被使用。  
* 这个函数的目的到底是什么？  
int myUnusedFunction(void)  
{  
    int i = 5;  
  
    /* 返回 5 */  
    return i;  
}
```

在前面的例子中，函数的最后两行和最后的`*/`会被编译器看到，因此会编译出错。一个更安全的方法是在你想屏蔽的代码周围使用`#if 0`指令。

```
#if 0  
/* #if 0 计算结果为假，因此这里和 #endif 之间的所有内容都会被  
* 预处理器移除。 */  
int myUnusedFunction(void)  
{
```

idiom，even though in this example, it would be optional if the code was only ever compiled with compilers that supported C11.

Many compilers support the `#pragma once` directive, which has the same results:

my-header-file.h

```
#pragma once  
// Code for header file
```

However, `#pragma once` is not part of the C standard, so the code is less portable if you use it.

A few headers do not use the include guard idiom. One specific example is the standard `<assert.h>` header. It may be included multiple times in a single translation unit, and the effect of doing so depends on whether the macro `NDEBUG` is defined each time the header is included. You may occasionally have an analogous requirement; such cases will be few and far between. Ordinarily, your headers should be protected by the include guard idiom.

Section 30.2: #if 0 to block out code sections

If there are sections of code that you are considering removing or want to temporarily disable, you can comment it out with a block comment.

```
/* Block comment around whole function to keep it from getting used.  
* What's even the purpose of this function?  
int myUnusedFunction(void)  
{  
    int i = 5;  
    return i;  
}
```

However, if the source code you have surrounded with a block comment has block style comments in the source, the ending`*/` of the existing block comments can cause your new block comment to be invalid and cause compilation problems.

```
/* Block comment around whole function to keep it from getting used.  
* What's even the purpose of this function?  
int myUnusedFunction(void)  
{  
    int i = 5;  
  
    /* Return 5 */  
    return i;  
}
```

In the previous example, the last two lines of the function and the last`*/` are seen by the compiler, so it would compile with errors. A safer method is to use an`#if 0` directive around the code you want to block out.

```
#if 0  
/* #if 0 evaluates to false, so everything between here and the #endif are  
* removed by the preprocessor. */  
int myUnusedFunction(void)  
{
```

```

int i = 5;
return i;
}
#endif

```

这样做的一个好处是，当你想回头查找代码时，搜索 "#if 0" 比搜索所有注释要容易得多。

另一个非常重要的好处是，你可以使用#if 0嵌套注释代码。这是注释无法做到的。

使用#if 0的另一种替代方法是使用一个不会被#define但更能描述为什么代码被屏蔽的名称。例如，如果有一个看似无用的死代码函数，你可以使用#if defined(POSSIBLE_DEAD_CODE)或者#if defined(FUTURE_CODE_REL_020201)来表示该代码是在其他功能到位后需要的代码或类似情况。这样，当回头删除或启用这些代码时，这些代码段很容易找到。

```

int i = 5;
return i;
}
#endif

```

A benefit with this is that when you want to go back and find the code, it's much easier to do a search for "#if 0" than searching all your comments.

Another very important benefit is that you can nest commenting out code with `#if 0`. This cannot be done with comments.

An alternative to using `#if 0` is to use a name that will not be `#defined` but is more descriptive of why the code is being blocked out. For instance if there is a function that seems to be useless dead code you might use `#if defined(POSSIBLE_DEAD_CODE)` or `#if defined(FUTURE_CODE_REL_020201)` for code needed once other functionality is in place or something similar. Then when going back through to remove or enable that source, those sections of source are easy to find.

第30.3节：类函数宏

类函数宏类似于inline函数，在某些情况下很有用，比如临时调试日志：

```

#ifndef DEBUG
#define LOGFILENAME "/tmp/logfile.log" # defi

ne LOG(str) do {
    FILE *fp = fo
pen(LOGFILENAME, "a");      \ if (fp) {
    \   fprintf(fp, "%s:%d %s", __FILE__,
__LINE__, \ /* 不打印空指针 */ \
    str ?str :<null>);      \   fclose(fp);
    \ }
    \ else {
    \   perror("打开 '" LOGFILENAME "' 失败"); \ }
} while (0)

#else
/* 如果未定义 DEBUG，则使其成为无操作。 */
#define LOG(LINE) (void)0
#endif

```

```

#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc > 1)
LOG("有命令行参数");
    else
LOG("无命令行参数");
    return 0;
}

```

这里在两种情况下（有DEBUG或无DEBUG）调用的行为都与返回类型为void的函数相同。这确保了if/else条件语句按预期被解释。

在DEBUG情况下，这是通过do { ... } while(0)结构实现的。在另一种情况下，(void)0是一个无副作用的语句，直接被忽略。

```

#ifndef DEBUG
#define LOGFILENAME "/tmp/logfile.log"

#define LOG(str) do { \
    FILE *fp = fopen(LOGFILENAME, "a"); \
    if (fp) { \
        fprintf(fp, "%s:%d %s\n", __FILE__, __LINE__, \
            /* don't print null pointer */ \
            str ?str :<null>); \
        fclose(fp); \
    } \
    else { \
        perror("Opening '" LOGFILENAME "' failed"); \
    } \
} while (0)
#else
/* Make it a NOOP if DEBUG is not defined. */
#define LOG(LINE) (void)0
#endif

```

```

#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc > 1)
LOG("There are command line arguments");
    else
LOG("No command line arguments");
    return 0;
}

```

Here in both cases (with DEBUG or not) the call behaves the same way as a function with `void` return type. This ensures that the `if/else` conditionals are interpreted as expected.

In the DEBUG case this is implemented through a `do { ... } while(0)` construct. In the other case, `(void)0` is a statement with no side effect that is just ignored.

后一种情况的另一种写法是

```
#define LOG(LINE) do { /* empty */ } while (0)
```

使其在所有情况下在语法上等同于第一个。

如果你使用GCC，还可以实现一个类似函数的宏，使用非标准的GNU扩展—[语句表达式](#)来返回结果。例如：

```
#include <stdio.h>

#define POW(X, Y) \
({ \
int i, r = 1; \
for (i = 0; i < Y; ++i) \
    r *= X; \
r; \ // 返回值是最后一个操作的结果
})

int main(void)
{
    int result;

result = POW(2, 3);printf("Result:
    %d", result);}
```

第30.4节：源文件包含

#include预处理指令最常见的用法如下：

```
#include <stdio.h>
#include "myheader.h"
```

#include 用文件中引用的内容替换该语句。尖括号 (<>) 表示系统中安装的头文件，双引号 ("") 表示用户提供的文件。

宏本身可以展开其他宏一次，正如以下示例所示：

```
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h"
/* 依此类推 */
#else
#define INCFILE "versN.h"
#endif
/* ... */
#include INCFILE
```

第30.5节：条件包含和条件函数签名修改

为了有条件地包含一段代码，预处理器提供了多个指令（例如 #if、#ifdef、#else、#endif 等）。

```
/* 定义一个条件 `printf` 宏，只有在定义了 `DEBUG` 时才打印
```

An alternative for the latter would be

```
#define LOG(LINE) do { /* empty */ } while (0)
```

such that it is in all cases syntactically equivalent to the first.

If you use GCC, you can also implement a function-like macro that returns result using a non-standard GNU extension — [statement expressions](#). For example:

```
#include <stdio.h>

#define POW(X, Y) \
({ \
int i, r = 1; \
for (i = 0; i < Y; ++i) \
    r *= X; \
r; \ // returned value is result of last operation
})

int main(void)
{
    int result;

result = POW(2, 3);
printf("Result: %d\n", result);}
```

Section 30.4: Source file inclusion

The most common uses of `#include` preprocessing directives are as in the following:

```
#include <stdio.h>
#include "myheader.h"
```

`#include` replaces the statement with the contents of the file referred to. Angle brackets (<>) refer to header files installed on the system, while quotation marks ("") are for user-supplied files.

Macros themselves can expand other macros once, as this example illustrates:

```
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h"
/* and so on */
#else
#define INCFILE "versN.h"
#endif
/* ... */
#include INCFILE
```

Section 30.5: Conditional inclusion and conditional function signature modification

To conditionally include a block of code, the preprocessor has several directives (e.g. `#if`, `#ifdef`, `#else`, `#endif`, etc).

```
/* Defines a conditional `printf` macro, which only prints if `DEBUG`
```

```
* */

#ifndef DEBUG
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

普通的C关系运算符可以用于#if条件

```
#if __STDC_VERSION__ >= 201112L
/* 对于C11或更高版本执行操作 */
#elif __STDC_VERSION__ >= 199901L
/* 对于C99执行操作 */
#else
/* 对于C99之前的版本执行操作 */
#endif
```

#if指令的行为类似于C语言的if语句，它只能包含整型常量表达式，且不允许有类型转换。它支持一个额外的一元运算符defined(标识符)，如果该标识符已定义，则返回1，否则返回0。

```
#if defined(DEBUG) && !defined QUIET
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

条件函数签名修改

在大多数情况下，应用程序的发布版本预计具有尽可能少的开销。然而，在测试中间版本时，关于发现的问题的额外日志和信息可能会很有帮助。

例如，假设有一个函数SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)，在进行测试版本时希望它能生成关于其使用的日志。然而该函数在多个地方被使用，并且希望在生成日志时，部分信息能显示该函数是从哪里被调用的。

因此，使用条件编译，你可以在声明该函数的包含文件中使用如下方式。
这将用调试版本的函数替换标准版本的函数。预处理器用于将对函数SerOpPluAllRead()的调用替换为对函数SerOpPluAllRead_Debug()的调用，后者带有两个额外的参数，即函数被使用的文件名和行号。

条件编译用于选择是否用调试版本覆盖标准函数。

```
#if 0
// 我们调试版本函数的声明和原型。
SHORT SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo);

// 宏定义，用于用带有额外参数的调试函数替换使用旧名称的函数调用。

#define SerOpPluAllRead(pPif, usLock) SerOpPluAllRead_Debug(pPif, usLock, __FILE__, __LINE__)
#else
// 通常用于构建的标准函数声明。
SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd);
#endif
```

```
* has been defined
*/
#ifndef DEBUG
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Normal C relational operators may be used for the #if condition

```
#if __STDC_VERSION__ >= 201112L
/* Do stuff for C11 or higher */
#elif __STDC_VERSION__ >= 199901L
/* Do stuff for C99 */
#else
/* Do stuff for pre C99 */
#endif
```

The #if directives behaves similar to the C if statement, it shall only contain integral constant expressions, and no casts. It supports one additional unary operator, defined(identifier), which returns 1 if the identifier is defined, and 0 otherwise.

```
#if defined(DEBUG) && !defined QUIET
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Conditional Function Signature Modification

In most cases a release build of an application is expected to have as little overhead as possible. However during testing of an interim build, additional logs and information about problems found can be helpful.

For example assume there is some function SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd) which when doing a test build it is desired will generate a log about its use. However this function is used in multiple places and it is desired that when generating the log, part of the information is to know where the function being called from.

So using conditional compilation you can have something like the following in the include file declaring the function. This replaces the standard version of the function with a debug version of the function. The preprocessor is used to replace calls to the function SerOpPluAllRead() with calls to the function SerOpPluAllRead_Debug() with two additional arguments, the name of the file and the line number of where the function is used.

Conditional compilation is used to choose whether to override the standard function with a debug version or not.

```
#if 0
// function declaration and prototype for our debug version of the function.
SHORT SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo);

// macro definition to replace function call using old name with debug function with additional arguments.
#define SerOpPluAllRead(pPif, usLock) SerOpPluAllRead_Debug(pPif, usLock, __FILE__, __LINE__)
#else
// standard function declaration that is normally used with builds.
SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd);
#endif
```

这允许您重写函数SerOpPluAllRead()的标准版本，使用一个版本来提供调用该函数的文件名和文件中的行号。

有一个重要的注意事项：任何使用此函数的文件必须包含使用此方法的头文件，以便预处理器能够修改该函数。否则您将看到链接错误。

函数的定义大致如下。该源码的作用是请求预处理器将函数SerOpPluAllRead()重命名为SerOpPluAllRead_Debug()，并修改参数列表，增加两个额外参数，即指向调用该函数的文件名的指针和函数使用所在文件的行号。

```
#if defined(SerOpPluAllRead)
// 前向声明替换函数，我们将在创建日志后调用该函数。
SHORT SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd);

SHORT SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo)
{
    int iLen = 0;
    char xBuffer[256];

    // 仅打印文件名的最后30个字符以缩短日志。
    iLen = strlen(aszFilePath);
    if (iLen > 30) {
        iLen = iLen - 30;
    }
    else {
        iLen = 0;
    }

    sprintf(xBuffer, "SerOpPluAllRead_Debug(): husHandle = %d, 文件 %s, 行号 = %d",
    pPif->husHandle, aszFilePath + iLen, nLineNo);
    IssueDebugLog(xBuffer);

    // 既然我们已经发出了日志，继续进行标准处理。
    return SerOpPluAllRead_Special(pPif, usLockHnd);
}

// 当我们生成日志时使用的特殊替代函数名。
SHORT SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd)
#else
// 标准的、正常的函数名（签名），被我们的调试版本替代。
SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)
#endif
{
    if (STUB_SELF == SstReadAsMaster()) {
        return OpPluAllRead(pPif, usLockHnd);
    }
    return OP_NOT_MASTER;
}
```

第30.6节：_cplusplus 用于在C++代码中使用C外部函数 用C++编译 - 名称修饰

有时，由于语言差异，包含文件必须根据编译器是C编译器还是C++编译器，生成不同的预处理器输出。

例如，一个函数或其他外部符号定义在C源文件中，但在C++源文件中使用。由于C++使用名称修饰（或名称装饰）来根据函数参数类型生成唯一的函数名，

This allows you to override the standard version of the function SerOpPluAllRead() with a version that will provide the name of the file and line number in the file of where the function is called.

There is one important consideration: any file using this function must include the header file where this approach is used in order for the preprocessor to modify the function. Otherwise you will see a linker error.

The definition of the function would look something like the following. What this source does is to request that the preprocessor rename the function SerOpPluAllRead() to be SerOpPluAllRead_Debug() and to modify the argument list to include two additional arguments, a pointer to the name of the file where the function was called and the line number in the file at which the function is used.

```
#if defined(SerOpPluAllRead)
// forward declare the replacement function which we will call once we create our log.
SHORT SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd);

SHORT SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo)
{
    int iLen = 0;
    char xBuffer[256];

    // only print the last 30 characters of the file name to shorten the logs.
    iLen = strlen(aszFilePath);
    if (iLen > 30) {
        iLen = iLen - 30;
    }
    else {
        iLen = 0;
    }

    sprintf(xBuffer, "SerOpPluAllRead_Debug(): husHandle = %d, File %s, lineno = %d",
    pPif->husHandle, aszFilePath + iLen, nLineNo);
    IssueDebugLog(xBuffer);

    // now that we have issued the log, continue with standard processing.
    return SerOpPluAllRead_Special(pPif, usLockHnd);
}

// our special replacement function name for when we are generating logs.
SHORT SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd)
#else
// standard, normal function name (signature) that is replaced with our debug version.
SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)
#endif
{
    if (STUB_SELF == SstReadAsMaster()) {
        return OpPluAllRead(pPif, usLockHnd);
    }
    return OP_NOT_MASTER;
}
```

Section 30.6: _cplusplus for using C externals in C++ code compiled with C++ - name mangling

There are times when an include file has to generate different output from the preprocessor depending on whether the compiler is a C compiler or a C++ compiler due to language differences.

For example a function or other external is defined in a C source file but is used in a C++ source file. Since C++ uses name mangling (or name decoration) in order to generate unique function names based on function argument

在C++源文件中使用的C函数声明会导致链接错误。C++编译器会根据C++的名称修饰规则修改指定的外部名称用于编译器输出。结果是在将C++编译器输出与C编译器输出链接时，由于找不到外部符号而产生链接错误。

由于C编译器不进行名称修饰，而C++编译器对所有外部标签（函数名或变量名）都会进行名称修饰，预定义的预处理器宏`_cplusplus`被引入以实现编译器检测。

为了解决C和C++之间外部名称编译器输出不兼容的问题，宏`_cplusplus`在C++预处理器中被定义，而在C预处理器中未定义。该宏名可以与条件预处理指令`#ifdef`或`#if`结合`defined()`操作符使用，以判断源代码或包含文件是作为C++还是C进行编译。

```
#ifdef __cplusplus  
tf("C++");#else  
  
printf("C");#endif
```

或者你可以使用

```
#if defined(__cplusplus)printf(  
"C++");  
#else  
printf("C");#endif
```

为了指定从用C编译器编译的C源文件中使用的函数在C++源文件中的正确函数名，可以检查`_cplusplus`定义的常量，从而使用`extern "C" { /* ... */ };` 用于在头文件被包含在C++源文件中时声明C的外部变量。
但是当用C编译器编译时，`extern "C" { /* ... */ };` 不会被使用。这个条件编译是必要的，因为 `extern "C" { /* ... */ };` 在C++中是有效的，但在C中无效。

```
#ifdef __cplusplus  
// 如果我们是用C++编译器编译的，那么将  
// 以下函数声明为C函数以防止名称改编 (name mangling) 。  
extern "C" {  
#endif  
  
// 导出的C函数列表。  
int foo (void);  
  
#ifdef __cplusplus  
// 如果这是C++编译器，我们需要关闭extern声明。  
};  
#endif
```

第30.7节：标记粘贴

标记粘贴允许将两个宏参数连接在一起。例如，`front##back` 产生 `frontback`。一个著名的例子是Win32的 `<TCHAR.H>` 头文件。在标准C中，可以写成 `L"string"` 来声明宽字符字符串。然而，Windows API允许通过简单地 `#define`ing `UNICODE` 来实现宽字符字符串和窄字符字符串之间的转换。为了实现字符串字面量，`TCHAR.H` 使用了这个

```
#ifdef UNICODE  
#define TEXT(x) L##x
```

types, a C function declaration used in a C++ source file will cause link errors. The C++ compiler will modify the specified external name for the compiler output using the name mangling rules for C++. The result is link errors due to externals not found when the C++ compiler output is linked with the C compiler output.

Since C compilers do not do name mangling but C++ compilers do for all external labels (function names or variable names) generated by the C++ compiler, a predefined preprocessor macro, `_cplusplus`, was introduced to allow for compiler detection.

In order to work around this problem of incompatible compiler output for external names between C and C++, the macro `_cplusplus` is defined in the C++ Preprocessor and is not defined in the C Preprocessor. This macro name can be used with the conditional preprocessor `#ifdef` directive or `#if` with the `defined()` operator to tell whether a source code or include file is being compiled as C++ or C.

```
#ifdef __cplusplus  
printf("C++\n");  
#else  
printf("C\n");  
#endif
```

Or you could use

```
#if defined(__cplusplus)  
printf("C++\n");  
#else  
printf("C\n");  
#endif
```

In order to specify the correct function name of a function from a C source file compiled with the C compiler that is being used in a C++ source file you could check for the `_cplusplus` defined constant in order to cause the `extern "C" { /* ... */ };` to be used to declare C externals when the header file is included in a C++ source file. However when compiled with a C compiler, the `extern "C" { /* ... */ };` is not used. This conditional compilation is needed because `extern "C" { /* ... */ };` is valid in C++ but not in C.

```
#ifdef __cplusplus  
// if we are being compiled with a C++ compiler then declare the  
// following functions as C functions to prevent name mangling.  
extern "C" {  
#endif  
  
// exported C function list.  
int foo (void);  
  
#ifdef __cplusplus  
// if this is a C++ compiler, we need to close off the extern declaration.  
};  
#endif
```

Section 30.7: Token pasting

Token pasting allows one to glue together two macro arguments. For example, `front##back` yields `frontback`. A famous example is Win32's `<TCHAR.H>` header. In standard C, one can write `L"string"` to declare a wide character string. However, Windows API allows one to convert between wide character strings and narrow character strings simply by `#define`ing `UNICODE`. In order to implement the string literals, `TCHAR.H` uses this

```
#ifdef UNICODE  
#define TEXT(x) L##x
```

每当用户编写TEXT("hello, world")，且定义了UNICODE时，C预处理器会将L与宏参数连接起来。将L与"hello, world"连接后得到L"hello, world"。

第30.8节：预定义宏

预定义宏是指C预处理器已经理解的宏，程序无需定义即可使用。示例包括

强制预定义宏

- `_FILE_`, 表示当前源文件的文件名（字符串字面量）,
- `_LINE_`表示当前行号（整数常量）,
- `_DATE_`表示编译日期（字符串字面量）,
- `_TIME_`表示编译时间（字符串字面量）。

还有一个相关的预定义标识符`_func_` (ISO/IEC 9899:2011§6.4.2.2) , 它不是宏：

标识符`_func_`应由翻译器隐式声明，仿佛在每个函数定义的开括号之后立即声明了：

```
static const char _func_[] = "function-name";
```

出现，其中`function-name`是词法上封闭函数的名称。

`_FILE_`、`_LINE_` 和 `_func_` 对调试特别有用。例如：

```
fprintf(stderr, "%s: %s: %d: 分母为0", _FILE_, _func_, _LINE_);
```

在 C99 之前的编译器中，可能支持也可能不支持 `_func_`，或者有一个功能相同但名称不同的宏。例如，gcc 在 C89 模式下使用 `_FUNCTION_`。

以下宏允许查询实现的详细信息：

- `_STDC_VERSION_` 实现的 C 标准版本。该常量整数采用 `yyyymmL` 格式 (C11 的值为 `201112L`, C99 的值为 `199901L` ; C89/C90 未定义该宏)
- `_STDC_HOSTED_` 如果是托管实现，则为 1，否则为 0。
- `_STDC_` 如果为 1，则表示实现符合 C 标准。

其他预定义宏（非强制）

ISO/IEC 9899:2011 § 6.10.9.2 环境宏：

- `_STDC_ISO_10646_` 形式为 `yyyymmL` 的整数常量（例如 `199712L`）。如果定义了该符号，则存储在 `wchar_t` 类型对象中的 Unicode 必需字符集中的每个字符，其值与该字符的短标识符相同。Unicode 必需字符集包括 ISO/IEC 10646 定义的所有字符，以及截至指定年月的所有修订和技术勘误。如果使用其他编码，则该宏不应被定义，实际使用的编码由实现决定。
- `_STDC_MB_MIGHT_NEQ_WC_` 整数常量1，表示在`wchar_t`的编码中，基本字符集的成员不必具有与其作为单个字符的整数字符常量时相等的代码值。

Whenever a user writes `TEXT("hello, world")`, and `UNICODE` is defined, the C preprocessor concatenates L and the macro argument. L concatenated with "hello, world" gives L"hello, world".

Section 30.8: Predefined Macros

A predefined macro is a macro that is already understood by the C pre processor without the program needing to define it. Examples include

Mandatory Pre-Defined Macros

- `_FILE_`, which gives the file name of the current source file (a string literal),
- `_LINE_` for the current line number (an integer constant),
- `_DATE_` for the compilation date (a string literal),
- `_TIME_` for the compilation time (a string literal).

There's also a related predefined identifier, `_func_` (ISO/IEC 9899:2011 §6.4.2.2), which is *not* a macro:

The identifier `_func_` shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration:

```
static const char _func_[] = "function-name";
```

appeared, where `function-name` is the name of the lexically-enclosing function.

`_FILE_`, `_LINE_` and `_func_` are especially useful for debugging purposes. For example:

```
fprintf(stderr, "%s: %s: %d: Denominator is 0", _FILE_, _func_, _LINE_);
```

Pre-C99 compilers, may or may not support `_func_` or may have a macro that acts the same that is named differently. For example, gcc used `_FUNCTION_` in C89 mode.

The below macros allow to ask for detail on the implementation:

- `_STDC_VERSION_` The version of the C Standard implemented. This is a constant integer using the format `yyyymmL` (the value `201112L` for C11, the value `199901L` for C99; it wasn't defined for C89/C90)
- `_STDC_HOSTED_` 1 if it's a hosted implementation, else 0.
- `_STDC_` If 1, the implementation conforms to the C Standard.

Other Pre-Defined Macros (non mandatory)

ISO/IEC 9899:2011 §6.10.9.2 Environment macros:

- `_STDC_ISO_10646_` An integer constant of the form `yyyymmL` (for example, `199712L`). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type `wchar_t`, has the same value as the short identifier of that character. The Unicode required set consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.
- `_STDC_MB_MIGHT_NEQ_WC_` The integer constant 1, intended to indicate that, in the encoding for `wchar_t`, a member of the basic character set need not have a code value equal to its value when

用作单个字符的整数字符常量时。

- `_STDC_UTF_16_` 整数常量1，表示`char16_t`类型的值采用UTF-16编码。如果使用其他编码，则该宏不应被定义，实际使用的编码由实现定义。
- `_STDC_UTF_32_` 整数常量1，表示`char32_t`类型的值采用UTF-32编码。如果使用其他编码，则该宏不应被定义，实际使用的编码由实现定义。

ISO/IEC 9899:2011 §6.10.8.3 条件特性宏

- `_STDC_ANALYZABLE_` 整数常量1，表示符合附录L（可分析性）中的规范。
- `_STDC_IEC_559_` 整数常量1，表示符合附录F（IEC 60559 浮点算术）中的规范。
- `_STDC_IEC_559_COMPLEX_` 整数常量1，表示遵守附录G（IEC 60559 兼容复数算术）中的规范。
- `_STDC_LIB_EXT1_` 整数常量201112L，表示支持附录K（边界检查接口）中定义的扩展。
- `_STDC_NO_ATOMICS_` 整数常量1，表示实现不支持原子类型（包括`_Atomic`类型限定符）和`<stdatomic.h>`头文件。
- `_STDC_NO_COMPLEX_` 整数常量1，表示该实现不支持复数类型或`<complex.h>`头文件。
- `_STDC_NO_THREADS_` 整数常量1，表示实现不支持`<threads.h>`头文件。
- `_STDC_NO_VLA_` 整数常量1，表示实现不支持变长数组或可变修饰类型。

used as the lone character in an integer character constant.

- `_STDC_UTF_16_` The integer constant 1, intended to indicate that values of type `char16_t` are UTF-16 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.
- `_STDC_UTF_32_` The integer constant 1, intended to indicate that values of type `char32_t` are UTF-32 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

ISO/IEC 9899:2011 §6.10.8.3 Conditional feature macros

- `_STDC_ANALYZABLE_` The integer constant 1, intended to indicate conformance to the specifications in annex L (Analyzability).
- `_STDC_IEC_559_` The integer constant 1, intended to indicate conformance to the specifications in annex F (IEC 60559 floating-point arithmetic).
- `_STDC_IEC_559_COMPLEX_` The integer constant 1, intended to indicate adherence to the specifications in annex G (IEC 60559 compatible complex arithmetic).
- `_STDC_LIB_EXT1_` The integer constant 201112L, intended to indicate support for the extensions defined in annex K (Bounds-checking interfaces).
- `_STDC_NO_ATOMICS_` The integer constant 1, intended to indicate that the implementation does not support atomic types (including the `_Atomic` type qualifier) and the `<stdatomic.h>` header.
- `_STDC_NO_COMPLEX_` The integer constant 1, intended to indicate that the implementation does not support complex types or the `<complex.h>` header.
- `_STDC_NO_THREADS_` The integer constant 1, intended to indicate that the implementation does not support the `<threads.h>` header.
- `_STDC_NO_VLA_` The integer constant 1, intended to indicate that the implementation does not support variable length arrays or variably modified types.

第30.9节：可变参数宏

版本 ≥ C99

带可变参数的宏：

假设你想创建一个用于调试代码的打印宏，以下宏作为示例：

```
#define debug_print(msg) printf("%s:%d %s", __FILE__, __LINE__, msg)
```

一些使用示例：

函数`somfunc()`在失败时返回-1，成功时返回0，并且它在代码中被多个不同地方调用：

```
int retVal = somfunc();

if(retVal == -1)
{
    debug_printf("somfunc() has failed");
}

/* 其他代码 */
```

Section 30.9: Variadic arguments macro

Version ≥ C99

Macros with variadic args:

Let's say you want to create some print-macro for debugging your code, let's take this macro as an example:

```
#define debug_print(msg) printf("%s:%d %s", __FILE__, __LINE__, msg)
```

Some examples of usage:

The function `somfunc()` returns -1 if failed and 0 if succeeded, and it is called from plenty different places within the code:

```
int retVal = somfunc();

if(retVal == -1)
{
    debug_printf("somfunc() has failed");

    /* some other code */
```

```
RetVal = somefunc();
```

```
ifRetVal == -1)
{
    debug_printf("somefunc() has failed");
}
```

如果 `somefunc()` 的实现发生变化，并且它现在返回不同的值以匹配不同的可能的错误类型，会发生什么？你仍然想使用调试宏并打印错误值。

```
debug_printfRetVal); /* 这显然会失败 */
debug_printf("%d", retVal); /* 这也会失败 */
```

为了解决这个问题，引入了 `_VA_ARGS_` 宏。该宏允许多参数的 X 宏：

示例：

```
#define debug_print(msg, ...) printf(msg, __VA_ARGS__) \
    printf("错误发生在文件:行 (%s:%d)", __FILE__, __LINE)
```

用法：

```
int retVal = somefunc();
debug_print("somefunc() 的 retVal 是-> %d", retVal);
```

该宏允许你传递多个参数并打印它们，但现在它禁止你完全不传递任何参数。

```
debug_print("Hey");
```

这会引发一些语法错误，因为宏至少需要一个以上的参数，而预处理器不会忽略 `debug_print()` 宏中缺少逗号的情况。此外，`debug_print("Hey");` 也会引发语法错误，因为传递给宏的参数不能为空。

为了解决这个问题，引入了 `##__VA_ARGS__` 宏，该宏表示如果没有可变参数，预处理器会从代码中删除逗号。

示例：

```
#define debug_print(msg, ...) printf(msg, ##__VA_ARGS__) \
    printf("Error occurred in file:line (%s:%d)", __FILE__, __LINE)
```

用法：

```
debug_print("某函数的返回值？");
debug_print("%d", somefunc());
```

第30.10节：宏替换

宏替换最简单的形式是定义一个显式常量，如下所示

```
#define ARRSIZE 100
int array[ARRSIZE];
```

```
RetVal = somefunc();
```

```
ifRetVal == -1)
{
    debug_printf("somefunc() has failed");
}
```

What happens if the implementation of `somefunc()` changes, and it now returns different values matching different possible error types? You still want use the debug macro and print the error value.

```
debug_printfRetVal); /* this would obviously fail */
debug_printf("%d", retVal); /* this would also fail */
```

To solve this problem the `_VA_ARGS_` macro was introduced. This macro allows multiple parameters X-macro's:

Example:

```
#define debug_print(msg, ...) printf(msg, __VA_ARGS__) \
    printf("\nError occurred in file:line (%s:%d)\n", __FILE__, __LINE)
```

Usage:

```
int retVal = somefunc();
debug_print("RetVal of somefunc() is-> %d", retVal);
```

This macro allows you to pass multiple parameters and print them, but now it forbids you from sending any parameters at all.

```
debug_print("Hey");
```

This would raise some syntax error as the macro expects at least one more argument and the pre-processor would not ignore the lack of comma in the `debug_print()` macro. Also `debug_print("Hey",);` would raise a syntax error as you can't keep the argument passed to macro empty.

To solve this, `##__VA_ARGS__` macro was introduced, this macro states that if no variable arguments exist, the comma is deleted by the pre-processor from code.

Example:

```
#define debug_print(msg, ...) printf(msg, ##__VA_ARGS__) \
    printf("\nError occurred in file:line (%s:%d)\n", __FILE__, __LINE)
```

Usage:

```
debug_print("Ret val of somefunc()?");
debug_print("%d", somefunc());
```

Section 30.10: Macro Replacement

The simplest form of macro replacement is to define a manifest constant, as in

```
#define ARRSIZE 100
int array[ARRSIZE];
```

这定义了一个函数式宏，将变量乘以10并存储新值：

```
#define TIMES10(A) ((A) *= 10)

double b = 34;
int c = 23;

TIMES10(b); // 好的：((b) *= 10);
TIMES10(c); // 好的：((c) *= 10);
TIMES10(5); // 错误：((5) *= 10);
```

替换在程序文本的任何其他解释之前完成。在第一次调用TIMES10时，定义中的名称A被替换为b，然后将扩展后的文本放置在调用处。注意，这个TIMES10的定义并不等同于

```
#define TIMES10(A) ((A) = (A) * 10)
```

因为这可能会对A的替换进行两次求值，可能会产生不希望的副作用。

下面定义了一个类似函数的宏，其值是其参数中的最大值。它的优点是适用于任何兼容类型的参数，并且生成内联代码，无需函数调用的开销。缺点是会对其参数中的一个或另一个进行第二次求值（包括副作用），并且如果多次调用，会生成比函数更多的代码。

```
#define max(a, b) ((a) > (b) ? (a) : (b))

int maxVal = max(11, 43); /* 43 */
int maxValExpr = max(11 + 36, 51 - 7); /* 47 */

/* 不应这样做，因为表达式会被求值两次 */
int j = 0, i = 0;
int sideEffect = max(++i, ++j); /* i == 4 */
```

因此，这类对参数进行多次求值的宏通常在生产代码中避免使用。

自C11起，引入了_Generic特性，可以避免这种多次调用的问题。

宏展开中大量的括号（定义右侧）确保参数和结果表达式被正确绑定，并且能很好地适应宏调用的上下文。

第30.11节：错误指令

如果预处理器遇到#error指令，编译将被中止，并打印包含的诊断信息。

```
#define DEBUG

#ifndef DEBUG
#error "不支持调试构建"
#endif

int main(void) {
    return 0;
}
```

可能的输出：

This defines a *function-like* macro that multiplies a variable by 10 and stores the new value:

```
#define TIMES10(A) ((A) *= 10)

double b = 34;
int c = 23;

TIMES10(b); // good: ((b) *= 10);
TIMES10(c); // good: ((c) *= 10);
TIMES10(5); // bad: ((5) *= 10);
```

The replacement is done before any other interpretation of the program text. In the first call to TIMES10 the name A from the definition is replaced by b and the so expanded text is then put in place of the call. Note that this definition of TIMES10 is not equivalent to

```
#define TIMES10(A) ((A) = (A) * 10)
```

because this could evaluate the replacement of A, twice, which can have unwanted side effects.

The following defines a function-like macro which value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and of generating more code than a function if invoked several times.

```
#define max(a, b) ((a) > (b) ? (a) : (b))

int maxVal = max(11, 43); /* 43 */
int maxValExpr = max(11 + 36, 51 - 7); /* 47 */

/* Should not be done, due to expression being evaluated twice */
int j = 0, i = 0;
int sideEffect = max(++i, ++j); /* i == 4 */
```

Because of this, such macros that evaluate their arguments multiple times are usually avoided in production code. Since C11 there is the _Generic feature that allows to avoid such multiple invocations.

The abundant parentheses in the macro expansions (right hand side of the definition) ensure that the arguments and the resulting expression are bound properly and fit well into the context in which the macro is called.

Section 30.11: Error directive

If the preprocessor encounters an #error directive, compilation is halted and the diagnostic message included is printed.

```
#define DEBUG

#ifndef DEBUG
#error "Debug Builds Not Supported"
#endif

int main(void) {
    return 0;
}
```

Possible output:

```
$ gcc error.c  
error.c: error: #error "不支持调试构建"
```

第30.12节：FOREACH实现

我们也可以使用宏来使代码更易读写。例如，我们可以实现宏来实现C语言中针对某些数据结构（如单链表、双链表、队列等）的foreach结构。

这里有一个小例子。

```
#include <stdio.h>  
#include <stdlib.h>  
  
struct LinkedListNode  
{  
    int 数据;  
    struct LinkedListNode *next;  
};  
  
#define FOREACH_LIST(node, list) \  
    for (node=list; node; node=node->next)  
  
/* 用法 */  
int main(void)  
{  
    struct LinkedListNode *list, **plist = &list, *node;  
    int i;  
  
    for (i=0; i<10; i++)  
    {  
        *plist = malloc(sizeof(struct LinkedListNode));  
        (*plist)->data = i;  
        (*plist)->next = NULL;  
        plist = &(*plist)->next;  
    }  
  
    /* 在这里打印元素 */  
    FOREACH_LIST(node, list)  
    {  
        printf("%d", node->data);  
    }  
}
```

您可以为此类数据结构制作一个标准接口，并编写一个通用的FOREACH实现，如下所示：

```
#include <stdio.h>  
#include <stdlib.h>  
  
typedef struct CollectionItem_  
{  
    int 数据;  
    struct CollectionItem_ *next;  
} CollectionItem;  
  
typedef struct Collection_  
{  
    /* 接口函数 */  
    void* (*first)(void *coll);  
    void* (*last) (void *coll);  
    void* (*next) (void *coll, CollectionItem *currItem);  
} Collection;
```

```
$ gcc error.c  
error.c: error: #error "Debug Builds Not Supported"
```

Section 30.12: FOREACH implementation

We can also use macros for making code easier to read and write. For example we can implement macros for implementing the foreach construct in C for some data structures like singly- and doubly-linked lists, queues, etc.

Here is a small example.

```
#include <stdio.h>  
#include <stdlib.h>  
  
struct LinkedListNode  
{  
    int data;  
    struct LinkedListNode *next;  
};  
  
#define FOREACH_LIST(node, list) \  
    for (node=list; node; node=node->next)  
  
/* Usage */  
int main(void)  
{  
    struct LinkedListNode *list, **plist = &list, *node;  
    int i;  
  
    for (i=0; i<10; i++)  
    {  
        *plist = malloc(sizeof(struct LinkedListNode));  
        (*plist)->data = i;  
        (*plist)->next = NULL;  
        plist = &(*plist)->next;  
    }  
  
    /* printing the elements here */  
    FOREACH_LIST(node, list)  
    {  
        printf("%d\n", node->data);  
    }  
}
```

You can make a standard interface for such data-structures and write a generic implementation of FOREACH as:

```
#include <stdio.h>  
#include <stdlib.h>  
  
typedef struct CollectionItem_  
{  
    int data;  
    struct CollectionItem_ *next;  
} CollectionItem;  
  
typedef struct Collection_  
{  
    /* interface functions */  
    void* (*first)(void *coll);  
    void* (*last) (void *coll);  
    void* (*next) (void *coll, CollectionItem *currItem);  
} Collection;
```

```

CollectionItem *collectionHead;
/* 其他字段 */
} Collection;

/* 必须实现 */
void *first(void *coll)
{
    return ((Collection*)coll)->collectionHead;
}

/* 必须实现 */
void *last(void *coll)
{
    return NULL;
}

/* 必须实现 */
void *next(void *coll, CollectionItem *curr)
{
    return curr->next;
}

CollectionItem *new_CollectionItem(int data)
{
CollectionItem *item = malloc(sizeof(CollectionItem));
    item->data = data;
item->next = NULL;
    return item;
}

void Add_Collection(Collection *coll, int data)
{
CollectionItem **item = &coll->collectionHead;
    while(*item)
item = &(*item)->next;
    (*item) = new_CollectionItem(data);
}

Collection *new_Collection()
{
Collection *nc = malloc(sizeof(Collection));
    nc->first = first;
nc->last = last;
    nc->next = next;
    return nc;
}

/* 通用实现 */
#define FOREACH(node, collection) \
    for (node = (collection)->first(collection); \
         node != (collection)->last(collection); \
         node = (collection)->next(collection, node))

int main(void)
{
Collection *coll = new_Collection();
    CollectionItem *node;
    int i;

    for(i=0; i<10; i++)
    {
Add_Collection(coll, i);

```

```

CollectionItem *collectionHead;
/* Other fields */
} Collection;

/* must implement */
void *first(void *coll)
{
    return ((Collection*)coll)->collectionHead;
}

/* must implement */
void *last(void *coll)
{
    return NULL;
}

/* must implement */
void *next(void *coll, CollectionItem *curr)
{
    return curr->next;
}

CollectionItem *new_CollectionItem(int data)
{
CollectionItem *item = malloc(sizeof(CollectionItem));
    item->data = data;
item->next = NULL;
    return item;
}

void Add_Collection(Collection *coll, int data)
{
CollectionItem **item = &coll->collectionHead;
    while(*item)
item = &(*item)->next;
    (*item) = new_CollectionItem(data);
}

Collection *new_Collection()
{
Collection *nc = malloc(sizeof(Collection));
    nc->first = first;
nc->last = last;
    nc->next = next;
    return nc;
}

/* generic implementation */
#define FOREACH(node, collection) \
    for (node = (collection)->first(collection); \
         node != (collection)->last(collection); \
         node = (collection)->next(collection, node))

int main(void)
{
    Collection *coll = new_Collection();
    CollectionItem *node;
    int i;

    for(i=0; i<10; i++)
    {
        Add_Collection(coll, i);

```

```
}

/* 在这里打印元素 */
FOREACH(node, coll)
{
    printf("%d", node->data);
}

}
```

要使用此通用实现，只需为您的数据结构实现这些函数。

1. void* (*first)(void *coll);
2. void* (*last) (void *coll);
3. void* (*next) (void *coll, CollectionItem *currItem);

```
}

/* printing the elements here */
FOREACH(node, coll)
{
    printf("%d\n", node->data);
}

}
```

To use this generic implementation just implement these functions for your data structure.

1. void* (*first)(void *coll);
2. void* (*last) (void *coll);
3. void* (*next) (void *coll, CollectionItem *currItem);

第31章：信号处理

参数	详情
sig	要设置信号处理程序的信号，取值为SIGABRT、SIGFPE、SIGILL、SIGTERM、SIGINT、SIGSEGV或某些实现定义的值
func	信号处理程序，可以是以下之一：SIG_DFL，表示默认处理程序；SIG_IGN，表示忽略信号；或者是具有签名void foo(int sig);的函数指针。

第31.1节：使用“signal()”进行信号处理

信号编号可以是同步的（如SIGSEGV——分段错误），当它们由程序自身的故障触发时；也可以是异步的（如SIGINT——交互式中断），当它们由程序外部发起时，例如通过按键Ctrl-C。

signal() 函数是 ISO C 标准的一部分，可用于分配一个函数来处理特定的信号

```
#include <stdio.h> /* printf() */
#include <stdlib.h> /* abort() */
#include <signal.h> /* signal() */

void handler_nonportable(int sig)
{
    /* 未定义行为，可能在特定平台上正常 */
    printf("捕获信号: %d", sig);

    /* 调用 abort 是安全的 */
    abort();
}

sig_atomic_t volatile finished = 0;

void handler(int sig)
{
    switch (sig) {
        /* 硬件中断不应返回 */
        case SIGSEGV:
        case SIGFPE:
        case SIGILL:
    }
}

/* 在 C11 之前使用 _Exit */
_EXIT(EXIT_FAILURE);

default:
    /* 重置信号为默认处理程序，这样如果返回时出现问题，我们就不会再次被调用。 */
    signal(sig, SIG_DFL);
    /* 通知所有人我们已经完成 */
    finished = sig;
    return;
}

int main(void)
{
```

Chapter 31: Signal handling

Parameter	Details
sig	The signal to set the signal handler to, one of SIGABRT, SIGFPE, SIGILL, SIGTERM, SIGINT, SIGSEGV or some implementation defined value
func	The signal handler, which is either of the following: SIG_DFL, for the default handler, SIG_IGN to ignore the signal, or a function pointer with the signature void foo(int sig);.

Section 31.1: Signal Handling with “signal()”

[Signal numbers](#) can be synchronous (like SIGSEGV – segmentation fault) when they are triggered by a malfunctioning of the program itself or asynchronous (like SIGINT - interactive attention) when they are initiated from outside the program, e.g. by a keypress as Ctrl-C.

The signal() function is part of the ISO C standard and can be used to assign a function to handle a specific signal

```
#include <stdio.h> /* printf() */
#include <stdlib.h> /* abort() */
#include <signal.h> /* signal() */

void handler_nonportable(int sig)
{
    /* undefined behavior, maybe fine on specific platform */
    printf("Caught: %d\n", sig);

    /* abort is safe to call */
    abort();
}

sig_atomic_t volatile finished = 0;

void handler(int sig)
{
    switch (sig) {
        /* hardware interrupts should not return */
        case SIGSEGV:
        case SIGFPE:
        case SIGILL:
    }
}

/* quick_exit is safe to call */
quick_exit(EXIT_FAILURE);

default:
    /* Reset the signal to the default handler,
       so we will not be called again if things go
       wrong on return. */
    signal(sig, SIG_DFL);
    /* let everybody know that we are finished */
    finished = sig;
    return;
}

int main(void)
{
```

```

/* 捕获 SIGSEGV 信号，该信号在发生段错误（即空指针访问）时触发 */
if (signal(SIGSEGV, &handler) == SIG_ERR) {
    perror("无法建立SIGSEGV信号处理程序");
    return EXIT_FAILURE;
}

/* 捕获SIGTERM信号，终止请求 */
if (signal(SIGTERM, &handler) == SIG_ERR) {
    perror("无法建立SIGTERM信号处理程序");
    return EXIT_FAILURE;
}

/* 忽略SIGINT信号，通过将处理器设置为'SIG_IGN'。 */
signal(SIGINT, SIG_IGN);

/* 在这里执行一些耗时操作，并留出时间通过键盘终止程序。 */

/* 然后：*/

if (finished) {
    fprintf(stderr, "程序已被信号 %d 终止", (int)finished);return EXIT_FAILURE;
}
}

/* 尝试强制产生段错误，并触发SIGSEGV信号 */
{
    char* ptr = 0;
    *ptr = 0;
}

/* 这段代码永远不应被执行 */
return EXIT_SUCCESS;
}

```

使用 `signal()` 会对信号处理程序内允许执行的操作施加重要限制，详见备注以获取更多信息。

POSIX 建议使用 `sigaction()` 替代 `signal()`，因为后者行为未明确定义且实现差异较大。POSIX 还定义了比 ISO C 标准更多的信号，包括 SIGUSR1 和 SIGUSR2，程序员可以自由使用它们来实现任何目的。

```

/* Catch the SIGSEGV signal, raised on segmentation faults (i.e. NULL ptr access) */
if (signal(SIGSEGV, &handler) == SIG_ERR) {
    perror("could not establish handler for SIGSEGV");
    return EXIT_FAILURE;
}

/* Catch the SIGTERM signal, termination request */
if (signal(SIGTERM, &handler) == SIG_ERR) {
    perror("could not establish handler for SIGTERM");
    return EXIT_FAILURE;
}

/* Ignore the SIGINT signal, by setting the handler to 'SIG_IGN'. */
signal(SIGINT, SIG_IGN);

/* Do something that takes some time here, and leaves
   the time to terminate the program from the keyboard. */

/* Then: */

if (finished) {
    fprintf(stderr, "we have been terminated by signal %d\n", (int)finished);
    return EXIT_FAILURE;
}

/* Try to force a segmentation fault, and raise a SIGSEGV */
{
    char* ptr = 0;
    *ptr = 0;
}

/* This should never be executed */
return EXIT_SUCCESS;
}

```

Using `signal()` imposes important limitations what you are allowed to do inside the signal handlers, see the remarks for further information.

POSIX recommends the usage of `sigaction()` instead of `signal()`, due to its underspecified behavior and significant implementation variations. POSIX also defines many more signals than ISO C standard, including SIGUSR1 and SIGUSR2, which can be used freely by the programmer for any purpose.

第32章：可变参数

参数

va_list ap 参数指针，变长参数列表中的当前位置
最后一个非变长函数参数的名称，以便编译器找到正确的位置开始处理变长参数；不能声明为 register 变量、函数或数组类型

类型 要读取的变长参数的提升类型（例如，int 用于 short int 参数）

va_list src 当前要复制的参数指针

va_list dst 要填充的新参数列表

可变参数用于printf系列函数（printf、fprintf等）以及其他函数，允许函数每次调用时参数数量不同，因此得名varargs。

要实现使用可变参数特性的函数，请使用#include <stdarg.h>。

调用接受可变数量参数的函数时，确保作用域内有带尾部省略号的完整原型：例如void err_exit(const char *format, ...);

第32.1节：使用显式计数参数确定va_list的长度

对于任何可变参数函数，函数必须知道如何解释可变参数列表。对于printf()或scanf()函数，格式字符串告诉函数预期的内容。

最简单的技术是传递其他参数的显式计数（这些参数通常都是相同类型）。

下面代码中的可变参数函数演示了这一点，该函数计算一系列整数的和，整数数量可变，但该数量作为参数在可变参数列表之前指定。

```
#include <stdio.h>
#include <stdarg.h>

/* 第一个参数是后续要相加的整数参数的数量。*/
int sum(int n, ...) {
    int sum = 0;
    va_list it; /* 保存可变参数列表的信息。*/

    va_start(it, n); /* 开始处理可变参数 */
    while (n--)
        sum += va_arg(it, int); /* 获取并累加下一个可变参数 */
    va_end(it); /* 结束可变参数处理 */

    return sum;
}

int main(void)
{
    printf("%d", sum(5, 1, 2, 3, 4, 5)); /* 输出 15 */printf("%d", sum(
    10, 5, 9, 2, 5, 111, 6666, 42, 1, 43, -6218)); /* 输出 666 */return 0;
}
```

Chapter 32: Variable arguments

Parameter

va_list ap argument pointer, current position in the list of variadic arguments

last name of last non-variadic function argument, so the compiler finds the correct place to start processing variadic arguments; may not be declared as a register variable, a function, or an array type

type promoted type of the variadic argument to read (e.g. int for a short int argument)

va_list src current argument pointer to copy

va_list dst new argument list to be filled in

Variable arguments are used by functions in the printf family (printf, fprintf, etc) and others to allow a function to be called with a different number of arguments each time, hence the name varargs.

To implement functions using the variable arguments feature, use #include <stdarg.h>.

To call functions which take a variable number of arguments, ensure there is a full prototype with the trailing ellipsis in scope: void err_exit(const char *format, ...); for example.

Section 32.1: Using an explicit count argument to determine the length of the va_list

With any variadic function, the function must know how to interpret the variable arguments list. With the printf() or scanf() functions, the format string tells the function what to expect.

The simplest technique is to pass an explicit count of the other arguments (which are normally all the same type). This is demonstrated in the variadic function in the code below which calculates the sum of a series of integers, where there may be any number of integers but that count is specified as an argument prior to the variable argument list.

```
#include <stdio.h>
#include <stdarg.h>

/* first arg is the number of following int args to sum. */
int sum(int n, ...) {
    int sum = 0;
    va_list it; /* hold information about the variadic argument list. */

    va_start(it, n); /* start variadic argument processing */
    while (n--)
        sum += va_arg(it, int); /* get and sum the next variadic argument */
    va_end(it); /* end variadic argument processing */

    return sum;
}

int main(void)
{
    printf("%d\n", sum(5, 1, 2, 3, 4, 5)); /* prints 15 */
    printf("%d\n", sum(10, 5, 9, 2, 5, 111, 6666, 42, 1, 43, -6218)); /* prints 666 */
    return 0;
}
```

第32.2节：使用终止值来确定

va_list的结束

对于任何可变参数函数，函数必须知道如何解释可变参数列表。传统的“方法”（以printf为例）是预先指定参数个数。然而，这并不总是一个好主意：

```
/* 第一个参数指定参数个数；其余参数也是int类型 */
extern int sum(int n, ...);

/* 但从代码中很难看出这一点。 */
sum(5, 2, 1, 4, 3, 6)

/* 如果例如某个参数后来被移除，会发生什么？ */
sum(5, 2, 1, 3, 6) /* 灾难 */
```

有时添加一个显式的终止符更稳健，POSIX的execvp()函数就是一个例子。这里有另一个函数，用于计算一系列double类型数字的和：

```
#include <stdarg.h>
#include <stdio.h>
#include <math.h>

/* 计算参数之和，直到遇到终止符NAN */
double sum (double x, ...) {
    double sum = 0;
    va_list va;

    va_start(va, x);
    for (; !isnan(x); x = va_arg(va, double)) {
        sum += x;
    }
    va_end(va);

    return sum;
}

int main (void) {
    printf("%g", sum(5., 2., 1., 4., 3., 6., NAN)); printf("%g", sum(
    1, 0.5, 0.25, 0.125, 0.0625, 0.03125, NAN));
}
```

良好的终止符值：

- 整数（应全部为正数或非负数）—0或-1
- 浮点类型—NAN
- 指针类型—NULL
- 枚举类型—某些特殊值

第32.3节：实现具有`printf()`类似接口的函数

可变长度参数列表的一个常见用途是实现围绕printf()系列函数的薄包装函数。一个例子是一组错误报告函数。

errmsg.h

Section 32.2: Using terminator values to determine the end of va_list

With any variadic function, the function must know how to interpret the variable arguments list. The “traditional” approach (exemplified by printf) is to specify number of arguments up front. However, this is not always a good idea:

```
/* First argument specifies the number of parameters; the remainder are also int */
extern int sum(int n, ...);

/* But it's far from obvious from the code. */
sum(5, 2, 1, 4, 3, 6)

/* What happens if i.e. one argument is removed later on? */
sum(5, 2, 1, 3, 6) /* Disaster */
```

Sometimes it's more robust to add an explicit terminator, exemplified by the POSIX [execvp\(\)](#) function. Here's another function to calculate the sum of a series of double numbers:

```
#include <stdarg.h>
#include <stdio.h>
#include <math.h>

/* Sums args up until the terminator NAN */
double sum (double x, ...) {
    double sum = 0;
    va_list va;

    va_start(va, x);
    for (; !isnan(x); x = va_arg(va, double)) {
        sum += x;
    }
    va_end(va);

    return sum;
}

int main (void) {
    printf("%g\n", sum(5., 2., 1., 4., 3., 6., NAN));
    printf("%g\n", sum(1, 0.5, 0.25, 0.125, 0.0625, 0.03125, NAN));
}
```

Good terminator values:

- integer (supposed to be all positive or non-negative) — 0 or -1
- floating point types — NAN
- pointer types — NULL
- enumerator types — some special value

Section 32.3: Implementing functions with a `printf()`-like interface

One common use of variable-length argument lists is to implement functions that are a thin wrapper around the printf() family of functions. One such example is a set of error reporting functions.

errmsg.h

```

#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h> // C11

void verrmsg(int errnum, const char *fmt, va_list ap);
noreturn void errmsg(int exitcode, int errnum, const char *fmt, ...);
void warnmsg(int errnum, const char *fmt, ...);

#endif

```

这是一个最基本的示例；此类包可以更加复杂。通常，程序员会使用以下任一方式 errmsg() 或 warnmsg()，它们内部使用了 verrmsg()。不过，如果有人需要做更多的事情，那么暴露的 verrmsg() 函数将会有用。你可以避免暴露它，直到你真正需要它（YAGNI——你不会需要它），但这个需求最终会出现（你确实会需要它——YAGNI）。

errmsg.c

这段代码只需将可变参数转发给 vfprintf() 函数，以输出到标准错误流。它还会报告传递给函数的系统错误号（errno）对应的系统错误信息。

```

#include "errmsg.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void
verrmsg(int errnum, const char *fmt, va_list ap)
{
    如果(fmt)
        vfprintf(stderr, fmt, ap);
    if (errnum != 0)
        fprintf(stderr, ": %s", strerror(errnum));putc('\n', stderr);
}

无效
errmsg(int exitcode, int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
    exit(exitcode);
}

无效
warnmsg(int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
}

```

使用 errmsg.h

```

#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h> // C11

void verrmsg(int errnum, const char *fmt, va_list ap);
noreturn void errmsg(int exitcode, int errnum, const char *fmt, ...);
void warnmsg(int errnum, const char *fmt, ...);

#endif

```

This is a bare-bones example; such packages can be much elaborate. Normally, programmers will use either errmsg() or warnmsg(), which themselves use verrmsg() internally. If someone comes up with a need to do more, though, then the exposed verrmsg() function will be useful. You could avoid exposing it until you have a need for it (YAGNI — you aren't gonna need it), but the need will arise eventually (you *are gonna need it* — YAGNI).

errmsg.c

This code only needs to forward the variadic arguments to the [vfprintf\(\)](#) function for outputting to standard error. It also reports the system error message corresponding to the system error number (errno) passed to the functions.

```

#include "errmsg.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void
verrmsg(int errnum, const char *fmt, va_list ap)
{
    if (fmt)
        vfprintf(stderr, fmt, ap);
    if (errnum != 0)
        fprintf(stderr, ": %s", strerror(errnum));
    putc('\n', stderr);
}

void
errmsg(int exitcode, int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
    exit(exitcode);
}

void
warnmsg(int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
}

```

Using errmsg.h

现在你可以如下使用这些函数：

```
#include "errmsg.h"
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char buffer[BUFSIZ];
    int fd;
    if (argc != 2)
    {
        fprintf(stderr, "用法：%s 文件名", argv[0]);exit(EXIT_FAILURE);
    }
    const char *filename = argv[1];

    if ((fd = open(filename, O_RDONLY)) == -1)
        errmsg(EXIT_FAILURE, errno, "cannot open %s", filename);
    if (read(fd, buffer, sizeof(buffer)) != sizeof(buffer))
        errmsg(EXIT_FAILURE, errno, "cannot read %zu bytes from %s", sizeof(buffer), filename);
    if (close(fd) == -1)
        warnmsg(errno, "cannot close %s", filename);
    /* 继续程序 */
    return 0;
}
```

如果 `open()` 或 `read()` 系统调用失败，错误信息将写入标准错误，程序以退出码 1 退出。如果 `close()` 系统调用失败，错误仅作为警告信息打印，程序继续执行。

检查 `printf()` 格式的正确使用

如果你使用的是 GCC (GNU C 编译器，属于 GNU 编译器集合的一部分) 或 Clang，那么你可以让编译器检查传递给错误消息函数的参数是否符合 `printf()` 的预期。由于并非所有编译器都支持该扩展，因此需要有条件地编译，这有点繁琐。但它提供的保护是值得的。

首先，我们需要知道如何检测编译器是 GCC 还是模拟 GCC 的 Clang。答案是 GCC 定义了 `_GNUC_` 来表示这一点。

有关属性的信息，请参见 [common function attributes](#)，特别是 `format` 属性。

重写 errmsg.h

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h> // C11

#if !defined(PRINTFLIKE)
#if defined(__GNUC__)
#define PRINTFLIKE(n,m) __attribute__((format.printf,n,m)))
#else
#define PRINTFLIKE(n,m) /* 如果仅仅是 */
#endif
#endif
```

Now you can use those functions as follows:

```
#include "errmsg.h"
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char buffer[BUFSIZ];
    int fd;
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    const char *filename = argv[1];

    if ((fd = open(filename, O_RDONLY)) == -1)
        errmsg(EXIT_FAILURE, errno, "cannot open %s", filename);
    if (read(fd, buffer, sizeof(buffer)) != sizeof(buffer))
        errmsg(EXIT_FAILURE, errno, "cannot read %zu bytes from %s", sizeof(buffer), filename);
    if (close(fd) == -1)
        warnmsg(errno, "cannot close %s", filename);
    /* continue the program */
    return 0;
}
```

If either the `open()` or `read()` system calls fails, the error is written to standard error and the program exits with exit code 1. If the `close()` system call fails, the error is merely printed as a warning message, and the program continues.

Checking the correct use of `printf()` formats

If you are using GCC (the GNU C Compiler, which is part of the GNU Compiler Collection), or using Clang, then you can have the compiler check that the arguments you pass to the error message functions match what `printf()` expects. Since not all compilers support the extension, it needs to be compiled conditionally, which is a little bit fiddly. However, the protection it gives is worth the effort.

First, we need to know how to detect that the compiler is GCC or Clang emulating GCC. The answer is that GCC defines `_GNUC_` to indicate that.

See [common function attributes](#) for information about the attributes — specifically the `format` attribute.

Rewritten errmsg.h

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h> // C11

#if !defined(PRINTFLIKE)
#if defined(__GNUC__)
#define PRINTFLIKE(n,m) __attribute__((format.printf,n,m)))
#else
#define PRINTFLIKE(n,m) /* If only */
#endif
#endif
```

```
#endif /* __GNUC__ */
#endif /* PRINTFLIKE */

void verrmsg(int errnum, const char *fmt, va_list ap);
void noreturn errmsg(int exitcode, int errnum, const char *fmt, ...)
    PRINTFLIKE(3, 4);
void warnmsg(int errnum, const char *fmt, ...)
    PRINTFLIKE(2, 3);

#endif
```

现在，如果你犯了这样的错误：

```
errmsg(EXIT_FAILURE, errno, "无法打开文件 '%d' 进行读取", filename);
```

(其中%d应为%s) , 编译器将报错：

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes \
>     -Wold-style-definition -c erruse.c
erruse.c: 函数'main'中：
erruse.c: 20: 64 : 错误：格式'%d'期望参数类型为'int'，但第4个参数类型为'const char
*' [-Werror=format=]
errmsg(EXIT_FAILURE, errno, "无法打开文件 '%d' 进行读取", filename);
                                     ^~~~~~
                                     %s
cc1: 所有警告均视为错误
$
```

第32.4节：使用格式字符串

使用格式字符串提供了关于后续可变参数数量和类型的信息，从而避免了需要显式的计数参数或终止值。

下面的示例展示了一个包装标准printf()函数的函数，只允许使用类型为char、int和double（十进制浮点格式）的可变参数。这里，和printf()一样，包装函数的第一个参数是格式字符串。随着格式字符串的解析，函数能够确定是否还期望另一个可变参数以及其类型。

```
#include <stdio.h>
#include <stdarg.h>

int simple_printf(const char *format, ...)
{
    va_list ap; /* 保存可变参数列表的信息。 */
    int printed = 0; /* 已打印字符计数 */

    va_start(ap, format); /* 开始处理可变参数 */

    while (*format != '\0') /* 读取格式字符串直到字符串结束符 */
    {
        int f = 0;

        if (*format == '%')
        {
            ++format;
            switch(*format)
            {
                case 'c' :
```

```
#endif /* __GNUC__ */
#endif /* PRINTFLIKE */

void verrmsg(int errnum, const char *fmt, va_list ap);
void noreturn errmsg(int exitcode, int errnum, const char *fmt, ...)
    PRINTFLIKE(3, 4);
void warnmsg(int errnum, const char *fmt, ...)
    PRINTFLIKE(2, 3);

#endif
```

Now, if you make a mistake like:

```
errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
```

(where the %d should be %s), then the compiler will complain:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes \
>     -Wold-style-definition -c erruse.c
erruse.c: In function 'main':
erruse.c:20:64: error: format '%d' expects argument of type 'int', but argument 4 has type 'const char
*' [-Werror=format=]
errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
                                     ^~~~~~
                                     %s
cc1: all warnings being treated as errors
$
```

Section 32.4: Using a format string

Using a format string provides information about the expected number and type of the subsequent variadic arguments in such a way as to avoid the need for an explicit count argument or a terminator value.

The example below shows a function that wraps the standard `printf()` function, only allowing for the use of variadic arguments of the type `char`, `int` and `double` (in decimal floating point format). Here, like with `printf()`, the first argument to the wrapping function is the format string. As the format string is parsed the function is able to determine if there is another variadic argument expected and what its type should be.

```
#include <stdio.h>
#include <stdarg.h>

int simple_printf(const char *format, ...)
{
    va_list ap; /* hold information about the variadic argument list. */
    int printed = 0; /* count of printed characters */

    va_start(ap, format); /* start variadic argument processing */

    while (*format != '\0') /* read format string until string terminator */
    {
        int f = 0;

        if (*format == '%')
        {
            ++format;
            switch(*format)
            {
                case 'c' :
```

```

f = printf("%d", va_arg(ap, int)); /* 打印下一个可变参数, 注意类型从 char 提升为 int */
    break;
case 'd' :
f = printf("%d", va_arg(ap, int)); /* 打印下一个可变参数 */
    break;

    case 'f' :
f = printf("%f", va_arg(ap, double)); /* 打印下一个可变参数 */
    break;
default :
f = -1; /* 无效的格式说明符 */
    break;
}
else
{
f = printf("%c", *format); /* 打印其他字符 */
}

if (f < 0) /* 检查错误 */
{
printed = f;
    break;
}
else
{
printed += f;
}
++format; /* 移动到字符串中的下一个字符 */
}

va_end(ap); /* 结束可变参数处理 */

return printed;
}

int main (int argc, char *argv[])
{
    int x = 40;
    int y = 0;

y = simple_printf("这句话中有 %d 个字符", x);    simple_printf("打印了 %d 个字符", y);
}

```

```

f = printf("%d", va_arg(ap, int)); /* print next variadic argument, note type
promotion from char to int */
    break;
case 'd' :
f = printf("%d", va_arg(ap, int)); /* print next variadic argument */
    break;

case 'f' :
f = printf("%f", va_arg(ap, double)); /* print next variadic argument */
    break;
default :
f = -1; /* invalid format specifier */
    break;
}
else
{
f = printf("%c", *format); /* print any other characters */
}

if (f < 0) /* check for errors */
{
printed = f;
    break;
}
else
{
printed += f;
}
++format; /* move on to next character in string */
}

va_end(ap); /* end variadic argument processing */

return printed;
}

int main (int argc, char *argv[])
{
    int x = 40;
    int y = 0;

y = simple_printf("There are %d characters in this sentence", x);
simple_printf("\n%d were printed\n", y);
}

```

第33章：断言

参数	详情
标量类型的表达式。	
消息	包含在诊断消息中的字符串字面量。

断言 (assertion) 是一个谓词，表示在软件遇到该断言时，所呈现的条件必须为真。最常见的是简单断言 (simple assertions)，它们在执行时进行验证。然而，静态断言 (static assertions) 是在编译时检查的。

第33.1节：简单断言

断言是一条语句，用于断定当代码执行到该行时某个事实必须为真。断言对于确保满足预期条件非常有用。当传递给断言的条件为真时，不会有任何操作。条件为假时的行为取决于编译器标志。当启用断言时，条件为假会导致程序立即停止。禁用断言时，则不采取任何操作。通常的做法是在内部和调试版本中启用断言，在发布版本中禁用，尽管断言在发布版本中也常被启用。（程序中终止是否优于错误取决于具体情况。）断言应仅用于捕获内部编程错误，通常意味着传入了错误的参数。

```
#include <stdio.h>
/* 取消注释以禁用 `assert()` */
/* #define NDEBUG */
#include <assert.h>

int main(void)
{
    int x = -1;
    assert(x >= 0);printf("x = %d", x);
    return 0;
}
```

可能的输出（未定义NDEBUG时）：

```
a.out: main.c:9: main: Assertion `x >= 0' failed.
```

可能的输出（定义了NDEBUG时）：

```
x = -1
```

全局定义NDEBUG是一个良好的实践，这样你可以轻松地编译代码，使所有断言要么全部启用，要么全部禁用。一个简单的方法是在编译器选项中定义NDEBUG，或者在共享配置头文件（例如config.h）中定义它。

第33.2节：静态断言

版本 ≥ C11

静态断言用于检查代码编译时条件是否为真。如果不为真，编译器必须发出错误信息并停止编译过程。

Chapter 33: Assertion

Parameter	Details
expression	expression of scalar type.
message	string literal to be included in the diagnostic message.

An **assertion** is a predicate that the presented condition must be true at the moment the assertion is encountered by the software. Most common are **simple assertions**, which are validated at execution time. However, **static assertions** are checked at compile time.

Section 33.1: Simple Assertion

An assertion is a statement used to assert that a fact must be true when that line of code is reached. Assertions are useful for ensuring that expected conditions are met. When the condition passed to an assertion is true, there is no action. The behavior on false conditions depends on compiler flags. When assertions are enabled, a false input causes an immediate program halt. When they are disabled, no action is taken. It is common practice to enable assertions in internal and debug builds, and disable them in release builds, though assertions are often enabled in release. (Whether termination is better or worse than errors depends on the program.) Assertions should be used only to catch internal programming errors, which usually means being passed bad parameters.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int main(void)
{
    int x = -1;
    assert(x >= 0);
    printf("x = %d\n", x);
    return 0;
}
```

Possible output with NDEBUG undefined:

```
a.out: main.c:9: main: Assertion `x >= 0' failed.
```

Possible output with NDEBUG defined:

```
x = -1
```

It's good practice to define NDEBUG globally, so that you can easily compile your code with all assertions either on or off. An easy way to do this is define NDEBUG as an option to the compiler, or define it in a shared configuration header (e.g. config.h).

Section 33.2: Static Assertion

Version ≥ C11

Static assertions are used to check if a condition is true when the code is compiled. If it isn't, the compiler is required to issue an error message and stop the compiling process.

静态断言是在编译时检查的，而非运行时。条件必须是常量表达式，如果为假将导致编译错误。第一个参数，即被检查的条件，必须是常量表达式，第二个参数是字符串字面量。

与 assert 不同，_Static_assert 是一个关键字。在<assert.h>中定义了一个方便的宏static_assert。

```
#include <assert.h>

enum {N = 5};
_Static_assert(N == 5, "N 不等于 5");
static_assert(N > 10, "N 不大于 10"); /* 编译错误 */
版本 = C99
```

在 C11 之前，没有对静态断言的直接支持。然而，在 C99 中，可以通过宏模拟静态断言，如果编译时条件为假，则会触发编译失败。与 _Static_assert 不同，第二个参数需要是一个合法的标识符，以便用它创建变量名。如果断言失败，该变量名会出现在编译器错误中，因为该变量被用在了语法错误的数组声明中。

```
#define STATIC_MSG(msg, l) STATIC_MSG2(msg, l)
#define STATIC_MSG2(msg,l) on_line_##l##_##msg
#define STATIC_ASSERT(x, msg) extern char STATIC_MSG(msg, __LINE__) [(x)?1:-1]

enum { N = 5 };
STATIC_ASSERT(N == 5, N_must_equal_5);
STATIC_ASSERT(N > 5, N_must_be_greater_than_5); /* 编译错误 */
```

在 C99 之前，不能在代码块的任意位置声明变量，因此必须非常小心使用此宏，确保它只出现在变量声明有效的位
置。

第 33.3 节：断言错误信息

有一个技巧可以在断言时显示错误信息。通常，你会这样写代码

```
void f(void *p)
{
    assert(p != NULL);
    /* more code */
}
```

如果断言失败，错误信息将类似于

断言失败：p != NULL，文件 main.c，第 5 行

但是，你也可以使用逻辑与（&&）来给出错误信息

```
void f(void *p)
{
    assert(p != NULL && "函数 f: p 不能为空");
    /* more code */
}
```

现在，如果断言失败，错误信息将类似于

A static assertion is one that is checked at compile time, not run time. The condition must be a constant expression, and if false will result in a compiler error. The first argument, the condition that is checked, must be a constant expression, and the second a string literal.

Unlike assert, _Static_assert is a keyword. A convenience macro static_assert is defined in <assert.h>.

```
#include <assert.h>

enum {N = 5};
_Static_assert(N == 5, "N does not equal 5");
static_assert(N > 10, "N is not greater than 10"); /* compiler error */
Version = C99
```

Prior to C11, there was no direct support for static assertions. However, in C99, static assertions could be emulated with macros that would trigger a compilation failure if the compile time condition was false. Unlike _Static_assert, the second parameter needs to be a proper token name so that a variable name can be created with it. If the assertion fails, the variable name is seen in the compiler error, since that variable was used in a syntactically incorrect array declaration.

```
#define STATIC_MSG(msg, l) STATIC_MSG2(msg, l)
#define STATIC_MSG2(msg,l) on_line_##l##_##msg
#define STATIC_ASSERT(x, msg) extern char STATIC_MSG(msg, __LINE__) [(x)?1:-1]

enum { N = 5 };
STATIC_ASSERT(N == 5, N_must_equal_5);
STATIC_ASSERT(N > 5, N_must_be_greater_than_5); /* compile error */
```

Before C99, you could not declare variables at arbitrary locations in a block, so you would have to be extremely cautious about using this macro, ensuring that it only appears where a variable declaration would be valid.

Section 33.3: Assert Error Messages

A trick exists that can display an error message along with an assertion. Normally, you would write code like this

```
void f(void *p)
{
    assert(p != NULL);
    /* more code */
}
```

If the assertion failed, an error message would resemble

Assertion failed: p != NULL, file main.c, line 5

However, you can use logical AND (&&) to give an error message as well

```
void f(void *p)
{
    assert(p != NULL && "function f: p cannot be NULL");
    /* more code */
}
```

Now, if the assertion fails, an error message will read something like this

断言失败 : p != NULL && "函数 f : p 不能为空", 文件 main.c, 第 5 行

之所以这样做有效，是因为字符串字面量总是被评估为非零（真）。向布尔表达式中添加 `&& 1` 不会产生影响。因此，添加 `&& "错误信息"` 也没有影响，唯一的区别是编译器会显示整个失败的表达式。

第 33.4 节：不可达代码的断言

在开发过程中，当必须防止某些代码路径被控制流访问时，你可以使用 `assert(0)` 用于指示此类条件是错误的：

```
switch (color) {
    case COLOR_RED:
    case COLOR_GREEN:
    case COLOR_BLUE:
        break;

    default:
        assert(0);
}
```

每当 `assert()` 宏的参数计算结果为假时，该宏会将诊断信息写入标准错误流，然后终止程序。该信息包括 `assert()` 语句所在的文件和行号，对调试非常有帮助。通过定义宏 `NDEBUG` 可以禁用断言。

另一种在发生错误时终止程序的方法是使用标准库函数 `exit`、`quick_exit` 或 `abort`。`exit` 和 `quick_exit` 接受一个参数，该参数可以传递回环境。`abort()`（因此包括 `assert`）可能会导致程序非常严重的终止，某些本应在执行结束时进行的清理操作可能不会被执行。

`assert()` 的主要优点是它会自动打印调试信息。调用 `abort()` 的优点是它不能像断言那样被禁用，但它可能不会显示任何调试信息。

在某些情况下，同时使用这两种结构可能是有益的：

```
if (color == COLOR_RED || color == COLOR_GREEN) {
    ...
} 否则如果 (color == COLOR_BLUE) {
    ...
} else {
    assert(0), abort();
}
```

当断言（asserts）被启用时，`assert()` 调用将打印调试信息并终止程序。执行永远不会到达 `abort()` 调用处。当断言被禁用时，`assert()` 调用不执行任何操作，`abort()` 会被调用。这确保程序在此错误条件下始终终止；启用和禁用断言仅影响是否打印调试输出。

你绝不应该在生产代码中保留这样的 `assert`，因为调试信息对最终用户没有帮助，且 `abort` 通常是过于严重的终止，会阻止为 `exit` 或 `quick_exit` 安装的清理处理程序运行。

Assertion failed: p != NULL && "function f: p cannot be NULL", file main.c, line 5

The reason as to why this works is that a string literal always evaluates to non-zero (true). Adding `&& 1` to a Boolean expression has no effect. Thus, adding `&& "error message"` has no effect either, except that the compiler will display the entire expression that failed.

Section 33.4: Assertion of Unreachable Code

During development, when certain code paths must be prevented from the reach of control flow, you may use `assert(0)` to indicate that such a condition is erroneous:

```
switch (color) {
    case COLOR_RED:
    case COLOR_GREEN:
    case COLOR_BLUE:
        break;

    default:
        assert(0);
}
```

Whenever the argument of the `assert()` macro evaluates false, the macro will write diagnostic information to the standard error stream and then abort the program. This information includes the file and line number of the `assert()` statement and can be very helpful in debugging. Asserts can be disabled by defining the macro `NDEBUG`.

Another way to terminate a program when an error occurs are with the standard library functions `exit`, `quick_exit` or `abort`. `exit` and `quick_exit` take an argument that can be passed back to your environment. `abort()` (and thus `assert`) can be a really severe termination of your program, and certain cleanups that would otherwise be performed at the end of the execution, may not be performed.

The primary advantage of `assert()` is that it automatically prints debugging information. Calling `abort()` has the advantage that it cannot be disabled like an `assert`, but it may not cause any debugging information to be displayed. In some situations, using both constructs together may be beneficial:

```
if (color == COLOR_RED || color == COLOR_GREEN) {
    ...
} else if (color == COLOR_BLUE) {
    ...
} else {
    assert(0), abort();
}
```

When asserts are *enabled*, the `assert()` call will print debug information and terminate the program. Execution never reaches the `abort()` call. When asserts are *disabled*, the `assert()` call does nothing and `abort()` is called. This ensures that the program *always* terminates for this error condition; enabling and disabling asserts only effects whether or not debug output is printed.

You should never leave such an `assert` in production code, because the debug information is not helpful for end users and because `abort` is generally a much too severe termination that inhibit cleanup handlers that are installed for `exit` or `quick_exit` to run.

第33.5节：前置条件和后置条件

断言的一个用例是前置条件和后置条件。这对于维护不变式和

Section 33.5: Precondition and Postcondition

One use case for assertion is precondition and postcondition. This can be very useful to maintain `invariant` and

契约式设计 (design by contract) 非常有用。例如，长度总是零或正数，因此该函数必须返回零或正数值。

```
#include <stdio.h>
/* 取消注释以禁用 `assert()` */
/* #define NDEBUG */
#include <assert.h>

int length2 (int *a, int count)
{
    int i, result = 0;

    /* 前置条件：*/
    /* NULL 是无效的向量 */
    assert (a != NULL);
    /* 维度数不能为负。*/
    assert (count >= 0);

    /* 计算 */
    for (i = 0; i < count; ++i)
    {
        result = result + (a[i] * a[i]);
    }

    /* 后置条件：*/
    /* 结果长度不能为负。 */
    assert (result >= 0);
    return result;
}

#define COUNT 3

int main (void)
{
    int a[COUNT] = {1, 2, 3};
    int *b = NULL;
    int r;
    r = length2 (a, COUNT);printf ("r
        = %i", r);    r = length2 (
    b, COUNT);printf ("r = %i", r);ret
    urn 0;
}
```

[design by contract](#). For a example a length is always zero or positive so this function must return a zero or positive value.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int length2 (int *a, int count)
{
    int i, result = 0;

    /* Precondition: */
    /* NULL is an invalid vector */
    assert (a != NULL);
    /* Number of dimensions can not be negative.*/
    assert (count >= 0);

    /* Calculation */
    for (i = 0; i < count; ++i)
    {
        result = result + (a[i] * a[i]);
    }

    /* Postcondition: */
    /* Resulting length can not be negative. */
    assert (result >= 0);
    return result;
}

#define COUNT 3

int main (void)
{
    int a[COUNT] = {1, 2, 3};
    int *b = NULL;
    int r;
    r = length2 (a, COUNT);
    printf ("r = %i\n", r);
    r = length2 (b, COUNT);
    printf ("r = %i\n", r);
    return 0;
}
```

第34章：泛型选择

参数	详情
generic-assoc-list	generic-association OR generic-assoc-list , generic-association
generic-association	类型名 : 赋值表达式 OR default : 赋值表达式

第34.1节：检查变量是否为某种限定类型

```
#include <stdio.h>

#define is_const_int(x) _Generic((&x), \
    const int *: "一个常量整型", \
    int *: "一个非常量整型", \
    default: "其他类型")

int main(void)
{
    const int i = 1;
    int j = 1;
    double k = 1.0;
    printf("i 是 %s", is_const_int(i));printf("j 是 %s", is_const_int(j));printf("k 是 %s", is_const_int(k));
}
```

输出：

```
i 是一个常量整型
j 是一个非常量整型
k 是其他类型
```

但是，如果类型泛型宏是这样实现的：

```
#define is_const_int(x) _Generic((x),
    const int: "一个常量整型",
    int: "一个非常量整型",
    default: "其他类型")
```

输出是：

```
i 是一个非常量整型
j 是一个非常量整型
k 是其他类型
```

这是因为在评估_Generic

主表达式的控制表达式时，所有类型限定符都会被去除。

第34.2节：基于多个参数的泛型选择

如果想对类型泛型表达式的多个参数进行选择，且所有相关类型都是算术类型，避免嵌套_Generic表达式的一个简单方法是使用控制表达式中参数的加法：

Chapter 34: Generic selection

Parameter	Details
generic-assoc-list	generic-association OR generic-assoc-list , generic-association
generic-association	type-name : assignment-expression OR default : assignment-expression

Section 34.1: Check whether a variable is of a certain qualified type

```
#include <stdio.h>

#define is_const_int(x) _Generic((&x), \
    const int *: "a const int", \
    int *: "a non-const int", \
    default: "of other type")

int main(void)
{
    const int i = 1;
    int j = 1;
    double k = 1.0;
    printf("i is %s\n", is_const_int(i));
    printf("j is %s\n", is_const_int(j));
    printf("k is %s\n", is_const_int(k));
}
```

Output:

```
i is a const int
j is a non-const int
k is of other type
```

However, if the type generic macro is implemented like this:

```
#define is_const_int(x) _Generic((x),
    const int: "a const int",
    int: "a non-const int",
    default: "of other type")
```

The output is:

```
i is a non-const int
j is a non-const int
k is of other type
```

This is because all type qualifiers are dropped for the evaluation of the controlling expression of a _Generic primary expression.

Section 34.2: Generic selection based on multiple arguments

If a selection on multiple arguments for a type generic expression is wanted, and all types in question are arithmetic types, an easy way to avoid nested _Generic expressions is to use addition of the parameters in the controlling expression:

```

int max_int(int, int);
unsigned max_unsigned(unsigned, unsigned);
double max_double(double, double);

#define MAX(X, Y) _Generic((X)+(Y),
    int:     max_int,
    unsigned: max_unsigned,
    default: max_double)
    ((X), (Y))

```

这里，控制表达式 $(X)+(Y)$ 仅根据其类型进行检查，而不进行求值。为了确定所选类型，会执行算术操作数的常规转换。

对于更复杂的情况，可以通过将多个参数嵌套在一起，基于操作符的多个参数进行选择。

此示例在四个外部实现的函数之间进行选择，这些函数接受两个 int 和/或 string 参数的组合，并返回它们的和。

```

int AddIntInt(int a, int b);
int AddIntStr(int a, const char* b);
int AddStrInt(const char* a, int b );
int AddStrStr(const char* a, const char* b);

#define AddStr(y) \
_Generic((y), \
    int: AddStrInt, \
    char*: AddStrStr, \
    const char*: AddStrStr )

#define AddInt(y) \
_Generic((y), \
    int: AddIntInt, \
    char*: AddIntStr, \
    const char*: AddIntStr )

#define Add(x, y) \
_Generic((x) , \
    int: AddInt(y) , \
    char*: AddStr(y) , \
    const char*: AddStr(y)) \
    ((x), (y))

int main( void )
{
    int result = 0;
result = Add( 100 , 999 );
    result = Add( 100 , "999" );
    result = Add( "100" , 999 );
    result = Add( "100" , "999" );

const int a = -123;
char b[] = "4321";
result = Add( a , b );

int c = 1;
const char d[] = "0";
result = Add( d , ++c );
}

```

尽管看起来参数 y 被评估了多次，但实际上并非如此。两个参数都只在宏 Add 的末尾被评估一次： (x, y) ，就像普通函数调用一样。

```

int max_int(int, int);
unsigned max_unsigned(unsigned, unsigned);
double max_double(double, double);

#define MAX(X, Y) _Generic((X)+(Y), \
    int:     max_int, \
    unsigned: max_unsigned, \
    default: max_double) \
    ((X), (Y))

```

Here, the controlling expression $(X)+(Y)$ is only inspected according to its type and not evaluated. The usual conversions for arithmetic operands are performed to determine the selected type.

For more complex situation, a selection can be made based on more than one argument to the operator, by nesting them together.

This example selects between four externally implemented functions, that take combinations of two int and/or string arguments, and return their sum.

```

int AddIntInt(int a, int b);
int AddIntStr(int a, const char* b);
int AddStrInt(const char* a, int b );
int AddStrStr(const char* a, const char* b);

#define AddStr(y) \
_Generic((y), \
    int: AddStrInt, \
    char*: AddStrStr, \
    const char*: AddStrStr )

#define AddInt(y) \
_Generic((y), \
    int: AddIntInt, \
    char*: AddIntStr, \
    const char*: AddIntStr )

#define Add(x, y) \
_Generic((x) , \
    int: AddInt(y) , \
    char*: AddStr(y) , \
    const char*: AddStr(y)) \
    ((x), (y))

int main( void )
{
    int result = 0;
result = Add( 100 , 999 );
    result = Add( 100 , "999" );
    result = Add( "100" , 999 );
    result = Add( "100" , "999" );

const int a = -123;
char b[] = "4321";
result = Add( a , b );

int c = 1;
const char d[] = "0";
result = Add( d , ++c );
}

```

Even though it appears as if argument y is evaluated more than once, it isn't 1. Both arguments are evaluated only once, at the end of macro Add: (x, y) , just like in an ordinary function call.

1 (引用自：ISO:IEC 9899:201X 6.5.1.1 泛型选择 3)

泛型选择的控制表达式不会被求值。

第34.3节：类型泛型打印宏

```
#include <stdio.h>

void print_int(int x) { printf("int: %d", x); } void print_dbl(double x) { printf("double: %g", x); } void print_default() { puts("unknown argument"); } #define print(X) _Generic((X), \
```

int: print_int, \
double: print_dbl, \
default: print_default)(X)

```
int main(void) {  
    print(42);  
    print(3.14);  
    print("hello, world");  
}
```

输出：

```
int: 42  
double: 3.14  
unknown argument
```

注意，如果类型既不是int也不是double，将会产生警告。要消除警告，可以将该类型添加到print(X)宏中。

1 (Quoted from: ISO:IEC 9899:201X 6.5.1.1 Generic selection 3)

The controlling expression of a generic selection is not evaluated.

Section 34.3: Type-generic printing macro

```
#include <stdio.h>

void print_int(int x) { printf("int: %d\n", x); }  
void print_dbl(double x) { printf("double: %g\n", x); }  
void print_default() { puts("unknown argument"); }

#define print(X) _Generic((X), \  
    int: print_int, \  
    double: print_dbl, \  
    default: print_default)(X)

int main(void) {  
    print(42);  
    print(3.14);  
    print("hello, world");  
}
```

Output:

```
int: 42  
double: 3.14  
unknown argument
```

Note that if the type is neither int nor double, a warning would be generated. To eliminate the warning, you can add that type to the print(X) macro.

第35章：X宏

X宏是一种基于预处理器的技术，用于最小化重复代码并维护数据/代码对应关系。通过使用一个主宏表示整个扩展组，该宏的替换文本由对内部宏的序列扩展组成，每个数据对应一个扩展，从而支持基于一组公共数据的多个不同宏扩展。内部宏传统上命名为X()，因此该技术得名。

第35.1节：X宏在printf中的简单应用

```
/* 定义一组预处理器标记以调用X */
#define X_123 X(1) X(2) X(3)

/* 定义要使用的X */
#define X(val) printf("X(%d) made this print", val);X_123

#undef X
/* 取消定义X是良好做法，便于以后重用 */
```

此示例将导致预处理器生成以下代码：

```
printf("X(%d) made this print", 1);printf("X(%d)
) made this print", 2);printf("X(%d) made thi
s print", 3);
```

第35.2节：扩展：将X宏作为参数传递

X宏方法可以通过将“X”宏的名称作为主宏的参数来稍作推广。这有助于避免宏名称冲突，并允许使用通用宏作为“X”宏。

与所有X宏一样，主宏表示一个项目列表，其意义特定于该宏。在这种变体中，该宏可能定义如下：

```
/* 声明项目列表 */
#define ITEM_LIST(X)
    X(item1)
    X(item2)
    X(item3)
/* 列表结束 */
```

然后可以生成如下代码来打印项目名称：

```
/* 定义要应用的宏 */
#define PRINTSTRING(value) printf( #value " ");

/* 将宏应用于项目列表 */
ITEM_LIST(PRINTSTRING)
```

这会展开成如下代码：

```
printf( "item1" ""); printf( "item2" ""); printf( "item3" "");
```

与标准的X宏不同，标准X宏中“X”名称是主宏的内置特性，而这种风格

Chapter 35: X-macros

X-macros are a preprocessor-based technique for minimizing repetitious code and maintaining data / code correspondences. Multiple distinct macro expansions based on a common set of data are supported by representing the whole group of expansions via a single master macro, with that macro's replacement text consisting of a sequence of expansions of an inner macro, one for each datum. The inner macro is traditionally named X(), hence the name of the technique.

Section 35.1: Trivial use of X-macros for printf

```
/* define a list of preprocessor tokens on which to call X */
#define X_123 X(1) X(2) X(3)

/* define X to use */
#define X(val) printf("X(%d) made this print\n", val);
X_123
#undef X
/* good practice to undef X to facilitate reuse later on */
```

This example will result in the preprocessor generating the following code:

```
printf("X(%d) made this print\n", 1);
printf("X(%d) made this print\n", 2);
printf("X(%d) made this print\n", 3);
```

Section 35.2: Extension: Give the X macro as an argument

The X-macro approach can be generalized a bit by making the name of the “X” macro an argument of the master macro. This has the advantages of helping to avoid macro name collisions and of allowing use of a general-purpose macro as the “X” macro.

As always with X macros, the master macro represents a list of items whose significance is specific to that macro. In this variation, such a macro might be defined like so:

```
/* declare list of items */
#define ITEM_LIST(X) \
    X(item1) \
    X(item2) \
    X(item3) \
/* end of list */
```

One might then generate code to print the item names like so:

```
/* define macro to apply */
#define PRINTSTRING(value) printf( #value "\n");

/* apply macro to the list of items */
ITEM_LIST(PRINTSTRING)
```

That expands to this code:

```
printf( "item1" "\n"); printf( "item2" "\n"); printf( "item3" "\n");
```

In contrast to standard X macros, where the “X” name is a built-in characteristic of the master macro, with this style

之后取消定义作为参数使用的宏（本例中的PRINTSTRING）可能是不必要的，甚至是不希望的。

it may be unnecessary or even undesirable to afterward undefine the macro used as the argument (PRINTSTRING in this example).

第35.3节：枚举值和标识符

```
/* 声明枚举项 */
#define FOREACH
    X(item1)
    X(item2)
    X(item3)
/* 列表结束 */

/* 定义枚举值 */
#define X(id) MyEnum_ ## id,
enum MyEnum { FOREACH };
#undef X

/* 将枚举值转换为其标识符 */
const char * enum2string(int enumValue)
{
    const char* stringValue = NULL;
#define X(id) if (enumValue == MyEnum_ ## id) stringValue = #id;
    FOREACH
#undef X
    return stringValue;
}
```

接下来，您可以在代码中使用枚举值，并轻松打印其标识符，方法如下：

```
printf("%s", enum2string(MyEnum_item2));
```

第35.4节：代码生成

X-宏可以用于代码生成，通过编写重复代码：遍历列表以执行某些任务，或声明一组常量、对象或函数。

这里我们使用X-宏声明一个包含4个命令的枚举及其名称字符串映射

然后我们可以打印枚举的字符串值。

```
/* 我们所有的命令 */
#define COMMANDS(OP) OP(Open) OP(Close) OP(Save) OP(Quit)

/* 生成枚举 Commands: {cmdOpen, cmdClose, cmdSave, cmdQuit, }; */
#define ENUM_NAME(name) cmd##name,
enum Commands {
COMMANDS(ENUM_NAME)
};
#undef ENUM_NAME

/* 生成字符串表 */
#define COMMAND_OP(name) #name,
const char* const commandNames[] = {
    COMMANDS(COMMAND_OP)
};
#undef COMMAND_OP

/* 以下代码打印 "Quit": */

```

Section 35.3: Enum Value and Identifier

```
/* declare items of the enum */
#define FOREACH \
    X(item1) \
    X(item2) \
    X(item3) \
/* end of list */

/* define the enum values */
#define X(id) MyEnum_ ## id,
enum MyEnum { FOREACH };
#undef X

/* convert an enum value to its identifier */
const char * enum2string(int enumValue)
{
    const char* stringValue = NULL;
#define X(id) if (enumValue == MyEnum_ ## id) stringValue = #id;
    FOREACH
#undef X
    return stringValue;
}
```

Next you can use the enumerated value in your code and easily print its identifier using :

```
printf("%s\n", enum2string(MyEnum_item2));
```

Section 35.4: Code generation

X-Macros can be used for code generation, by writing repetitive code: iterate over a list to do some tasks, or to declare a set of constants, objects or functions.

Here we use X-macros to declare an enum containing 4 commands and a map of their names as strings

Then we can print the string values of the enum.

```
/* All our commands */
#define COMMANDS(OP) OP(Open) OP(Close) OP(Save) OP(Quit)

/* generate the enum Commands: {cmdOpen, cmdClose, cmdSave, cmdQuit, }; */
#define ENUM_NAME(name) cmd##name,
enum Commands {
COMMANDS(ENUM_NAME)
};
#undef ENUM_NAME

/* generate the string table */
#define COMMAND_OP(name) #name,
const char* const commandNames[] = {
    COMMANDS(COMMAND_OP)
};
#undef COMMAND_OP

/* the following prints "Quit\n": */

```

```
printf("%s", commandNames[cmdQuit]());
```

同样，我们可以生成一个跳转表，通过枚举值调用函数。

这要求所有函数具有相同的签名。如果它们不带参数且返回 int，我们会将这部分放在包含枚举定义的头文件中：

```
/* 声明所有函数为 extern */
#define EXTERN_FUNC(name) extern int doCmd##name(void);
COMMANDS(EXTERN_FUNC)
#undef EXTERN_FUNC

/* 声明函数指针类型和跳转表 */
typedef int (*CommandFunc)(void);
extern CommandFunc commandJumpTable[];
```

以下所有内容可以在不同的编译单元中，只要上述部分作为头文件被包含：

```
/* 生成跳转表 */
#define FUNC_NAME(name) doCmd##name,
CommandFunc commandJumpTable[] = {
    COMMANDS(FUNC_NAME)
};
#undef FUNC_NAME

/* 这样调用保存命令：*/
int result = commandJumpTable[cmdSave]();

/* 在其他地方，我们需要命令的实现 */
int doCmdOpen(void) {/* 执行打开命令的代码 */}
int doCmdClose(void) {/* 执行关闭命令的代码 */}
int doCmdSave(void) {/* 执行保存命令的代码 */}
int doCmdQuit(void) {/* 执行退出命令的代码 */}
```

这种技术在实际代码中的一个例子是Chromium中的GPU命令调度。

```
printf("%s\n", commandNames[cmdQuit]());
```

Similarly, we can generate a jump table to call functions by the enum value.

This requires all functions to have the same signature. If they take no arguments and return an int, we would put this in a header with the enum definition:

```
/* declare all functions as extern */
#define EXTERN_FUNC(name) extern int doCmd##name(void);
COMMANDS(EXTERN_FUNC)
#undef EXTERN_FUNC

/* declare the function pointer type and the jump table */
typedef int (*CommandFunc)(void);
extern CommandFunc commandJumpTable[];
```

All of the following can be in different compilation units assuming the part above is included as a header:

```
/* generate the jump table */
#define FUNC_NAME(name) doCmd##name,
CommandFunc commandJumpTable[] = {
    COMMANDS(FUNC_NAME)
};
#undef FUNC_NAME

/* call the save command like this: */
int result = commandJumpTable[cmdSave]();

/* somewhere else, we need the implementations of the commands */
int doCmdOpen(void) /* code performing open command */
int doCmdClose(void) /* code performing close command */
int doCmdSave(void) /* code performing save command */
int doCmdQuit(void) /* code performing quit command */
```

An example of this technique being used in real code is for [GPU command dispatching in Chromium](#).

第36章：别名和有效类型

第36.1节：有效类型

数据对象的有效类型是最后与其关联的类型信息（如果有的话）。

```
// 一个普通变量，有效类型为uint32_t，且该类型永远不会改变
uint32_t a = 0.0;

// *pa的有效类型也是uint32_t，简单来说
// 因为*pa就是对象a
uint32_t* pa = &a;

// q指向的对象尚无有效类型
void* q = malloc(sizeof uint32_t);
// q指向的对象仍然没有有效类型，
// 因为还没有任何写操作
uint32_t* qb = q;
// *qb现在具有uint32_t的有效类型，因为写入了一个uint32_t值
*qb = 37;

// r指向的对象尚无有效类型，尽管
// 它已被初始化
void* r = calloc(1, sizeof uint32_t);
// r指向的对象仍然没有有效类型，
// 因为还没有对其进行写入或读取操作
uint32_t* rc = r;
// *rc现在具有uint32_t的有效类型，因为以该类型读取了值。
// 读取操作是有效的，因为我们使用了calloc。
// 现在由r指向的对象（与*rc相同）已经获得了有效类型，尽管我们没有改变它的值。
uint32_t c = *rc;

// 指向s的对象尚无有效类型。
void* s = malloc(sizeof uint32_t);
// 由于将一个uint32_t值复制到s指向的对象中，
// 该对象现在具有有效类型uint32_t。
memcpy(s, r, sizeof uint32_t);
```

注意，对于后者，我们甚至不需要有一个指向该对象的 `uint32_t*` 指针。仅仅复制了另一个 `uint32_t` 对象就足够了。

第36.2节：restrict 限定符

如果我们有两个相同类型的指针参数，编译器无法做出任何假设，必须始终假设对 `*e` 的修改可能会改变 `*f`：

```
void fun(float* e, float* f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?", a, b);}

float fval = 4;
float eval = 77;
fun(&eval, &fval);
```

Chapter 36: Aliasing and effective type

Section 36.1: Effective type

The *effective type* of a data object is the last type information that was associated with it, if any.

```
// a normal variable, effective type uint32_t, and this type never changes
uint32_t a = 0.0;

// effective type of *pa is uint32_t, too, simply
// because *pa is the object a
uint32_t* pa = &a;

// the object pointed to by q has no effective type, yet
void* q = malloc(sizeof uint32_t);
// the object pointed to by q still has no effective type,
// because nobody has written to it
uint32_t* qb = q;
// *qb now has effective type uint32_t because a uint32_t value was written
*qb = 37;

// the object pointed to by r has no effective type, yet, although
// it is initialized
void* r = calloc(1, sizeof uint32_t);
// the object pointed to by r still has no effective type,
// because nobody has written to or read from it
uint32_t* rc = r;
// *rc now has effective type uint32_t because a value is read
// from it with that type. The read operation is valid because we used calloc.
// Now the object pointed to by r (which is the same as *rc) has
// gained an effective type, although we didn't change its value.
uint32_t c = *rc;

// the object pointed to by s has no effective type, yet.
void* s = malloc(sizeof uint32_t);
// the object pointed to by s now has effective type uint32_t
// because an uint32_t value is copied into it.
memcpy(s, r, sizeof uint32_t);
```

Observe that for the latter, it was not necessary that we even have an `uint32_t*` pointer to that object. The fact that we have copied another `uint32_t` object is sufficient.

Section 36.2: restrict qualification

If we have two pointer arguments of the same type, the compiler can't make any assumption and will always have to assume that the change to `*e` may change `*f`:

```
void fun(float* e, float* f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);

float fval = 4;
float eval = 77;
fun(&eval, &fval);
```

一切顺利，类似于

is 4 equal to 4?

被打印出来。如果我们传入相同的指针，程序仍然会正确执行并打印

is 4 equal to 22?

如果我们通过某些外部信息知道 e 和 f 永远不会指向同一个数据对象，这可能会导致效率低下。我们可以通过给指针参数添加 restrict 限定符来反映这一知识：

```
void fan(float*restrict e, float*restrict f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?", a, b);}
```

那么编译器可以始终假设 e 和 f 指向不同的对象。

第36.3节：更改字节

一旦一个对象具有有效类型，除非通过另一种类型的指针访问的是字符类型，即char、signed char或unsigned char，否则不应尝试通过另一种类型的指针修改该对象。

```
#include <inttypes.h>
#include <stdio.h>

int main(void) {
    uint32_t a = 57;
    // 不兼容类型的转换需要强制类型转换！
    unsigned char* ap = (unsigned char*)&a;
    for (size_t i = 0; i < sizeof a; ++i) {
        /* 将a的每个字节设置为42 */
        ap[i] = 42;
    }
    printf("a now has value %" PRIu32 "", a);}

```

这是一个有效的程序，输出结果为

a now has value 707406378

这是可行的原因：

- 访问是通过类型unsigned char看到的单个字节进行的，因此每次修改都是明确定义的。
- 通过a和通过*ap对对象的两种视图是别名，但由于ap是指向字符类型的指针，严格别名规则不适用。因此编译器必须假设a的值可能在for循环中被更改。a的修改值必须由已更改的字节构造而成。
- a的类型uint32_t没有填充位。其表示的所有位都用于表示值，这里是

all goes well and something like

is 4 equal to 4?

is printed. If we pass the same pointer, the program will still do the right thing and print

is 4 equal to 22?

This can turn out to be inefficient, if we know by some outside information that e and f will never point to the same data object. We can reflect that knowledge by adding restrict qualifiers to the pointer parameters:

```
void fan(float*restrict e, float*restrict f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}
```

Then the compiler may always suppose that e and f point to different objects.

Section 36.3: Changing bytes

Once an object has an effective type, you should not attempt to modify it through a pointer of another type, unless that other type is a character type, `char`, `signed char` or `unsigned char`.

```
#include <inttypes.h>
#include <stdio.h>

int main(void) {
    uint32_t a = 57;
    // conversion from incompatible types needs a cast !
    unsigned char* ap = (unsigned char*)&a;
    for (size_t i = 0; i < sizeof a; ++i) {
        /* set each byte of a to 42 */
        ap[i] = 42;
    }
    printf("a now has value %" PRIu32 "\n", a);
}
```

This is a valid program that prints

a now has value 707406378

This works because:

- The access is made to the individual bytes seen with type `unsigned char` so each modification is well defined.
- The two views to the object, through a and through `*ap`, alias, but since ap is a pointer to a character type, the strict aliasing rule does not apply. Thus the compiler has to assume that the value of a may have been changed in the `for` loop. The modified value of a must be constructed from the bytes that have been changed.
- The type of a, `uint32_t` has no padding bits. All its bits of the representation count for the value, here

第36.4节：不能通过非字符类型访问字符类型

如果一个对象定义为静态、线程或自动存储期，并且它具有字符类型，即char、unsigned char或 signed char，则不得通过非字符类型访问它。在下面的示例中，一个char数组被重新解释为int类型，每次解引用int指针b时行为都是未定义的。

```
int main( void )
{
    char a[100];
    int* b = ( int* )&a;
    *b = 1;

    static char c[100];
    b = ( int* )&c;
    *b = 2;

_Thread_local char d[100];
    b = ( int* )&d;
    *b = 3;
}
```

这是未定义的，因为它违反了“有效类型”规则，任何具有有效类型的数据对象都不能通过非字符类型访问。由于这里的另一种类型是int，因此这是不允许的。

即使对齐和指针大小已知匹配，也不能免除此规则，行为仍然是未定义的。

这特别意味着，在标准C中，没有办法保留一个字符类型的缓冲区对象，该对象可以通过不同类型的指针使用，就像使用通过malloc或类似函数接收的缓冲区一样。

实现上述示例相同目标的正确方法是使用union。

```
typedef union bufType bufType;
union bufType {
    char c[sizeof(int[25])];
    int i[25];
};

int main( void )
{
    bufType a = { .c = { 0 } }; // 保留缓冲区并初始化
    int* b = a.i;             // 无需强制转换
    *b = 1;

    static bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 2;

_Thread_local bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 3;
}
```

这里，union 确保编译器从一开始就知道缓冲区可以通过不同的

Section 36.4: Character types cannot be accessed through non-character types

If an object is defined with static, thread, or automatic storage duration, and it has a character type, either: `char`, `unsigned char`, or `signed char`, it may not be accessed by a non-character type. In the below example a `char` array is reinterpreted as the type `int`, and the behavior is undefined on every dereference of the `int` pointer `b`.

```
int main( void )
{
    char a[100];
    int* b = ( int* )&a;
    *b = 1;

    static char c[100];
    b = ( int* )&c;
    *b = 2;

    _Thread_local char d[100];
    b = ( int* )&d;
    *b = 3;
}
```

This is undefined because it violates the "effective type" rule, no data object that has an effective type may be accessed through another type that is not a character type. Since the other type here is `int`, this is not allowed.

Even if alignment and pointer sizes would be known to fit, this would not exempt from this rule, behavior would still be undefined.

This means in particular that there is no way in standard C to reserve a buffer object of character type that can be used through pointers with different types, as you would use a buffer that was received by `malloc` or similar function.

A correct way to achieve the same goal as in the above example would be to use a `union`.

```
typedef union bufType bufType;
union bufType {
    char c[sizeof(int[25])];
    int i[25];
};

int main( void )
{
    bufType a = { .c = { 0 } }; // reserve a buffer and initialize
    int* b = a.i;             // no cast necessary
    *b = 1;

    static bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 2;

    _Thread_local bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 3;
}
```

Here, the `union` ensures that the compiler knows from the start that the buffer could be accessed through different

视图访问。这也有一个优点，即缓冲区现在有一个“视图”`a.i`，已经是 `int` 类型，无需指针转换。

第36.5节：违反严格别名规则

在下面的代码中，为简化起见，假设 `float` 和 `uint32_t` 大小相同。

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    float b = *f;
    print("%g 应该等于 %g", a, b);}
```

`u` 和 `f` 有不同的基本类型，因此编译器可以假设它们指向不同的对象。不存在 `*f` 在 `a` 和 `b` 两次初始化之间被修改的可能性，因此编译器可能将代码优化为等价于

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    print("%g 应该等于 %g", a, a);}
```

也就是说，`*f` 的第二次加载操作可以被完全优化掉。

如果我们“正常”调用这个函数

```
float fval = 4;
uint32_t uval = 77;
fun(&uval, &fval);
```

一切顺利，类似于

将打印出4等于4

但是如果我们作弊，传入相同的指针，经过转换后，

```
float fval = 4;
uint32_t* up = (uint32_t*)&fval;
fun(up, &fval);
```

我们就违反了严格别名规则。然后行为变得未定义。如果编译器优化了第二次访问，输出可能如上所示，或者完全不同，因此你的程序最终处于一个完全不可靠的状态。

views. This also has the advantage that now the buffer has a "view" `a.i` that already is of type `int` and no pointer conversion is needed.

Section 36.5: Violating the strict aliasing rules

In the following code let us assume for simplicity that `float` and `uint32_t` have the same size.

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    float b = *f;
    print("%g should equal %g\n", a, b);
}
```

`u` and `f` have different base type, and thus the compiler can assume that they point to different objects. There is no possibility that `*f` could have changed between the two initializations of `a` and `b`, and so the compiler may optimize the code to something equivalent to

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    print("%g should equal %g\n", a, a);
}
```

That is, the second load operation of `*f` can be optimized out completely.

If we call this function "normally"

```
float fval = 4;
uint32_t uval = 77;
fun(&uval, &fval);
```

all goes well and something like

4 should equal 4

is printed. But if we cheat and pass the same pointer, after converting it,

```
float fval = 4;
uint32_t* up = (uint32_t*)&fval;
fun(up, &fval);
```

we violate the strict aliasing rule. Then the behavior becomes undefined. The output could be as above, if the compiler had optimized the second access, or something completely different, and so your program ends up in a completely unreliable state.

第37章：编译

C语言传统上是一种编译语言（与解释型语言相对）。C标准定义了翻译阶段，应用这些阶段的产物是程序映像（或编译程序）。在C11标准中，这些阶段列在§5.1.1.2。

第37.1节：编译器

在C预处理器包含了所有头文件并展开了所有宏之后，编译器就可以编译程序了。它通过将C源代码转换成目标代码文件来完成这一步，目标代码文件以.o结尾，包含了源代码的二进制版本。不过，目标代码本身并不能直接执行。为了生成可执行文件，你还必须为所有通过#include引入的库函数添加代码（这不同于仅仅包含声明，后者是#include的作用）。这就是链接器的工作。

一般来说，调用C编译器的具体顺序很大程度上取决于你所使用的系统。这里我们使用的是GCC编译器，但需要注意的是还有许多其他编译器存在：

```
% gcc -Wall -c foo.c
```

% 是操作系统的命令提示符。这条命令告诉编译器对文件foo.c运行预处理器，然后将其编译成目标代码文件foo.o。选项-c表示将源代码文件编译成目标文件，但不调用链接器。这个-c选项在POSIX系统（如Linux或macOS）上可用；其他系统可能使用不同的语法。

如果你的整个程序都写在一个源代码文件中，你也可以这样做：

```
% gcc -Wall foo.c -o foo
```

这条命令告诉编译器对foo.c运行预处理器，编译它，然后链接生成一个名为foo的可执行文件。选项-o表示接下来的一词是二进制可执行文件（程序）的名称。如果你不指定-o（比如只输入gcc foo.c），出于历史原因，可执行文件将被命名为a.out。

一般来说，编译器在将.c文件转换成可执行文件时会经过四个步骤：

1. 预处理 - 对.c文件中的#include指令和#define宏进行文本展开
2. 编译 - 将程序转换成汇编代码（你可以通过添加-S选项在这一步停止编译器选项）
3. assembly - 将汇编代码转换为机器码
4. linkage - 将目标代码链接到外部库以创建可执行文件

另请注意，我们使用的编译器名称是GCC，根据上下文，它既代表“GNU C编译器”，也代表“GNU编译器集合”。其他C编译器也存在。对于类Unix操作系统，许多编译器的名称为cc，意为“C编译器”，通常是指向其他编译器的符号链接。在Linux系统上，cc通常是GCC的别名。在macOS或OS-X上，它指向clang。

POSIX标准目前规定c99作为C编译器的名称—默认支持C99标准。早期版本的POSIX规定c89作为编译器。POSIX还规定该编译器必须支持我们上面使用的-c和-o选项。

注意：-Wall选项出现在两个gcc示例中，告诉编译器打印有关可疑结构的警告，强烈推荐使用。添加其他warning options（例如-Wextra）也是个好主意。

Chapter 37: Compilation

The C language is traditionally a compiled language (as opposed to interpreted). The C Standard defines **translation phases**, and the product of applying them is a program image (or compiled program). In [c11](#), the phases are listed in §5.1.1.2.

Section 37.1: The Compiler

After the C pre-processor has included all the header files and expanded all macros, the compiler can compile the program. It does this by turning the C source code into an object code file, which is a file ending in .o which contains the binary version of the source code. Object code is not directly executable, though. In order to make an executable, you also have to add code for all of the library functions that were #included into the file (this is not the same as including the declarations, which is what #include does). This is the job of the linker.

In general, the exact sequence how to invoke a C compiler depends much on the system that you are using. Here we are using the GCC compiler, though it should be noted that many more compilers exist:

```
% gcc -Wall -c foo.c
```

% is the OS' command prompt. This tells the compiler to run the pre-processor on the file foo.c and then compile it into the object code file foo.o. The -c option means to compile the source code file into an object file but not to invoke the linker. This option -c is available on POSIX systems, such as Linux or macOS; other systems may use different syntax.

If your entire program is in one source code file, you can instead do this:

```
% gcc -Wall foo.c -o foo
```

This tells the compiler to run the pre-processor on foo.c, compile it and then link it to create an executable called foo. The -o option states that the next word on the line is the name of the binary executable file (program). If you don't specify the -o, (if you just type gcc foo.c), the executable will be named a.out for historical reasons.

In general the compiler takes four steps when converting a .c file into an executable:

1. **pre-processing** - textually expands #include directives and #define macros in your .c file
2. **compilation** - converts the program into assembly (you can stop the compiler at this step by adding the -S option)
3. **assembly** - converts the assembly into machine code
4. **linkage** - links the object code to external libraries to create an executable

Note also that the name of the compiler we are using is GCC, which stands for both "GNU C compiler" and "GNU compiler collection", depending on context. Other C compilers exist. For Unix-like operating systems, many of them have the name cc, for "C compiler", which is often a symbolic link to some other compiler. On Linux systems, cc is often an alias for GCC. On macOS or OS-X, it points to clang.

The POSIX standards currently mandates [c99](#) as the name of a C compiler — it supports the C99 standard by default. Earlier versions of POSIX mandated [c89](#) as the compiler. POSIX also mandates that this compiler understands the options -c and -o that we used above.

Note: The -Wall option present in both gcc examples tells the compiler to print warnings about questionable constructions, which is strongly recommended. It is also a good idea to add other [warning options](#), e.g. -Wextra.

第37.2节：文件类型

编译C程序需要处理五种类型的文件：

- 源文件：这些文件包含函数定义，按惯例文件名以.c结尾。注意：

.cc和.cpp是C++文件；不是C文件。

例如, foo.c

- 头文件：这些文件包含函数原型和各种预处理器语句（见下文）。

它们用于允许源代码文件访问外部定义的函数。头文件按惯例以.h结尾。

例如, foo.h

- 目标文件：这些文件是编译器的输出。它们包含函数定义，位于

二进制形式，但它们本身不可执行。目标文件通常以.o结尾，尽管在某些操作系统（例如Windows、MS-DOS）中，它们通常以.obj结尾。

例如, foo.o到foo.obj

- 二进制可执行文件：这些是由称为“链接器”的程序生成的输出。链接器将

将多个目标文件链接在一起生成一个可以直接执行的二进制文件。二进制可执行文件在Unix操作系统上没有特殊的后缀，尽管它们通常以.exe结尾于Windows系统上。

例如, foo foo.exe

- 库：库是已编译的二进制文件，但本身不是可执行文件（即，没有main()函数）

在库中）。库包含可能被多个程序使用的函数。库应随附头文件，头文件中包含库中所有函数的原型；任何使用该库的源文件都应引用这些头文件（例如#include <library.h>）。然后链接器需要引用该库，以便程序能够成功编译。库有两种类型：静态库和动态库。

- 静态库：静态库（POSIX系统中为.a文件，Windows中为.lib文件——不要与DLL导入库文件混淆，后者也使用.lib扩展名）是静态地构建到程序中的。静态库的优点是程序确切知道使用的是哪个版本的库。另一方面，可执行文件的体积较大，因为所有使用的库函数都被包含在内。

例如, libfoo.a foo.lib

- 动态库：动态库（大多数POSIX系统中为.so文件，OSX中为.dylib文件，Windows中为.dll文件）是在程序运行时动态链接的。这些有时也被称为共享库，因为一个库映像可以被多个程序共享。动态库的优点是如果多个应用程序使用该库，则占用的磁盘空间更少。此外，它们允许库的更新（修复漏洞）而无需重新构建可执行文件。

例如, foo.so foo.dylib foo.dll

第37.3节：链接器

链接器的工作是将一堆目标文件 (.o文件) 链接成二进制可执行文件。链接过程主要涉及将符号地址解析为数值地址。链接过程的结果通常是一个可执行程序。

在链接过程中，链接器会拾取命令行中指定的所有目标模块，在前面添加一些系统特定的启动代码，并尝试将目标模块中的所有外部引用与其他目标文件中的外部定义进行匹配（目标文件可以直接在命令行中指定，也可以通过库隐式添加）。然后，它会为目标文件分配加载地址，即指定代码和数据

Section 37.2: File Types

Compiling C programs requires you to work with five kinds of files:

1. **Source files:** These files contain function definitions, and have names which end in .c by convention. Note:

.cc and .cpp are C++ files; not C files.

e.g., foo.c

2. **Header files:** These files contain function prototypes and various pre-processor statements (see below).

They are used to allow source code files to access externally-defined functions. Header files end in .h by convention.

e.g., foo.h

3. **Object files:** These files are produced as the output of the compiler. They consist of function definitions in

binary form, but they are not executable by themselves. Object files end in .o by convention, although on some operating systems (e.g. Windows, MS-DOS), they often end in .obj.

e.g., foo.o foo.obj

4. **Binary executables:** These are produced as the output of a program called a "linker". The linker links

together a number of object files to produce a binary file which can be directly executed. Binary executables have no special suffix on Unix operating systems, although they generally end in .exe on Windows.

e.g., foo foo.exe

5. **Libraries:** A library is a compiled binary but is not in itself an executable (i.e., there is no main() function in a library). A library contains functions that may be used by more than one program. A library should ship

with header files which contain prototypes for all functions in the library; these header files should be referenced (e.g; #include <library.h>) in any source file that uses the library. The linker then needs to be referred to the library so the program can successfully compiled. There are two types of libraries: static and dynamic.

- **Static library:** A static library (.a files for POSIX systems and .lib files for Windows — not to be confused with [DLL import library files](#), which also use the .lib extension) is statically built into the program. Static libraries have the advantage that the program knows exactly which version of a library is used. On the other hand, the sizes of executables are bigger as all used library functions are included.

e.g., libfoo.a foo.lib

- **Dynamic library:** A dynamic library (.so files for most POSIX systems, .dylib for OSX and .dll files for Windows) is dynamically linked at runtime by the program. These are also sometimes referred to as shared libraries because one library image can be shared by many programs. Dynamic libraries have the advantage of taking up less disk space if more than one application is using the library. Also, they allow library updates (bug fixes) without having to rebuild executables.

e.g., foo.so foo.dylib foo.dll

Section 37.3: The Linker

The job of the linker is to link together a bunch of object files (.o files) into a binary executable. The process of linking mainly involves *resolving symbolic addresses to numerical addresses*. The result of the link process is normally an executable program.

During the link process, the linker will pick up all the object modules specified on the command line, add some system-specific *startup code* in front and try to resolve all *external* references in the object module with *external definitions* in other object files (object files can be specified directly on the command line or may implicitly be added through libraries). It will then assign *load addresses* for the object files, that is, it specifies where the code and data

在最终程序的地址空间中将放置的位置。一旦获得加载地址，它就可以将目标代码中的所有符号地址替换为目标地址空间中的“真实”数值地址。程序现在已准备好执行。

这包括编译器从你的源代码文件创建的目标文件，以及为你预编译并收集到库文件中的目标文件。这些文件的名称以.a或.so结尾，通常你不需要了解它们，因为链接器知道它们的大部分位置，并会根据需要自动链接它们。

链接器的隐式调用

与预处理器类似，链接器是一个独立的程序，通常称为ld（例如，Linux使用collect2）。同样像预处理器，当你使用编译器时，链接器会自动为你调用。因此，使用链接器的正常方式如下：

```
% gcc foo.o bar.o baz.o -o myprog
```

这行命令告诉编译器将三个目标文件（foo.o、bar.o和baz.o）链接成一个名为myprog的二进制可执行文件。现在你有了一个名为myprog的文件，可以运行它，并且希望它能做一些很酷和/或有用的事情。

链接器的显式调用

可以直接调用链接器，但这通常不建议，而且通常非常依赖平台。也就是说，在Linux上有效的选项不一定适用于Solaris、AIX、macOS、Windows，其他平台亦是如此。如果你使用GCC，可以使用gcc -v来查看为你执行了什么。

链接器的选项

链接器也接受一些参数来修改其行为。以下命令会告诉gcc链接foo.o和bar.o，同时还包含 ncurses库。

```
% gcc foo.o bar.o -o foo -lncurses
```

这实际上（或多或少）等同于

```
% gcc foo.o bar.o /usr/lib/libcurses.so -o foo
```

（虽然 libcurses.so 也可能是 libcurses.a，它只是用 tar 创建的一个归档文件）。注意，你应该在目标文件之后列出库文件（无论是通过路径名还是通过 -lname 选项）。对于静态库，指定的顺序很重要；而对于共享库，顺序通常无关紧要。

注意，在许多系统上，如果你使用数学函数（来自 <math.h>），你需要指定 -lm 来加载数学库——但 Mac OS X 和 macOS Sierra 不需要这样做。还有一些库在 Linux 和其他 Unix 系统上是独立的库，但在 macOS 上不是——例如 POSIX 线程、POSIX 实时和网络库。因此，链接过程在不同平台之间有所不同。

其他编译选项

这就是开始编译你自己的 C 程序所需了解的全部内容。通常，我们还建议你使用 -Wall 命令行选项：

```
% gcc -Wall -c foo.cc
```

will end up in the address space of the finished program. Once it's got the load addresses, it can replace all the symbolic addresses in the object code with "real", numerical addresses in the target's address space. The program is ready to be executed now.

This includes both the object files that the compiler created from your source code files as well as object files that have been pre-compiled for you and collected into library files. These files have names which end in .a or .so, and you normally don't need to know about them, as the linker knows where most of them are located and will link them in automatically as needed.

Implicit invocation of the linker

Like the pre-processor, the linker is a separate program, often called ld (but Linux uses collect2, for example). Also like the pre-processor, the linker is invoked automatically for you when you use the compiler. Thus, the normal way of using the linker is as follows:

```
% gcc foo.o bar.o baz.o -o myprog
```

This line tells the compiler to link together three object files (foo.o, bar.o, and baz.o) into a binary executable file named myprog. Now you have a file called myprog that you can run and which will hopefully do something cool and/or useful.

Explicit invocation of the linker

It is possible to invoke the linker directly, but this is seldom advisable, and is typically very platform-specific. That is, options that work on Linux won't necessarily work on Solaris, AIX, macOS, Windows, and similarly for any other platform. If you work with GCC, you can use gcc -v to see what is executed on your behalf.

Options for the linker

The linker also takes some arguments to modify its behavior. The following command would tell gcc to link foo.o and bar.o, but also include the ncurses library.

```
% gcc foo.o bar.o -o foo -lncurses
```

This is actually (more or less) equivalent to

```
% gcc foo.o bar.o /usr/lib/libcurses.so -o foo
```

(although libcurses.so could be libcurses.a, which is just an archive created with ar). Note that you should list the libraries (either by pathname or via -lname options) after the object files. With static libraries, the order that they are specified matters; often, with shared libraries, the order doesn't matter.

Note that on many systems, if you are using mathematical functions (from <math.h>), you need to specify -lm to load the mathematics library — but Mac OS X and macOS Sierra do not require this. There are other libraries that are separate libraries on Linux and other Unix systems, but not on macOS — POSIX threads, and POSIX realtime, and networking libraries are examples. Consequently, the linking process varies between platforms.

Other compilation options

This is all you need to know to begin compiling your own C programs. Generally, we also recommend that you use the -Wall command-line option:

```
% gcc -Wall -c foo.cc
```

`-Wall` 选项会让编译器对合法但可疑的代码结构发出警告，帮助你及早发现许多错误。

如果你希望编译器发出更多警告（包括声明了但未使用的变量、忘记返回值等），可以使用这组选项，因为尽管名字是 `-Wall`，但它并没有开启所有可能的警告：

```
% gcc -Wall -Wextra -Wfloat-equal -Wundef -Wcast-align -Wwrite-strings -Wlogical-op \
-Wmissing-declarations -Wredundant-decls -Wshadow ...
```

注意，`clang` 有一个选项 `-Weverything`，确实会开启 `clang` 中的所有警告。

第37.4节：预处理器

在C编译器开始编译源代码文件之前，文件会经过预处理阶段。这个阶段可以由一个独立的程序完成，也可以完全集成在一个可执行文件中。无论哪种情况，编译器都会在正式编译开始之前自动调用它。预处理阶段通过应用文本替换，将你的源代码转换成另一个源代码或翻译单元。你可以把它看作是一个“修改过的”或“扩展的”源代码。这个扩展后的源代码可能作为文件系统中的一个真实文件存在，或者可能只在内存中短暂存储，随后被进一步处理。

预处理命令以井号（#）开头。有几种预处理命令；其中两个最重要的是：

1. 定义：

`#define` 主要用于定义常量。例如，

```
#define BIGNUM 1000000
int a = BIGNUM;
```

变成

```
int a = 1000000;
```

`#define` 以这种方式使用，是为了避免在源代码文件的许多不同位置显式写出某个常量值。如果以后需要更改常量值，这一点非常重要；只需在`#define` 中更改一次，远比在代码中多个分散位置逐一更改要少出错得多。

因为`#define` 只是进行高级的搜索和替换，你也可以声明宏。例如：

```
#define ISTRUE(stm) do{stm = stm ? 1 : 0;}while(0)
// 在函数中：
a = x;
ISTRUE(a);
```

变为：

```
// 在函数中：
a = x;
do {
a = a ? 1 : 0;
} while(0);
```

The `-Wall` option causes the compiler to warn you about legal but dubious code constructs, and will help you catch a lot of bugs very early.

If you want the compiler to throw more warnings at you (including variables that are declared but not used, forgetting to return a value etc.), you can use this set of options, as `-Wall`, despite the name, doesn't turn *all of the possible warnings* on:

```
% gcc -Wall -Wextra -Wfloat-equal -Wundef -Wcast-align -Wwrite-strings -Wlogical-op \
> -Wmissing-declarations -Wredundant-decls -Wshadow ...
```

Note that `clang` has an option `-Weverything` which really does turn on all warnings in `clang`.

Section 37.4: The Preprocessor

Before the C compiler starts compiling a source code file, the file is processed in a preprocessing phase. This phase can be done by a separate program or be completely integrated in one executable. In any case, it is invoked automatically by the compiler before compilation proper begins. The preprocessing phase converts your source code into another source code or translation unit by applying textual replacements. You can think of it as a “modified” or “expanded” source code. That expanded source may exist as a real file in the file system, or it may only be stored in memory for a short time before being processed further.

Preprocessor commands start with the pound sign (#). There are several preprocessor commands; two of the most important are:

1. Defines:

`#define` is mainly used to define constants. For instance,

```
#define BIGNUM 1000000
int a = BIGNUM;
```

becomes

```
int a = 1000000;
```

`#define` is used in this way so as to avoid having to explicitly write out some constant value in many different places in a source code file. This is important in case you need to change the constant value later on; it's much less bug-prone to change it once, in the `#define`, than to have to change it in multiple places scattered all over the code.

Because `#define` just does advanced search and replace, you can also declare macros. For instance:

```
#define ISTRUE(stm) do{stm = stm ? 1 : 0;}while(0)
// in the function:
a = x;
ISTRUE(a);
```

becomes:

```
// in the function:
a = x;
do {
a = a ? 1 : 0;
} while(0);
```

初步估计，这种效果大致与内联函数相同，但预处理器不会对#define宏进行类型检查。这是众所周知的易出错点，因此使用时必须非常小心。

还要注意，预处理器还会将注释替换为空格，如下所述。

2. 包含:

#include用于访问定义在源代码文件外部的函数定义。例如：

```
#include <stdio.h>
```

使预处理器将<stdio.h>的内容粘贴到源代码文件中该位置

#include语句在编译之前使用。#include几乎总是用来包含头文件，这些文件主要包含函数声明和#define语句。在这种情况下，我们使用#include以便能够使用诸如printf和scanf这样的函数，这些函数的声明位于文件中stdio.h。C编译器不允许你使用一个函数，除非该函数之前已经在该文件中声明或定义；因此，#include语句是重用之前编写的代码到你的C程序中的方式。

3. 逻辑操作：

```
#if defined A || defined B  
变量 = 另一个变量 + 1;  
#else  
变量 = 另一个变量 * 2;  
#endif
```

将被改为：

```
变量 = 另一个变量 + 1;
```

如果A或B之前在项目中的某处被定义过。如果不是这种情况，预处理器当然会执行以下操作：

```
变量 = 另一个变量 * 2;
```

这通常用于在不同系统上运行或在不同编译器上编译的代码。由于存在特定于编译器/系统的全局定义，你可以测试这些定义，并始终让编译器只使用它确定会编译的代码。

4.评论

预处理器将源文件中的所有注释替换为单个空格。注释由//开始直到行尾，或者由开头的/*和结尾的*/注释括号组合表示。

第37.5节：翻译阶段

根据C 2011标准，在§5.1.1.2 翻译阶段中，源代码到程序映像（例如可执行文件）的翻译被列为8个有序步骤。

At first approximation, this effect is roughly the same as with inline functions, but the preprocessor doesn't provide type checking for #define macros. This is well known to be error-prone and their use necessitates great caution.

Also note here, that the preprocessor would also replace comments with a blanks as explained below.

2. Includes:

#include是用于访问定义在源代码文件外部的函数定义。例如：

```
#include <stdio.h>
```

causes the preprocessor to paste the contents of <stdio.h> into the source code file at the location of the #include statement before it gets compiled. #include is almost always used to include header files, which are files which mainly contain function declarations and #define statements. In this case, we use #include in order to be able to use functions such as printf and scanf, whose declarations are located in the file stdio.h. C compilers do not allow you to use a function unless it has previously been declared or defined in that file; #include statements are thus the way to re-use previously-written code in your C programs.

3. Logic operations:

```
#if defined A || defined B  
variable = another_variable + 1;  
#else  
variable = another_variable * 2;  
#endif
```

will be changed to:

```
variable = another_variable + 1;
```

if A or B were defined somewhere in the project before. If this is not the case, of course the preprocessor will do this:

```
variable = another_variable * 2;
```

This is often used for code, that runs on different systems or compiles on different compilers. Since there are global defines, that are compiler/system specific you can test on those defines and always let the compiler just use the code he will compile for sure.

4. Comments

The Preprocessor replaces all comments in the source file by single spaces. Comments are indicated by // up to the end of the line, or a combination of opening /* and closing */ comment brackets.

Section 37.5: The Translation Phases

As of the C 2011 Standard, listed in §5.1.1.2 *Translation Phases*, the translation of source code to program image (e.g., the executable) are listed to occur in 8 ordered steps.

- 源文件输入被映射到源字符集（如有必要）。三字符序列（trigraphs）在此步骤中被替换。
- 续行（以\结尾的行）与下一行拼接。
- 源代码被解析为空白和预处理标记。
- 应用预处理器，执行指令，展开宏，并应用编译指示。每个通过#include引入的源文件都经历翻译阶段1到4（如有必要递归进行）。所有预处理相关的指令随后被删除。
- 字符常量和字符串字面量中的源字符集值被映射到执行字符集。
- 相邻的字符串字面量会被连接起来。
- 源代码被解析成标记，这些标记组成翻译单元。
- 外部引用被解析，程序映像被形成。

C 编译器的实现可能会将多个步骤合并在一起，但生成的映像仍必须表现得像上述步骤按顺序分别发生一样。

- The source file input is mapped to the source character set (if necessary). Trigraphs are replaced in this step.
- Continuation lines (lines that end with \) are spliced with the next line.
- The source code is parsed into whitespace and preprocessing tokens.
- The preprocessor is applied, which executes directives, expands macros, and applies pragmas. Each source file pulled in by #include undergoes translation phases 1 through 4 (recursively if necessary). All preprocessor related directives are then deleted.
- Source character set values in character constants and string literals are mapped to the execution character set.
- String literals adjacent to each other are concatenated.
- The source code is parsed into tokens, which comprise the translation unit.
- External references are resolved, and the program image is formed.

An implementation of a C compiler may combine several steps together, but the resulting image must still behave as if the above steps had occurred separately in the order listed above.

第38章：内联汇编

第38.1节：gcc 宏中的内联汇编

我们可以将汇编指令放入宏中，并像调用函数一样使用该宏。

```
#define mov(x,y) \
{ \
    __asm__ ("l.cmov %0,%1,%2" : "=r" (x) : "r" (y), "r" (0x0000000F)); \
}

/// 一些定义和赋值

unsigned char sbox[size][size];

/// 使用
mov(state[0][1], sbox[si][sj]);
```

在C代码中嵌入内联汇编指令可以提高程序的运行时间。这在时间关键的情况下非常有用，比如像AES这样的加密算法。例如，对于AES算法中需要的简单移位操作，我们可以用直接的Rotate Right汇编指令替代C语言的移位操作符`>>`。

在'AES256'的实现中，在'AddRoundKey()'函数里，我们有如下语句：

```
unsigned int w;           // 32位
unsigned char subkey[4]; // 8位, 4*8 = 32

subkey[0] = w >> 24;    // 保存8位, 最高有效字节, 最左边的8位组
subkey[1] = w >> 16;    // 保存8位, 从左数第二组8位
subkey[2] = w >> 8;     // 保存8位, 从右数第二组8位
subkey[3] = w;           // 保存8位, 最低有效字节, 最右边的8位组

/// subkey <- w
```

它们只是简单地将w的位值赋给了subkey数组。

我们可以用一个汇编的Rotate Right操作替换三个移位加赋值和一个赋值的C表达式。

```
__asm__ ("l.ror %0,%1,%2" : "=r" (* (unsigned int *) subkey) : "r" (w), "r" (0x10));
```

最终结果完全相同。

第38.2节：gcc基础汇编支持

gcc的基础汇编支持语法如下：

```
asm [ volatile ] ( AssemblerInstructions )
```

其中 `AssemblerInstructions` 是针对给定处理器的直接汇编代码。`volatile` 关键字是可选的且没有效果，因为 `gcc` 不会优化基本 `asm` 语句中的代码。`AssemblerInstructions` 可以包含多条汇编指令。如果你有一个必须存在于 C 函数外的 `asm` 例程，则使用基本 `asm` 语句。以下示例摘自 `GCC` 手册：

```
/* 注意此代码无法用 -masm=intel 编译 */
#define DebugBreak() asm("int $3")
```

Chapter 38: Inline assembly

Section 38.1: gcc Inline assembly in macros

We can put assembly instructions inside a macro and use the macro like you would call a function.

```
#define mov(x,y) \
{ \
    __asm__ ("l.cmov %0,%1,%2" : "=r" (x) : "r" (y), "r" (0x0000000F)); \
}

/// some definition and assignment
unsigned char sbox[size][size];
unsigned char sbox[size][size];

///Using
mov(state[0][1], sbox[si][sj]);
```

Using inline assembly instructions embedded in C code can improve the run time of a program. This is very helpful in time critical situations like cryptographic algorithms such as AES. For example, for a simple shift operation that is needed in the AES algorithm, we can substitute a direct Rotate Right assembly instruction with C shift operator `>>`.

In an implementation of 'AES256', in 'AddRoundKey()' function we have some statements like this:

```
unsigned int w;           // 32-bit
unsigned char subkey[4]; // 8-bit, 4*8 = 32

subkey[0] = w >> 24;    // hold 8 bit, MSB, leftmost group of 8-bits
subkey[1] = w >> 16;    // hold 8 bit, second group of 8-bit from left
subkey[2] = w >> 8;     // hold 8 bit, second group of 8-bit from right
subkey[3] = w;           // hold 8 bit, LSB, rightmost group of 8-bits

/// subkey <- w
```

They simply assign the bit value of `w` to `subkey` array.

We can change three shift + assign and one assign C expression with only one assembly Rotate Right operation.

```
__asm__ ("l.ror %0,%1,%2" : "=r" (* (unsigned int *) subkey) : "r" (w), "r" (0x10));
```

The final result is exactly same.

Section 38.2: gcc Basic asm support

Basic assembly support with `gcc` has the following syntax:

```
asm [ volatile ] ( AssemblerInstructions )
```

where `AssemblerInstructions` is the direct assembly code for the given processor. The `volatile` keyword is optional and has no effect as `gcc` does not optimize code within a basic `asm` statement. `AssemblerInstructions` can contain multiple assembly instructions. A basic `asm` statement is used if you have an `asm` routine that must exist outside of a C function. The following example is from the `GCC` manual:

```
/* Note that this code will not compile with -masm=intel */
#define DebugBreak() asm("int $3")
```

在此示例中，你可以在代码的其他地方使用 `DebugBreak()`，它将执行汇编指令 `int $3`。注意，尽管 `gcc` 不会修改基本 `asm` 语句中的任何代码，优化器仍可能移动连续的 `asm` 语句。如果你有多条必须按特定顺序执行的汇编指令，请将它们包含在一个 `asm` 语句中。

第 38.3 节：gcc 扩展 asm 支持

`gcc` 中扩展 `asm` 支持的语法如下：

```
asm [volatile] ( AssemblerTemplate
    : OutputOperands
    [ : InputOperands
    [ : Clobbers ] ])
```



```
asm [volatile] goto ( AssemblerTemplate
    :
    : 输入操作数
    : 被破坏的寄存器
    : 跳转标签)
```

其中 `AssemblerTemplate` 是汇编指令的模板，`OutputOperands` 是任何可能被汇编代码修改的 C 变量，`InputOperands` 是作为输入参数使用的任何 C 变量，`Clobbers` 是被汇编代码修改的寄存器列表，`GotoLabels` 是汇编代码中可能使用的任何 `goto` 语句标签。

扩展格式用于 C 函数内部，是内联汇编的更典型用法。下面是 Linux 内核中针对 ARM 处理器进行 16 位和 32 位字节交换的示例：

```
/* 来自 Linux 内核版本 4.6.4 中 arch/arm/include/asm/swab.h */
#ifndef __LINUX_ARM_ARCH__ >= 6

static inline __attribute_const__ __u32 __arch_swahb32(__u32 x)
{
    __asm__ ("rev16 %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swahb32 __arch_swahb32
#define __arch_swab16(x) ((__u16)__arch_swahb32(x))

static inline __attribute_const__ __u32 __arch_swab32(__u32 x)
{
    __asm__ ("rev %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swab32 __arch_swab32

#endif
```

每个 `asm` 段使用变量 `x` 作为其输入和输出参数。然后 C 函数返回处理后的结果。

使用扩展 `asm` 格式时，`gcc` 可能会按照优化 C 代码的相同规则优化 `asm` 块中的汇编指令。如果你希望 `asm` 段保持不变，请为 `asm` 段使用 `volatile` 关键字。

In this example, you could then use `DebugBreak()` in other places in your code and it will execute the assembly instruction `int $3`. Note that even though `gcc` will not modify any code in a basic `asm` statement, the optimizer may still move consecutive `asm` statements around. If you have multiple assembly instructions that must occur in a specific order, include them in one `asm` statement.

Section 38.3: gcc Extended asm support

Extended `asm` support in `gcc` has the following syntax:

```
asm [volatile] ( AssemblerTemplate
    : OutputOperands
    [ : InputOperands
    [ : Clobbers ] ])
```



```
asm [volatile] goto ( AssemblerTemplate
    :
    : InputOperands
    : Clobbers
    : GotoLabels)
```

where `AssemblerTemplate` is the template for the assembler instruction, `OutputOperands` are any C variables that can be modified by the assembly code, `InputOperands` are any C variables used as input parameters, `Clobbers` are a list or registers that are modified by the assembly code, and `GotoLabels` are any `goto` statement labels that may be used in the assembly code.

The extended format is used within C functions and is the more typical usage of inline assembly. Below is an example from the Linux kernel for byte swapping 16-bit and 32-bit numbers for an ARM processor:

```
/* From arch/arm/include/asm/swab.h in Linux kernel version 4.6.4 */
#ifndef __LINUX_ARM_ARCH__ >= 6

static inline __attribute_const__ __u32 __arch_swahb32(__u32 x)
{
    __asm__ ("rev16 %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swahb32 __arch_swahb32
#define __arch_swab16(x) ((__u16)__arch_swahb32(x))

static inline __attribute_const__ __u32 __arch_swab32(__u32 x)
{
    __asm__ ("rev %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swab32 __arch_swab32

#endif
```

Each `asm` section uses the variable `x` as its input and output parameter. The C function then returns the manipulated result.

With the extended `asm` format, `gcc` may optimize the assembly instructions in an `asm` block following the same rules it uses for optimizing C code. If you want your `asm` section to remain untouched, use the `volatile` keyword for the `asm` section.

第39章：标识符作用域

第39.1节：函数原型作用域

```
#include <stdio.h>

/* 参数名 apple 具有函数原型作用域。这些名称在原型本身之外没有意义。下面演示了这一点
 */
int test_function(int apple);

int main(void)
{
    int orange = 5;

orange = test_function(orange);printf("%d\r
", orange); //orange = 6

    return 0;
}

int test_function(int fruit)
{
fruit += 1;
    return fruit;
}
```

请注意，如果在原型中引入类型名称，会出现令人困惑的错误信息：

```
int function(struct whatever *arg);

struct whatever
{
    int a;
    // ...
};

int function(struct whatever *arg)
{
    return arg->a;
}
```

使用 GCC 6.3.0，该代码（源文件 dc11.c）产生：

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -c dc11.c
dc11.c:1:25: 错误: 'struct whatever' 在参数列表中声明，将不会在此定义或声明之外可见 [-Werror]
    int function(struct whatever *arg);
                           ^~~~~~
dc11.c:9:9: 错误: 'function' 的类型冲突
    int function(struct whatever *arg);
           ^~~~~~
dc11.c:1:9: 注释: 'function' 的先前声明在此
    int function(struct whatever *arg);
           ^~~~~~
cc1: 所有警告均视为错误
$
```

Chapter 39: Identifier Scope

Section 39.1: Function Prototype Scope

```
#include <stdio.h>

/* The parameter name, apple, has function prototype scope. These names
are not significant outside the prototype itself. This is demonstrated
below. */

int test_function(int apple);

int main(void)
{
    int orange = 5;

orange = test_function(orange);
printf("%d\r\n", orange); //orange = 6

    return 0;
}

int test_function(int fruit)
{
fruit += 1;
    return fruit;
}
```

Note that you get puzzling error messages if you introduce a type name in a prototype:

```
int function(struct whatever *arg);

struct whatever
{
    int a;
    // ...
};

int function(struct whatever *arg)
{
    return arg->a;
}
```

With GCC 6.3.0, this code (source file dc11.c) produces:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -c dc11.c
dc11.c:1:25: error: 'struct whatever' declared inside parameter list will not be visible outside of
this definition or declaration [-Werror]
    int function(struct whatever *arg);
                           ^~~~~~
dc11.c:9:9: error: conflicting types for 'function'
    int function(struct whatever *arg)
           ^~~~~~
dc11.c:1:9: note: previous declaration of 'function' was here
    int function(struct whatever *arg);
           ^~~~~~
cc1: all warnings being treated as errors
$
```

将结构体定义放在函数声明之前，或者在函数声明之前添加一行`struct whatever;`，就没有问题。你不应该在函数原型中引入新的类型名，因为无法使用该类型，因此也无法定义或使用该函数。

第39.2节：块作用域

如果标识符对应的声明出现在块内（函数定义中的参数声明适用），则该标识符具有块作用域。作用域在对应块结束时终止。

同一作用域内不能有相同标识符的不同实体，但作用域可以重叠。在作用域重叠的情况下，唯一可见的是声明在最内层作用域中的那个。

```
#include <stdio.h>

void test(int bar)          // bar的作用域为test函数块
{
    int foo = 5;            // foo的作用域为test函数块
    {
        int bar = 10;        // bar的作用域为内层块，这与之前的
        // test:bar声明重叠，并且隐藏了test:bar
        printf("%d %d", foo, bar); // 5 10
    }                      // 内层bar的作用域结束printf("%d %d", foo,
    bar);      // 5 5, 此处bar为test:bar}
    / test:foo和test:bar的作用域结束int main(void)
    {

        int foo = 3;        // foo 的作用域是主函数块
        printf("%d", foo); // 3    test(
5);
        printf("%d", foo); // 3return 0;
    }                      // main作用域结束 :foo
```

Place the structure definition before the function declaration, or add `struct whatever;` as a line before the function declaration, and there is no problem. You should not introduce new type names in a function prototype because there's no way to use that type, and hence no way to define or use that function.

Section 39.2: Block Scope

An identifier has block scope if its corresponding declaration appears inside a block (parameter declaration in function definition apply). The scope ends at the end of the corresponding block.

No different entities with the same identifier can have the same scope, but scopes may overlap. In case of overlapping scopes the only visible one is the one declared in the innermost scope.

```
#include <stdio.h>

void test(int bar)          // bar has scope test function block
{
    int foo = 5;            // foo has scope test function block
    {
        int bar = 10;        // bar has scope inner block, this overlaps with previous
        // test:bar declaration, and it hides test:bar
        printf("%d %d\n", foo, bar); // 5 10
    }                      // end of scope for inner bar
    printf("%d %d\n", foo, bar); // 5 5, here bar is test:bar
}                          // end of scope for test:foo and test:bar

int main(void)
{
    int foo = 3;            // foo has scope main function block
    printf("%d\n", foo); // 3
    test(5);
    printf("%d\n", foo); // 3
    return 0;
}                          // end of scope for main:foo
```

第39.3节：文件作用域

```
#include <stdio.h>

/* 标识符foo在所有代码块之外声明。
它可以在声明之后直到翻译单元结束的任何地方使用。 */
static int foo;

void test_function(void)
{
    foo += 2;
}

int main(void)
{
    foo = 1;

    test_function();printf("%d
\r", foo); //foo = 3;

    return 0;
}
```

Section 39.3: File Scope

```
#include <stdio.h>

/* The identifier, foo, is declared outside all blocks.
It can be used anywhere after the declaration until the end of
the translation unit. */
static int foo;

void test_function(void)
{
    foo += 2;
}

int main(void)
{
    foo = 1;

    test_function();
    printf("%d\r\n", foo); //foo = 3;

    return 0;
}
```

第39.4节：函数作用域

函数作用域是标签的特殊作用域。这是由于它们的特殊属性。一个标签在其定义的整个函数中都是可见的，并且可以从同一函数中的任何点跳转（使用指令goto标签）到它。虽然不常用，以下示例说明了这一点：

```
#include <stdio.h>

int main(int argc,char *argv[]) {
    int a = 0;
    goto INSIDE;
OUTSIDE:
    if (a!=0) {
        int i=0;
INSIDE:
        printf("a=%d",a);goto OUTSIDE;
    }
}
```

INSIDE看似定义在if块内，就像变量i的作用域是该块一样，但事实并非如此。它在整个函数中都是可见的，正如指令goto INSIDE;所示。因此，在单个函数中不能有两个相同标识符的标签。

一种可能的用法是以下模式，用于实现对分配资源的正确复杂清理：

```
#include <stdlib.h>
#include <stdio.h>

void a_function(void) {
    double* a = malloc(sizeof(double[34]));
    if (!a) {
        fprintf(stderr,"can't allocate");return;
        /* 如果a为null，释放a没有意义 */
    }

    FILE* b = fopen("some_file", "r");
    if (!b) {
        fprintf(stderr,"无法打开");goto CLEANUP1;
        /* 释放 a；关闭 b 没意义 */
    }

    /* 做一些合理的操作 */
    if (error) {
        fprintf(stderr,"出现错误");goto CLEANUP2;
        /* 释放 a 并关闭 b 以防止泄漏 */
    }

    /* 再做一些其他操作 */
CLEANUP2:
    close(b);
CLEANUP1:
    free(a);
}
```

标签如CLEANUP1和CLEANUP2是特殊标识符，其行为不同于其他所有标识符。它们在函数内部任何地方都是可见的，甚至在执行标记语句之前的地方，或者在如果没有执行任何goto语句时根本无法到达的地方。标签通常用小写字母书写，而不是大写字母。

Section 39.4: Function scope

Function scope is the special scope for **labels**. This is due to their unusual property. A **label** is visible through the entire function it is defined and one can jump (using instruction `goto label`) to it from any point in the same function. While not useful, the following example illustrate the point:

```
#include <stdio.h>

int main(int argc,char *argv[]) {
    int a = 0;
    goto INSIDE;
OUTSIDE:
    if (a!=0) {
        int i=0;
INSIDE:
        printf("a=%d\n",a);
        goto OUTSIDE;
    }
}
```

INSIDE may seem defined *inside* the if block, as it is the case for i which scope is the block, but it is not. It is visible in the whole function as the instruction `goto INSIDE;` illustrates. Thus there can't be two labels with the same identifier in a single function.

A possible usage is the following pattern to realize correct complex cleanups of allocated resources:

```
#include <stdlib.h>
#include <stdio.h>

void a_function(void) {
    double* a = malloc(sizeof(double[34]));
    if (!a) {
        fprintf(stderr,"can't allocate\n");
        return; /* No point in freeing a if it is null */
    }

    FILE* b = fopen("some_file", "r");
    if (!b) {
        fprintf(stderr,"can't open\n");
        goto CLEANUP1; /* Free a; no point in closing b */
    }

    /* do something reasonable */
    if (error) {
        fprintf(stderr,"something's wrong\n");
        goto CLEANUP2; /* Free a and close b to prevent leaks */
    }

    /* do yet something else */
CLEANUP2:
    close(b);
CLEANUP1:
    free(a);
}
```

Labels such as CLEANUP1 and CLEANUP2 are special identifiers that behave differently from all other identifiers. They are visible from everywhere inside the function, even in places that are executed before the labeled statement, or even in places that could never be reached if none of the `goto` is executed. Labels are often written in lower-case rather than upper-case.

第40章：隐式和显式转换

第40.1节：函数调用中的整数转换

鉴于函数具有适当的原型，根据整数转换规则（C11 6.3.1.3），整数在调用函数时会被扩展。

6.3.1.3 有符号和无符号整数

当一个整数类型的值被转换为除 _Bool 以外的另一整数类型时，如果该值可以被新类型表示，则值保持不变。

否则，如果新类型是无符号的，则通过反复加上或减去新类型可表示的最大值加一，直到该值落入新类型的范围内。

否则，新类型是有符号的且该值无法被表示；结果要么是实现定义的，要么会引发实现定义的信号。

通常不应将宽的有符号类型截断为较窄的有符号类型，因为显然值无法适配，且这种操作没有明确的意义。上述C标准将这些情况定义为“实现定义”，即它们不可移植。

以下示例假设 int 为 32 位宽。

```
#include <stdio.h>
#include <stdint.h>

void param_u8(uint8_t val) {
    printf("%s val 的值是 %d", __func__, val); /* val 被提升为 int 类型 */
}

void param_u16(uint16_t val) {
    printf("%s val 的值是 %d", __func__, val); /* val 被提升为 int 类型 */
}

void param_u32(uint32_t val) {
    printf("%s val is %u", __func__, val); /* 这里 val 适合无符号类型 */
}

void param_u64(uint64_t val) {
    printf("%s val is " PRI64u "", __func__, val); /* 使用格式字符串修正 */
}

void param_s8(int8_t val) {
    printf("%s val 的值是 %d", __func__, val); /* val 被提升为 int 类型 */
}

void param_s16(int16_t val) {
    printf("%s val 的值是 %d", __func__, val); /* val 被提升为 int 类型 */
}

void param_s32(int32_t val) {
    printf("%s val is %d", __func__, val); /* val 与 int 宽度相同 */
}
```

Chapter 40: Implicit and Explicit Conversions

Section 40.1: Integer Conversions in Function Calls

Given that the function has a proper prototype, integers are widened for calls to functions according to the rules of integer conversion, C11 6.3.1.3.

6.3.1.3 Signed and unsigned integers

When a value with integer type is converted to another integer type other than _Bool, if the value can be represented by the new type, it is unchanged.

Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.

Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

Usually you should not truncate a wide signed type to a narrower signed type, because obviously the values can't fit and there is no clear meaning that this should have. The C standard cited above defines these cases to be "implementation-defined", that is, they are not portable.

The following example supposes that int is 32 bit wide.

```
#include <stdio.h>
#include <stdint.h>

void param_u8(uint8_t val) {
    printf("%s val is %d\n", __func__, val); /* val is promoted to int */
}

void param_u16(uint16_t val) {
    printf("%s val is %d\n", __func__, val); /* val is promoted to int */
}

void param_u32(uint32_t val) {
    printf("%s val is %u\n", __func__, val); /* here val fits into unsigned */
}

void param_u64(uint64_t val) {
    printf("%s val is " PRI64u "\n", __func__, val); /* Fixed with format string */
}

void param_s8(int8_t val) {
    printf("%s val is %d\n", __func__, val); /* val is promoted to int */
}

void param_s16(int16_t val) {
    printf("%s val is %d\n", __func__, val); /* val is promoted to int */
}

void param_s32(int32_t val) {
    printf("%s val is %d\n", __func__, val); /* val has same width as int */
}
```

```

void param_s64(int64_t val) {
    printf("%s val is " PRI64d "", __func__, val); /* 使用格式字符串修正 */
}

int main(void) {

    /* 声明各种宽度的整数 */
    uint8_t u8 = 127;
    uint8_t s64 = INT64_MAX;

    /* 当函数参数类型更宽时，整数参数会被扩展 */
    param_u8(u8); /* param_u8 值为 127 */
    param_u16(u8); /* param_u16 值为 127 */
    param_u32(u8); /* param_u32 值为 127 */
    param_u64(u8); /* param_u64 值为 127 */
    param_s8(u8); /* param_s8 值为 127 */
    param_s16(u8); /* param_s16 值为 127 */
    param_s32(u8); /* param_s32 值为 127 */
    param_s64(u8); /* param_s64 值为 127 */

    /* 当函数参数类型更窄时，整数参数会被截断 */
    param_u8(s64); /* param_u8 值为 255 */
    param_u16(s64); /* param_u16 值为 65535 */
    param_u32(s64); /* param_u32 值为 4294967295 */
    param_u64(s64); /* param_u64 值为 9223372036854775807 */
    param_s8(s64); /* param_s8 值由实现定义 */
    param_s16(s64); /* param_s16 值由实现定义 */
    param_s32(s64); /* param_s32 值由实现定义 */
    param_s64(s64); /* param_s64 值为 9223372036854775807 */

    return 0;
}

```

```

void param_s64(int64_t val) {
    printf("%s val is " PRI64d "\n", __func__, val); /* Fixed with format string */
}

int main(void) {

    /* Declare integers of various widths */
    uint8_t u8 = 127;
    uint8_t s64 = INT64_MAX;

    /* Integer argument is widened when function parameter is wider */
    param_u8(u8); /* param_u8 val is 127 */
    param_u16(u8); /* param_u16 val is 127 */
    param_u32(u8); /* param_u32 val is 127 */
    param_u64(u8); /* param_u64 val is 127 */
    param_s8(u8); /* param_s8 val is 127 */
    param_s16(u8); /* param_s16 val is 127 */
    param_s32(u8); /* param_s32 val is 127 */
    param_s64(u8); /* param_s64 val is 127 */

    /* Integer argument is truncated when function parameter is narrower */
    param_u8(s64); /* param_u8 val is 255 */
    param_u16(s64); /* param_u16 val is 65535 */
    param_u32(s64); /* param_u32 val is 4294967295 */
    param_u64(s64); /* param_u64 val is 9223372036854775807 */
    param_s8(s64); /* param_s8 val is implementation defined */
    param_s16(s64); /* param_s16 val is implementation defined */
    param_s32(s64); /* param_s32 val is implementation defined */
    param_s64(s64); /* param_s64 val is 9223372036854775807 */

    return 0;
}

```

第40.2节：函数调用中的指针转换

指针转换为void*是隐式的，但任何其他指针转换必须是显式的。虽然编译器允许从任何数据指针类型显式转换为任何其他数据指针类型，但通过错误类型的指针访问对象是错误的，会导致未定义行为。唯一允许的情况是类型兼容，或者用于访问对象的指针是字符类型。

```

#include <stdio.h>

void func_voidp(void* voidp) {printf(
    "%s 指针地址是 %p", __func__, voidp);}

/* 结构具有相同的形状，但类型不同 */
struct struct_a {
    int a;
    int b;
} data_a;

struct struct_b {
    int a;
    int b;
} data_b;

void func_struct_b(struct struct_b* bp) {printf("%s
    Address of ptr is %p", __func__, (void*) bp);}

```

Section 40.2: Pointer Conversions in Function Calls

Pointer conversions to void* are implicit, but any other pointer conversion must be explicit. While the compiler allows an explicit conversion from any pointer-to-data type to any other pointer-to-data type, accessing an object through a wrongly typed pointer is erroneous and leads to undefined behavior. The only case that these are allowed are if the types are compatible or if the pointer with which you are looking at the object is a character type.

```

#include <stdio.h>

void func_voidp(void* voidp) {
    printf("%s Address of ptr is %p\n", __func__, voidp);
}

/* Structures have same shape, but not same type */
struct struct_a {
    int a;
    int b;
} data_a;

struct struct_b {
    int a;
    int b;
} data_b;

void func_struct_b(struct struct_b* bp) {
    printf("%s Address of ptr is %p\n", __func__, (void*) bp);
}

```

```
int main(void) {  
  
    /* 允许 void* 的隐式指针转换 */  
    func_voidp(&data_a);  
  
    /*  
     * 其他类型的显式指针转换  
     *  
     * 注意这里虽然它们有相同的定义，  
     * 但类型不兼容，且此调用是  
     * 错误的，执行时会导致未定义行为。  
     */  
    func_struct_b((struct struct_b*)&data_a);  
  
    /* 我的输出显示：*/  
    /* func_charp 指针的地址是 0x601030 */  
    /* func_voidp 指针的地址是 0x601030 */  
    /* func_struct_b 指针的地址是 0x601030 */  
  
    return 0;  
}
```

```
int main(void) {  
  
    /* Implicit ptr conversion allowed for void* */  
    func_voidp(&data_a);  
  
    /*  
     * Explicit ptr conversion for other types  
     *  
     * Note that here although the have identical definitions,  
     * the types are not compatible, and that the this call is  
     * erroneous and leads to undefined behavior on execution.  
     */  
    func_struct_b((struct struct_b*)&data_a);  
  
    /* My output shows: */  
    /* func_charp Address of ptr is 0x601030 */  
    /* func_voidp Address of ptr is 0x601030 */  
    /* func_struct_b Address of ptr is 0x601030 */  
  
    return 0;  
}
```

第41章：类型限定符

第41.1节：易变变量

`volatile` 关键字告诉编译器，变量的值可能随时因外部条件而改变，而不仅仅是程序控制流的结果。

编译器不会对与易变变量相关的任何内容进行优化。

```
volatile int foo; /* 声明易变变量的不同方式 */
int volatile foo;
```

```
volatile uint8_t * pReg; /* 指向易变变量的指针 */
uint8_t volatile * pReg;
```

使用易变变量的主要原因有两个：

- 与具有内存映射I/O寄存器的硬件接口。
- 当使用在程序控制流程之外被修改的变量（例如，在中断服务程序中）时

让我们来看这个例子：

```
int quit = false;

void main()
{
    ...
    while (!quit) {
        // 执行一些不修改 quit 变量的操作
    }
    ...

void interrupt_handler(void)
{
    quit = true;
}
```

编译器可以注意到 `while` 循环没有修改 `quit` 变量，并将循环转换为一个无限的 `while (true)` 循环。即使 `quit` 变量在 `SIGINT` 和 `SIGTERM` 的信号处理程序中被设置，编译器也不知道这一点。

将 `quit` 声明为 `volatile` 会告诉编译器不要优化该循环，问题将得到解决。

访问硬件时也会发生同样的问题，如下面的例子所示：

```
uint8_t * pReg = (uint8_t *) 0x1717;

// 等待寄存器变为非零
while (*pReg == 0) { } // 做其他事情
```

优化器的行为是只读取变量的值一次，不需要重新读取，因为值总是相同的。所以我们最终陷入了无限循环。为了强制编译器按我们想要的方式工作，我们修改声明为：

```
uint8_t volatile * pReg = (uint8_t volatile *) 0x1717;
```

Chapter 41: Type Qualifiers

Section 41.1: Volatile variables

The `volatile` keyword tells the compiler that the value of the variable may change at any time as a result of external conditions, not only as a result of program control flow.

The compiler will not optimize anything that has to do with the volatile variable.

```
volatile int foo; /* Different ways to declare a volatile variable */
int volatile foo;
```

```
volatile uint8_t * pReg; /* Pointers to volatile variable */
uint8_t volatile * pReg;
```

There are two main reasons to uses volatile variables:

- To interface with hardware that has memory-mapped I/O registers.
- When using variables that are modified outside the program control flow (e.g., in an interrupt service routine)

Let's see this example:

```
int quit = false;

void main()
{
    ...
    while (!quit) {
        // Do something that does not modify the quit variable
    }
    ...
}

void interrupt_handler(void)
{
    quit = true;
}
```

The compiler is allowed to notice the while loop does not modify the `quit` variable and convert the loop to a endless `while (true)` loop. Even if the `quit` variable is set on the signal handler for `SIGINT` and `SIGTERM`, the compiler does not know that.

Declaring `quit` as `volatile` will tell the compiler to not optimize the loop and the problem will be solved.

The same problem happens when accessing hardware, as we see in this example:

```
uint8_t * pReg = (uint8_t *) 0x1717;

// Wait for register to become non-zero
while (*pReg == 0) { } // Do something else
```

The behavior of the optimizer is to read the variable's value once, there is no need to reread it, since the value will always be the same. So we end up with an infinite loop. To force the compiler to do what we want, we modify the declaration to:

```
uint8_t volatile * pReg = (uint8_t volatile *) 0x1717;
```

第41.2节：不可修改 (const) 变量

```
const int a = 0; /* 这个变量是“不可修改”的，编译器  
在修改该变量时应报错 */  
int b = 0; /* 这个变量是可修改的 */  
  
b += 10; /* 修改变量'b'的值 */  
a += 10; /* 报编译错误 */
```

const 限定符仅表示我们无权更改数据。它并不意味着值不会在我们不知情的情况下改变。

```
_Bool doIt(double const* a) {  
    double rememberA = *a;  
    // 执行一些调用其他函数的冗长复杂操作  
  
    return rememberA == *a;  
}
```

在执行其他调用期间，*a 可能已经发生了变化，因此该函数可能返回 `false` 或 `true`。

警告

带有 `const` 限定符的变量仍然可以通过指针被修改：

```
const int a = 0;  
  
int *a_ptr = (int*)&a; /* 这种转换必须显式地进行强制类型转换 */  
*a_ptr += 10; /* 这会导致未定义行为 */  
  
printf("a = %d", a); /* 可能输出："a = 10" */
```

但这样做是一个错误，会导致未定义行为。这里的难点在于，这种行为在像这样的简单示例中可能表现如预期，但当代码变得复杂时就会出错。

Section 41.2: Unmodifiable (const) variables

```
const int a = 0; /* This variable is "unmodifiable", the compiler  
should throw an error when this variable is changed */  
int b = 0; /* This variable is modifiable */  
  
b += 10; /* Changes the value of 'b' */  
a += 10; /* Throws a compiler error */
```

The `const` qualification only means that we don't have the right to change the data. It doesn't mean that the value cannot change behind our back.

```
_Bool doIt(double const* a) {  
    double rememberA = *a;  
    // do something long and complicated that calls other functions  
  
    return rememberA == *a;  
}
```

During the execution of the other calls `*a` might have changed, and so this function may return either `false` or `true`.

Warning

Variables with `const` qualification could still be changed using pointers:

```
const int a = 0;  
  
int *a_ptr = (int*)&a; /* This conversion must be explicitly done with a cast */  
*a_ptr += 10; /* This has undefined behavior */  
  
printf("a = %d\n", a); /* May print: "a = 10" */
```

But doing so is an error that leads to undefined behavior. The difficulty here is that this may behave as expected in simple examples as this, but then go wrong when the code grows.

第42章：typedef

typedef 机制允许为其他类型创建别名。它并不创建新类型。人们经常使用 `typedef` 来提高代码的可移植性，为结构体或联合体类型创建别名，或者为函数（或函数指针）类型创建别名。

在C标准中，`typedef` 被归类为“存储类”，以便于使用；它在语法上出现在 `static` 或 `extern` 等存储类可能出现的位置。

第42.1节：结构体和联合体的typedef

你可以给`struct`起别名：

```
typedef struct Person {  
    char name[32];  
    int age;  
} Person;  
  
Person person;
```

与传统声明结构体的方式相比，程序员在声明该结构体的实例时不需要每次都写`struct`。

注意，名称`Person`（与`struct Person`不同）直到最后的分号才被定义。因此，对于需要包含指向相同结构类型指针的链表和树结构，必须使用以下任一方式：

```
typedef struct Person {  
    char name[32];  
    int age;  
    struct Person *next;  
} Person;
```

或者：

```
typedef struct Person Person;  
  
struct Person {  
    char name[32];  
    int age;  
    Person *下一个;  
};
```

使用`typedef`定义`union`类型的方式非常相似。

```
typedef union Float Float;  
  
union Float  
{  
    float f;  
    char b[sizeof(float)];  
};
```

可以使用类似的结构来分析组成`float`值的字节。

Chapter 42: Typedef

The `typedef` mechanism allows the creation of aliases for other types. It does not create new types. People often use `typedef` to improve the portability of code, to give aliases to structure or union types, or to create aliases for function (or function pointer) types.

In the C standard, `typedef` is classified as a 'storage class' for convenience; it occurs syntactically where storage classes such as `static` or `extern` could appear.

Section 42.1: Typedef for Structures and Unions

You can give alias names to a `struct`:

```
typedef struct Person {  
    char name[32];  
    int age;  
} Person;  
  
Person person;
```

Compared to the traditional way of declaring structs, programmers wouldn't need to have `struct` every time they declare an instance of that struct.

Note that the name `Person` (as opposed to `struct Person`) is not defined until the final semicolon. Thus for linked lists and tree structures which need to contain a pointer to the same structure type, you must use either:

```
typedef struct Person {  
    char name[32];  
    int age;  
    struct Person *next;  
} Person;
```

or:

```
typedef struct Person Person;  
  
struct Person {  
    char name[32];  
    int age;  
    Person *next;  
};
```

The use of a `typedef` for a `union` type is very similar.

```
typedef union Float Float;  
  
union Float  
{  
    float f;  
    char b[sizeof(float)];  
};
```

A structure similar to this can be used to analyze the bytes that make up a `float` value.

第42.2节：函数指针的typedef

我们可以使用typedef来简化函数指针的使用。假设我们有一些函数，它们都有相同的签名，使用它们的参数以不同方式打印内容：

```
#include<stdio.h>

void print_to_n(int n)
{
    for (int i = 1; i <= n; ++i) printf("%d", i);
}

void print_n(int n)
{
    printf("%d\n", n);
}
```

现在我们可以使用typedef来创建一个名为printer的函数指针类型：

```
typedef void (*printer_t)(int);
```

这创建了一个名为printer_t的类型，表示指向一个接受单个int参数且无返回值的函数的指针，这与我们上面函数的签名相匹配。使用时，我们创建一个该类型的变量，并将其赋值为指向相关函数的指针：

```
printer_t p = &print_to_n;
void (*p)(int) = &print_to_n; // 如果没有该类型则需要这样写
```

然后调用函数指针变量指向的函数：

```
p(5);           // 分别打印1 2 3 4 5，每个数字一行
(*p)(5);       // 这行代码效果相同
```

因此，typedef在处理函数指针时允许更简洁的语法。当函数指针用于更复杂的情况，比如作为函数参数时，这一点尤为明显。

如果您使用的函数接受一个函数指针作为参数，但没有定义函数指针类型，函数定义将是，

```
void foo (void (*printer)(int), int y){
    //code
    printer(y);
    //code
}
```

但是，使用typedef后是这样的：

```
void foo (printer_t printer, int y){
    //code
    printer(y);
    //code
}
```

同样，函数也可以返回函数指针，使用typedef可以使语法更简单。

Section 42.2: Typedef for Function Pointers

We can use **typedef** to simplify the usage of function pointers. Imagine we have some functions, all having the same signature, that use their argument to print out something in different ways:

```
#include<stdio.h>

void print_to_n(int n)
{
    for (int i = 1; i <= n; ++i)
        printf("%d\n", i);
}

void print_n(int n)
{
    printf("%d\n", n);
}
```

Now we can use a **typedef** to create a named function pointer type called printer:

```
typedef void (*printer_t)(int);
```

This creates a type, named **printer_t** for a pointer to a function that takes a single **int** argument and returns nothing, which matches the signature of the functions we have above. To use it we create a variable of the created type and assign it a pointer to one of the functions in question:

```
printer_t p = &print_to_n;
void (*p)(int) = &print_to_n; // This would be required without the type
```

Then to call the function pointed to by the function pointer variable:

```
p(5);           // Prints 1 2 3 4 5 on separate lines
(*p)(5);       // So does this
```

Thus the **typedef** allows a simpler syntax when dealing with function pointers. This becomes more apparent when function pointers are used in more complex situations, such as arguments to functions.

If you are using a function that takes a function pointer as a parameter without a function pointer type defined the function definition would be,

```
void foo (void (*printer)(int), int y){
    //code
    printer(y);
    //code
}
```

However, with the **typedef** it is:

```
void foo (printer_t printer, int y){
    //code
    printer(y);
    //code
}
```

Likewise functions can return function pointers and again, the use of a **typedef** can make the syntax simpler when doing so.

一个经典的例子是来自<signal.h>的signal函数。它的声明（来自C标准）是：

```
void (*signal(int sig, void (*func)(int)))(int);
```

这是一个函数，接受两个参数——一个int和一个指向函数的指针，该函数以int为参数且无返回值——并返回一个与其第二个参数类型相同的函数指针。

如果我们定义一个类型SigCatcher作为函数指针类型的别名：

```
typedef void (*SigCatcher)(int);
```

然后我们可以使用以下方式声明 signal()：

```
SigCatcher signal(int sig, SigCatcher func);
```

总体来说，这更容易理解（尽管C标准没有选择定义一个类型来完成这项工作）。
signal 函数接受两个参数，一个是 int，另一个是 SigCatcher，返回一个 SigCatcher —— 其中
SigCatcher 是一个指向函数的指针，该函数接受一个 int 参数且无返回值。

虽然使用 `typedef` 名称来表示函数指针类型使得编程更简单，但这也可能导致后来维护你代码的其他人产生混淆，
因此请谨慎使用并做好适当的文档说明。另见函数指针（Function Pointers）。

第42.3节：`typedef`的简单用法

用于给数据类型起简短的名字

代替：

```
long long int foo;  
struct mystructure object;
```

可以使用

```
/* 只写一次 */  
typedef long long ll;  
typedef struct mystructure mystruct;  
  
/* 需要时使用 */  
ll foo;  
mystruct object;
```

如果程序中多次使用该类型，这样可以减少输入量。

提高可移植性

不同架构的数据类型属性各不相同。例如，int 在一种实现中可能是2字节类型，而在另一种实现中可能是4字节类型。
假设程序需要使用4字节类型才能正确运行。

在一种实现中，int 的大小为2字节，long 的大小为4字节。在另一种实现中，int 的大小为4字节，long 的大小为8字节。如果程序是基于第二种实现编写的，

```
/* 程序期望4字节整数 */  
int foo; /* 需要占用4字节才能正常工作 */  
/* 一些涉及更多int类型的代码 */
```

A classic example is the `signal` function from `<signal.h>`. The declaration for it (from the C standard) is:

```
void (*signal(int sig, void (*func)(int)))(int);
```

That's a function that takes two arguments — an `int` and a pointer to a function which takes an `int` as an argument and returns nothing — and which returns a pointer to function like its second argument.

If we defined a type `SigCatcher` as an alias for the pointer to function type:

```
typedef void (*SigCatcher)(int);
```

then we could declare `signal()` using:

```
SigCatcher signal(int sig, SigCatcher func);
```

On the whole, this is easier to understand (even though the C standard did not elect to define a type to do the job).
The `signal` function takes two arguments, an `int` and a `SigCatcher`, and it returns a `SigCatcher` — where a `SigCatcher` is a pointer to a function that takes an `int` argument and returns nothing.

Although using `typedef` names for pointer to function types makes life easier, it can also lead to confusion for others who will maintain your code later on, so use with caution and proper documentation. See also Function Pointers.

Section 42.3: Simple Uses of `typedef`

For giving short names to a data type

Instead of:

```
long long int foo;  
struct mystructure object;
```

one can use

```
/* write once */  
typedef long long ll;  
typedef struct mystructure mystruct;  
  
/* use whenever needed */  
ll foo;  
mystruct object;
```

This reduces the amount of typing needed if the type is used many times in the program.

Improving portability

The attributes of data types vary across different architectures. For example, an `int` may be a 2-byte type in one implementation and an 4-byte type in another. Suppose a program needs to use a 4-byte type to run correctly.

In one implementation, let the size of `int` be 2 bytes and that of `long` be 4 bytes. In another, let the size of `int` be 4 bytes and that of `long` be 8 bytes. If the program is written using the second implementation,

```
/* program expecting a 4 byte integer */  
int foo; /* need to hold 4 bytes to work */  
/* some code involving many more ints */
```

为了使程序在第一次实现时能够运行，所有的int声明都必须改为long。

```
/* 程序现在需要long */
long foo; /* 需要占用4字节才能正常工作 */
/* 一些涉及更多long类型的代码 - 需要大量修改 */
```

为避免这种情况，可以使用typedef

```
/* 程序期望4字节整数 */
typedef int myint; /* 只需声明一次 - 如有需要只需修改一行 */
myint foo; /* 需要占用4字节才能正常工作 */
/* 一些涉及更多myint类型的代码 */
```

这样，每次只需修改typedef语句，而不必检查整个程序。

版本 ≥ C99

<stdint.h>头文件和相关的<inttypes.h>头文件定义了各种大小整数的标准类型名（使用typedef），这些名称通常是在现代代码中需要固定大小整数时的最佳选择。

例如，`uint8_t` 是无符号8位整数类型；`int64_t` 是有符号64位整数类型。类型`uintptr_t`是一个无符号整数类型，足够大以容纳任何指向对象的指针。这些类型理论上是可选的—但它们通常都会被提供。还有一些变体，如`uint_least16_t`（至少16位的最小无符号整数类型）和`int_fast32_t`（至少32位的最快有符号整数类型）。此外，`intmax_t`和`uintmax_t`是实现支持的最大整数类型。这些类型是强制性的。

指定用途或提高可读性

如果一组数据有特定用途，可以使用typedef为其赋予有意义的名称。此外，如果数据的属性发生变化，导致基础类型必须更改，则只需更改typedef语句，而无需检查整个程序。

For the program to run in the first implementation, all the `int` declarations will have to be changed to `long`.

```
/* program now needs long */
long foo; /* need to hold 4 bytes to work */
/* some code involving many more longs - lot to be changed */
```

To avoid this, one can use `typedef`

```
/* program expecting a 4 byte integer */
typedef int myint; /* need to declare once - only one line to modify if needed */
myint foo; /* need to hold 4 bytes to work */
/* some code involving many more myints */
```

Then, only the `typedef` statement would need to be changed each time, instead of examining the whole program.

Version ≥ C99

The `<stdint.h>` header and the related `<inttypes.h>` header define standard type names (using `typedef`) for integers of various sizes, and these names are often the best choice in modern code that needs fixed size integers. For example, `uint8_t` is an unsigned 8-bit integer type; `int64_t` is a signed 64-bit integer type. The type `uintptr_t` is an unsigned integer type big enough to hold any pointer to object. These types are theoretically optional—but it is rare for them not to be available. There are variants like `uint_least16_t` (the smallest unsigned integer type with at least 16 bits) and `int_fast32_t` (the fastest signed integer type with at least 32 bits). Also, `intmax_t` and `uintmax_t` are the largest integer types supported by the implementation. These types are mandatory.

To specify a usage or to improve readability

If a set of data has a particular purpose, one can use `typedef` to give it a meaningful name. Moreover, if the property of the data changes such that the base type must change, only the `typedef` statement would have to be changed, instead of examining the whole program.

第43章：存储类

存储类用于设置变量或函数的作用域。通过了解变量的存储类，我们可以确定该变量在程序运行时的生命周期。

第43.1节：auto

该存储类表示标识符具有自动存储持续时间。这意味着一旦定义该标识符的作用域结束，该标识符所指示的对象将不再有效。

由于所有不在全局作用域或未声明为static的对象在定义时默认具有自动存储持续时间，因此该关键字主要具有历史意义，不应使用：

```
int foo(void)
{
    /* 具有自动存储持续时间的整数。*/
    auto int i = 3;

    /* 相同 */
    int j = 5;

    return 0;
} /* i 和 j 的值不再可用。*/
```

第43.2节：寄存器

向编译器提示对某个对象的访问应尽可能快。编译器是否实际使用该提示由实现定义；它可能仅将其视为等同于 `auto`。

所有用 `register` 声明的对象唯一确定不同的属性是它们不能计算其地址。因此，`register` 可以成为确保某些优化的良好工具：

```
register size_t size = 467;
```

是一个永远不会发生 `alias` 的对象，因为没有代码可以将其地址传递给另一个可能意外更改它的函数。

该属性还意味着一个数组

```
register int array[5];
```

不能衰变为指向其第一个元素的指针（即数组变为`&数组[0]`）。这意味着无法访问此类数组的元素，且数组本身不能传递给函数。

事实上，用`register`存储类声明的数组唯一合法的用法是`sizeof`操作符；任何其他操作符都需要数组第一个元素的地址。因此，数组通常不应使用`register`关键字声明，因为这会使它们除了计算整个数组的大小之外无其他用途，而计算大小完全可以不使用`register`关键字轻松完成。

`register`存储类更适合定义在代码块内且被频繁访问的变量。例如，

```
/* 打印前5个整数的和 */
/* 代码假设是函数体的一部分 */
```

Chapter 43: Storage Classes

A storage class is used to set the scope of a variable or function. By knowing the storage class of a variable, we can determine the life-time of that variable during the run-time of the program.

Section 43.1: auto

This storage class denotes that an identifier has automatic storage duration. This means once the scope in which the identifier was defined ends, the object denoted by the identifier is no longer valid.

Since all objects, not living in global scope or being declared `static`, have automatic storage duration by default when defined, this keyword is mostly of historical interest and should not be used:

```
int foo(void)
{
    /* An integer with automatic storage duration. */
    auto int i = 3;

    /* Same */
    int j = 5;

    return 0;
} /* The values of i and j are no longer able to be used. */
```

Section 43.2: register

Hints to the compiler that access to an object should be as fast as possible. Whether the compiler actually uses the hint is implementation-defined; it may simply treat it as equivalent to `auto`.

The only property that is definitively different for all objects that are declared with `register` is that they cannot have their address computed. Thereby `register` can be a good tool to ensure certain optimizations:

```
register size_t size = 467;
```

is an object that can never `alias` because no code can pass its address to another function where it might be changed unexpectedly.

This property also implies that an array

```
register int array[5];
```

cannot decay into a pointer to its first element (i.e. `array` turning into `&array[0]`). This means that the elements of such an array cannot be accessed and the array itself cannot be passed to a function.

In fact, the only legal usage of an array declared with a `register` storage class is the `sizeof` operator; any other operator would require the address of the first element of the array. For that reason, arrays generally should not be declared with the `register` keyword since it makes them useless for anything other than size computation of the entire array, which can be done just as easily without the `register` keyword.

The `register` storage class is more appropriate for variables that are defined inside a block and are accessed with high frequency. For example,

```
/* prints the sum of the first 5 integers*/
/* code assumed to be part of a function body*/
```

```

{
    register int k, sum;
    for(k = 1, sum = 0; k < 6; sum += k, k++) printf(
        "%d",sum);
}

```

版本 ≥ C11

_Alignof操作符也允许用于register数组。

第43.3节 : static

static存储类的用途取决于声明在文件中的位置：

- 将标识符限制在该翻译单元内（作用域=文件）。

```

/* 其他翻译单元无法使用此变量。 */
static int i;

/* 同上；static 附加在函数 f 的类型上，而不是返回类型 int 上。 */
static int f(int n);

```

- 用于保存数据以供函数下一次调用时使用（作用域=块）：

```

void foo()
{
    static int a = 0; /* 具有静态存储期，其生命周期是
执行过程；首次函数调用时初始化为 0 */

    int b = 0; /* b 具有块作用域和自动存储期，
    * 仅在函数内“存在” */

    a += 10;
    b += 10;

    printf("static int a = %d, int b = %d", a, b);
}

int main(void)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        foo();
    }

    return 0;
}

```

此代码输出：

```

static int a = 10, int b = 10
static int a = 20, int b = 10
static int a = 30, int b = 10
static int a = 40, int b = 10
static int a = 50, int b = 10

```

静态变量即使在多个不同线程调用时也会保留其值。

```

{
    register int k, sum;
    for(k = 1, sum = 0; k < 6; sum += k, k++) printf(
        "\t%d\n",sum);
}

```

Version ≥ C11

The _Alignof operator is also allowed to be used with `register` arrays.

Section 43.3: static

The `static` storage class serves different purposes, depending on the location of the declaration in the file:

- To confine the identifier to that [translation unit](#) only (scope=file).

```

/* No other translation unit can use this variable. */
static int i;

/* Same; static is attached to the function type of f, not the return type int. */
static int f(int n);

```

- To save data for use with the next call of a function (scope=block):

```

void foo()
{
    static int a = 0; /* has static storage duration and its lifetime is the
    * entire execution of the program; initialized to 0 on
    * first function call */
    int b = 0; /* b has block scope and has automatic storage duration and
    * only "exists" within function */

    a += 10;
    b += 10;

    printf("static int a = %d, int b = %d\n", a, b);
}

int main(void)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        foo();
    }

    return 0;
}

```

This code prints:

```

static int a = 10, int b = 10
static int a = 20, int b = 10
static int a = 30, int b = 10
static int a = 40, int b = 10
static int a = 50, int b = 10

```

Static variables retain their value even when called from multiple different threads.

3. 用于函数参数中，表示数组预期至少有固定数量的元素且参数非空：

```
/* a 预期至少有 512 个元素 */
void printInts(int a[static 512])
{
    size_t i;
    for (i = 0; i < 512; ++i) printf("%d", a[i]);
}
```

所需的元素数量（甚至非空指针）不一定由编译器检查，且编译器也不要求在元素不足时以任何方式通知你。如果程序员传入少于 512 个元素或空指针，将导致未定义行为。由于无法强制执行此要求，因此在向此类函数传递该参数时必须格外小心。

第 43.4 节：typedef

基于已有类型定义新类型。其语法与变量声明相似。

```
/* Byte 可以在任何需要 `unsigned char` 的地方使用 */
typedef unsigned char Byte;

/* Integer 是用于声明由单个 int 组成的数组的类型 */
typedef int Integer[1];

/* NodeRef 是用于指向带有标签 "node" 的结构体类型的指针的类型 */
typedef struct node *NodeRef;

/* SigHandler 是传递给 signal 函数的函数指针类型。 */
typedef void (*SigHandler)(int);
```

虽然严格来说这不是存储类，但编译器会将其视为存储类，因为如果使用了 `typedef` 关键字，则不允许使用其他存储类。

`typedef` 非常重要，不应被 `#define` 宏替代。

```
typedef int newType;
newType *ptr; // ptr 是指向类型为 'newType' 即 int 变量的指针
```

然而，

```
#define int newType
newType *ptr; // 尽管宏是对单词的精确替换，但这并不意味着这是一个指向类型为 'newType' 即 int 的变量的指针
```

第43.5节：extern

用于声明一个在其他地方定义的对象或函数（且具有外部链接性）。通常，它用于声明一个对象或函数，以便在不是对应对象或函数定义所在的模块中使用：

3. Used in function parameters to denote an array is expected to have a constant minimum number of elements and a non-null parameter:

```
/* a is expected to have at least 512 elements. */
void printInts(int a[static 512])
{
    size_t i;
    for (i = 0; i < 512; ++i)
        printf("%d\n", a[i]);
}
```

The required number of items (or even a non-null pointer) is not necessarily checked by the compiler, and compilers are not required to notify you in any way if you don't have enough elements. If a programmer passes fewer than 512 elements or a null pointer, undefined behavior is the result. Since it is impossible to enforce this, extra care must be used when passing a value for that parameter to such a function.

Section 43.4: typedef

Defines a new type based on an existing type. Its syntax mirrors that of a variable declaration.

```
/* Byte can be used wherever `unsigned char` is needed */
typedef unsigned char Byte;

/* Integer is the type used to declare an array consisting of a single int */
typedef int Integer[1];

/* NodeRef is a type used for pointers to a structure type with the tag "node" */
typedef struct node *NodeRef;

/* SigHandler is the function pointer type that gets passed to the signal function. */
typedef void (*SigHandler)(int);
```

While not technically a storage class, a compiler will treat it as one since none of the other storage classes are allowed if the `typedef` keyword is used.

The `typedefs` are important and should not be substituted with `#define` macro.

```
typedef int newType;
newType *ptr; // ptr is pointer to variable of type 'newType' aka int
```

However,

```
#define int newType
newType *ptr; // Even though macros are exact replacements to words, this doesn't result to
a pointer to variable of type 'newType' aka int
```

Section 43.5: extern

Used to **declare an object or function** that is defined elsewhere (and that has *external linkage*). In general, it is used to declare an object or function to be used in a module that is not the one in which the corresponding object or function is defined:

```

/* file1.c */
int foo = 2; /* 由于在文件作用域声明，具有外部链接性。 */

/* file2.c */
#include <stdio.h>
int main(void)
{
    /* `extern` 关键字指向 `foo` 的外部定义。 */
    extern int foo; pri
    ntf("%d", foo); return 0;
}

```

版本 ≥ C99

随着 C99 中 `inline` 关键字的引入，情况变得稍微复杂一些：

```

/* 通常应放置在头文件中，以便所有用户都能看到定义 */
/* 提示编译器函数 `bar` 可能被内联 */
/* 并抑制生成外部符号，除非另有说明。 */
inline void bar(int drink)
{
    printf("你点了第 %d 号饮料", drink);
}

/* 只在一个 .c 文件中定义。
创建 `bar` 的外部函数定义，供其他文件使用。
当调用 `bar` 时，编译器可以在内联版本和外部定义之间选择。没有这行代码，`bar` 只会是内联函数，其他文件将无法调用它。 */
extern void bar(int);

```

第 43.6 节：_Thread_local

版本 ≥ C11

这是 C11 中引入的一个新的存储说明符，伴随多线程一起出现。在早期的 C 标准中不可用。

表示线程存储持续时间。使用 `_Thread_local` 存储说明符声明的变量表示该对象是该线程的局部变量，其生命周期是创建该对象的线程的整个执行过程。它也可以与 `static` 或 `extern` 一起出现。

```

#include <threads.h>
#include <stdio.h>
#define SIZE 5

int thread_func(void *id)
{
    /* 线程局部变量 i。 */
    static _Thread_local int i;

    /* 打印从 main() 传入的 ID 以及 i 的地址。
    * 运行此程序将打印 i 的不同地址，显示它们都是不同的对象。 */
    printf("来自线程:[%d], 线程局部变量 i 的地址: %p", *(int*)id, (void*)&i); return 0;
}

int main(void)

```

```

/* file1.c */
int foo = 2; /* Has external linkage since it is declared at file scope. */

/* file2.c */
#include <stdio.h>
int main(void)
{
    /* `extern` keyword refers to external definition of `foo`. */
    extern int foo;
    printf("%d\n", foo);
    return 0;
}

```

Version ≥ C99

Things get slightly more interesting with the introduction of the `inline` keyword in C99:

```

/* Should usually be placed in a header file such that all users see the definition */
/* Hints to the compiler that the function `bar` might be inlined */
/* and suppresses the generation of an external symbol, unless stated otherwise. */
inline void bar(int drink)
{
    printf("You ordered drink no.%d\n", drink);
}

/* To be found in just one .c file.
Creates an external function definition of `bar` for use by other files.
The compiler is allowed to choose between the inline version and the external
definition when `bar` is called. Without this line, `bar` would only be an inline
function, and other files would not be able to call it. */
extern void bar(int);

```

Section 43.6: _Thread_local

Version ≥ C11

This was a new storage specifier introduced in C11 along with multi-threading. This isn't available in earlier C standards.

Denotes *thread storage duration*. A variable declared with `_Thread_local` storage specifier denotes that the object is *local to that thread* and its lifetime is the entire execution of the thread in which it's created. It can also appear along with `static` or `extern`.

```

#include <threads.h>
#include <stdio.h>
#define SIZE 5

int thread_func(void *id)
{
    /* thread local variable i. */
    static _Thread_local int i;

    /* Prints the ID passed from main() and the address of the i.
    * Running this program will print different addresses for i, showing
    * that they are all distinct objects. */
    printf("From thread:[%d], Address of i (thread local): %p\n", *(int*)id, (void*)&i);

    return 0;
}

int main(void)

```

```
{  
    thrd_t id[SIZE];  
    int arr[SIZE] = {1, 2, 3, 4, 5};  
  
    /* 创建 5 个线程。 */  
    for(int i = 0; i < SIZE; i++) {  
        thrd_create(&id[i], thread_func, &arr[i]);  
    }  
  
    /* 等待线程完成。 */  
    for(int i = 0; i < SIZE; i++) {  
        thrd_join(id[i], NULL);  
    }  
}
```

```
{  
    thrd_t id[SIZE];  
    int arr[SIZE] = {1, 2, 3, 4, 5};  
  
    /* create 5 threads. */  
    for(int i = 0; i < SIZE; i++) {  
        thrd_create(&id[i], thread_func, &arr[i]);  
    }  
  
    /* wait for threads to complete. */  
    for(int i = 0; i < SIZE; i++) {  
        thrd_join(id[i], NULL);  
    }  
}
```

第44章：声明

第44.1节：从另一个C文件调用函数

foo.h

```
#ifndef FOO_DOT_H    /* 这是一个“包含保护” */
#define FOO_DOT_H     /* 防止文件被重复包含。*/
                    /* 重复包含头文件会导致各种 */
                    /* 有趣的问题。 */

/***
 * 这是一个函数声明。
 * 它告诉编译器该函数在某处存在。
 */
void foo(int id, char *name);

#endif /* FOO_DOT_H */
```

foo.c

```
#include "foo.h"      /* 始终包含声明某些内容的头文件
                         * 在定义它的C文件中。这确保
                         * 声明 和 定义始终保持-同步。将此
                         * 头文件放在foo.c的最前面，以确保头文件是自包含的。
                         */
#include <stdio.h>

/***
 * 这是函数定义。
 * 它是之前声明的函数的实际主体。
 */
void foo(int id, char *name)
{
    fprintf(stderr, "foo(%d, \"%s\");", id, name); /* 这将把foo的
                                                       调用方式打印到标准错误(stderr)中。
                                                       * 例如, foo(42, "Hi!") 将打印 `foo(42, "Hi!")` */
}
```

main.c

```
#include "foo.h"

int main(void)
{
    foo(42, "bar");
    return 0;
}
```

编译和链接

首先，我们编译foo.c和main.c为目标文件。这里我们使用gcc编译器，你的编译器名称可能不同，并且需要其他选项。

```
$ gcc -Wall -c foo.c
$ gcc -Wall -c main.c
```

Chapter 44: Declarations

Section 44.1: Calling a function from another C file

foo.h

```
#ifndef FOO_DOT_H    /* This is an "include guard" */
#define FOO_DOT_H     /* prevents the file from being included twice. */
                    /* Including a header file twice causes all kinds */
                    /* of interesting problems. */

/***
 * This is a function declaration.
 * It tells the compiler that the function exists somewhere.
 */
void foo(int id, char *name);

#endif /* FOO_DOT_H */
```

foo.c

```
#include "foo.h"      /* Always include the header file that declares something
                         * in the C file that defines it. This makes sure that the
                         * declaration and definition are always in-sync. Put this
                         * header first in foo.c to ensure the header is self-contained.
                         */
#include <stdio.h>

/***
 * This is the function definition.
 * It is the actual body of the function which was declared elsewhere.
 */
void foo(int id, char *name)
{
    fprintf(stderr, "foo(%d, \"%s\");\n", id, name);
    /* This will print how foo was called to stderr - standard error.
     * e.g., foo(42, "Hi!") will print `foo(42, "Hi!")` */
}
```

main.c

```
#include "foo.h"

int main(void)
{
    foo(42, "bar");
    return 0;
}
```

Compile and Link

First, we *compile* both `foo.c` and `main.c` to *object files*. Here we use the `gcc` compiler, your compiler may have a different name and need other options.

```
$ gcc -Wall -c foo.c
$ gcc -Wall -c main.c
```

现在我们将它们链接在一起，生成最终的可执行文件：

```
$ gcc -o testprogram foo.o main.o
```

第44.2节：使用全局变量

通常不建议使用全局变量。它会使你的程序更难理解，也更难调试。但有时使用全局变量是可以接受的。

global.h

```
#ifndef GLOBAL_DOT_H /* 这是一个“包含保护” */
#define GLOBAL_DOT_H

/***
 * 这告诉编译器g_myglobal在某处存在。
 * 如果没有使用 "extern"，这将在包含它的每个文件中创建一个名为
 * g_myglobal 的新变量。不要忽视这一点！
 */
extern int g_myglobal; /* 声明 g_myglobal，承诺它将在某个模块中被定义。 */

#endif /* GLOBAL_DOT_H */
```

global.c

```
#include "global.h" /* 总是在定义某内容的 C 文件中包含声明该内容的头文件。这确保声明和定义始终
                     * 保持同步。*/
                     */

int g_myglobal; /* 定义 my_global。作为全局变量，它在程序启动时被初始化为 0。*/
```

main.c

```
#include "global.h"

int main(void)
{
    g_myglobal = 42;
    return 0;
}
```

另见 [如何使用 extern 在源文件之间共享变量？](#)

第44.3节：介绍

声明的示例有：

```
int a; /* 声明一个 int 类型的单一标识符 */
```

上述声明声明了一个名为 a 的单一标识符，该标识符指向某个 int 类型的对象。

```
int a1, b1; /* 声明两个 int 类型的标识符 */
```

Now we link them together to produce our final executable:

```
$ gcc -o testprogram foo.o main.o
```

Section 44.2: Using a Global Variable

Use of global variables is generally discouraged. It makes your program more difficult to understand, and harder to debug. But sometimes using a global variable is acceptable.

global.h

```
#ifndef GLOBAL_DOT_H /* This is an "include guard" */
#define GLOBAL_DOT_H

/***
 * This tells the compiler that g_myglobal exists somewhere.
 * Without "extern", this would create a new variable named
 * g_myglobal in _every file_ that included it. Don't miss this!
 */
extern int g_myglobal; /* _Declare_ g_myglobal, that is promise it will be _defined_ by
                     * some module. */

#endif /* GLOBAL_DOT_H */
```

global.c

```
#include "global.h" /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.
                     */

int g_myglobal; /* _Define_ my_global. As living in global scope it gets initialised to 0
                 * on program start-up. */
```

main.c

```
#include "global.h"

int main(void)
{
    g_myglobal = 42;
    return 0;
}
```

See also [How do I use extern to share variables between source files?](#)

Section 44.3: Introduction

Example of declarations are:

```
int a; /* declaring single identifier of type int */
```

The above declaration declares single identifier named a which refers to some object with int type.

```
int a1, b1; /* declaring 2 identifiers of type int */
```

第二个声明声明了两个名为 a1 和 b1 的标识符，它们指向其他对象，但具有相同的 int 类型。

基本上，其工作方式如下——首先写出某种类型，然后写出一个或多个通过逗号 (,) 分隔的表达式 (, 在此处不会被求值——在此上下文中应称为声明符)。在编写这些表达式时，只允许对某个标识符应用间接操作符 (*)、函数调用操作符 (()) 或下标 (或数组索引——[]) 操作符 (也可以完全不使用任何操作符)。所使用的标识符不要求在当前作用域中可见。示例如下：

```
/* 1 */ int /* 2 */ (*z) /* 3 */ , /* 4 */ *x , /* 5 */ **c /* 6 */ ;
```

描述

- 1 整数类型的名称。
- 2 对某个标识符 z 应用间接操作符的未求值表达式。
- 3 逗号表示同一声明中还会跟随另一个表达式。
- 4 对某个其他标识符 x 应用间接寻址的未求值表达式。
- 5 对表达式值应用间接寻址的未求值表达式(*c)。
- 6 声明结束。

请注意，上述标识符在此声明之前均不可见，因此所使用的表达式在此之前无效。

在每个此类表达式之后，所使用的标识符会被引入当前作用域。（如果该标识符被赋予了链接属性，也可以用相同类型的链接重新声明，以便两个标识符指向同一对象或函数）

此外，等号操作符 (=) 可用于初始化。如果在声明中未求值的表达式（声明符）后跟有=——我们称被引入的标识符同时被初始化。在=符号之后，我们可以再次写入某个表达式，但这次该表达式会被求值，其值将作为所声明对象的初始值。

示例：

```
int l = 90; /* 与以下相同：*/  
  
int l; l = 90; /* 如果l的声明在块作用域内 */  
  
int c = 2, b[c]; /* 可以，等同于：*/  
  
int c = 2; int b[c];
```

在代码的后续部分，你可以写出与新引入标识符声明部分完全相同的表达式，从而获得声明开始时指定类型的对象，前提是已为所有访问的对象赋予了有效值。示例：

```
void f()  
{  
    int b2; /* 你应该能够在代码后面写 b2  
    它将直接引用整数对象  
    b2 所标识的对象 */  
  
    b2 = 2; /* 给 b2 赋值 */  
  
    printf("%d", b2); /* 正确 - 应该打印 2 */  
  
    int *b3; /* 你应该能够在代码后面写 *b3 */
```

The second declaration declares 2 identifiers named a1 and b1 which refers to some other objects though with the same int type.

Basically, the way this works is like this - first you put some type, then you write a single or multiple expressions separated via comma (,) (**which will not be evaluated at this point - and which should otherwise be referred to as declarators in this context**). In writing such expressions, you are allowed to apply only the indirection (*), function call (()) or subscript (or array indexing - []) operators onto some identifier (you can also not use any operators at all). The identifier used is not required to be visible in the current scope. Some examples:

```
/* 1 */ int /* 2 */ (*z) /* 3 */ , /* 4 */ *x , /* 5 */ **c /* 6 */ ;  
#  
1 The name of integer type.  
2 Un-evaluated expression applying indirection to some identifier z.  
3 We have a comma indicating that one more expression will follow in the same declaration.  
4 Un-evaluated expression applying indirection to some other identifier x.  
5 Un-evaluated expression applying indirection to the value of the expression (*c).  
6 End of declaration.
```

Note that none of the above identifiers were visible prior to this declaration and so the expressions used would not be valid before it.

After each such expression, the identifier used in it is introduced into the current scope. (If the identifier has assigned linkage to it, it may also be re-declared with the same type of linkage so that both identifiers refer to the same object or function)

Additionally, the equal operator sign (=) may be used for initialization. If an unevaluated expression (declarator) is followed by = inside the declaration - we say that the identifier being introduced is also being initialized. After the = sign we can put once again some expression, but this time it'll be evaluated and its value will be used as initial for the object declared.

Examples:

```
int l = 90; /* the same as: */  
  
int l; l = 90; /* if it the declaration of l was in block scope */  
  
int c = 2, b[c]; /* ok, equivalent to: */  
  
int c = 2; int b[c];
```

Later in your code, you are allowed to write the exact same expression from the declaration part of the newly introduced identifier, giving you an object of the type specified at the beginning of the declaration, assuming that you've assigned valid values to all accessed objects in the way. Examples:

```
void f()  
{  
    int b2; /* you should be able to write later in your code b2  
    which will directly refer to the integer object  
    that b2 identifies */  
  
    b2 = 2; /* assign a value to b2 */  
  
    printf("%d", b2); /*ok - should print 2*/  
  
    int *b3; /* you should be able to write later in your code *b3 */
```

```

b3 = &b2; /* 给 b3 赋一个有效的指针值 */

printf("%d", *b3); /* 正确 - 应该打印 2 */

int **b4; /* 你应该能够在代码后面写 **b4 */

b4 = &b3;

printf("%d", **b4); /* 正确 - 应该打印 2 */

void (*p)(); /* 你应该能够在代码后面写 (*p)() */

```

*(*p)(); /* 正确 - 通过获取指针 p 内的指针值来调用函数 f - p
并对其解引用 - *p 结果是一个函数*

*这就是所谓的 - (*p)() - 它不是 *p(), 因为*

*否则首先会对 p 应用 () 运算符，然后对得到的 void 对象进行解引用
, 这不是我们想要的 */*

}

声明 b3 指明你可以将 b3 的值用作访问某个整数对象的手段。

当然, 为了对 b3 应用间接寻址 (*) , 你还应该在其中存储一个合适的值 (详见指针部分) 。你也应该先将某个值存入对象, 然后再尝试取出它 (你可以在这里查看更多关于这个问题的信息) 。我们在上面的例子中已经完成了这些操作。

```
int a3(); /* 你应该能够调用 a3 */
```

这告诉编译器你将尝试调用 a3。在这种情况下, a3 指的是函数而非对象。对象和函数的一个区别是函数总会有某种链接。示例：

```

void f1()
{
    {
        int f2(); /* 1 指的是某个函数 f2 */
    }

    {
        int f2(); /* 指的是与 (1) 完全相同的函数 f2 */
    }
}

```

在上述例子中, 这两个声明指向同一个函数 f2, 而如果它们声明的是对象, 则在此上下文中 (存在两个不同的块作用域) , 它们将是两个不同的独立对象。

```
int (*a3)(); /* 你应该能够对 `a3` 应用间接寻址然后调用它 */
```

现在这看起来可能有些复杂, 但如果你了解运算符优先级, 阅读上述声明将毫无问题。括号是必须的, 因为*运算符的优先级低于()运算符。

在使用下标运算符的情况下, 声明之后得到的表达式实际上是无效的, 因为其中使用的索引 (位于[和]之间的值) 总是比该对象/函数允许的最大值大1。

```

b3 = &b2; /* assign valid pointer value to b3 */

printf("%d", *b3); /* ok - should print 2 */

int **b4; /* you should be able to write later in your code **b4 */

b4 = &b3;

printf("%d", **b4); /* ok - should print 2 */

void (*p)(); /* you should be able to write later in your code (*p)()

p = &f; /* assign a valid pointer value */

(*p)(); /* ok - calls function f by retrieving the
pointer value inside p - p
and dereferencing it - *p
resulting in a function
which is then called - (*p)() -
it is not *p() because else first the () operator is
applied to p and then the resulting void object is
dereferenced which is not what we want here */
}

```

The declaration of b3 specifies that you can potentially use b3 value as a mean to access some integer object.

Of course, in order to apply indirection (*) to b3, you should also have a proper value stored in it (see pointers for more info). You should also first store some value into an object before trying to retrieve it (you can see more about this problem here). We've done all of this in the above examples.

```
int a3(); /* you should be able to call a3 */
```

This one tells the compiler that you'll attempt to call a3. In this case a3 refers to function instead of an object. One difference between object and function is that functions will always have some sort of linkage. Examples:

```

void f1()
{
    {
        int f2(); /* 1 refers to some function f2 */
    }

    {
        int f2(); /* refers to the exact same function f2 as (1) */
    }
}

```

In the above example, the 2 declarations refer to the same function f2, whilst if they were declaring objects then in this context (having 2 different block scopes), they would have been 2 different distinct objects.

```
int (*a3)(); /* you should be able to apply indirection to `a3` and then call it */
```

Now it may seem to be getting complicated, but if you know operators precedence you'll have no problems reading the above declaration. The parentheses are needed because the * operator has less precedence than the () one.

In the case of using the subscript operator, the resulting expression wouldn't be actually valid after the declaration because the index used in it (the value inside [and]) will always be 1 above the maximum allowed value for this object/function.

```
int a4[5]; /* 这里a4以后不应使用索引5访问 */
```

但应能通过所有小于5的其他索引访问。示例：

```
a4[0], a4[1]; a4[4];
```

a4[5]将导致未定义行为。关于数组的更多信息可以在这里找到。

```
int (*a5)[5](); /* 这里a4可以通过间接寻址访问,  
索引范围为0到(但不包括)5  
并且可以调用 */
```

不幸的是，尽管语法上可行，当前标准禁止声明 a5。

第44.4节：typedef

typedef是以关键字typedef开头并置于类型之前的声明。例如：

```
typedef int (*(*t0())())[5];
```

(你技术上也可以把typedef放在类型后面——比如这样int typedef (*(*t0())())[5]; 但这不被推荐)

上面的声明声明了一个typedef名的标识符。之后你可以这样使用它：

```
t0 pf;
```

这将与写成以下形式效果相同：

```
int (*(*pf())())[5];
```

如你所见，typedef名“保存”了声明作为一种类型，以便稍后用于其他声明。这样你可以节省一些敲击键盘的次数。此外，使用typedef的声明仍然是声明，你不仅限于上述示例：

```
t0 (*pf1);
```

等同于：

```
int (*(**pf1())())[5];
```

第44.5节：使用全局常量

头文件可以用来声明全局使用的只读资源，比如字符串表。

将需要使用它们的任何文件（“翻译单元”）包含的内容声明在单独的头文件中。使用同一个头文件声明一个相关的枚举来标识所有字符串资源是很方便的：

resources.h:

```
#ifndef RESOURCES_H  
#define RESOURCES_H  
  
typedef enum { /* 定义描述可能有效资源ID的类型。 */  
    RESOURCE_UNDEFINED = -1, /* 用于初始化任何 EnumResourceID 类型的变量 */
```

```
int a4[5]; /* here a4 shouldn't be accessed using the index 5 later on */
```

But it should be accessible by all other indexes lower than 5. Examples:

```
a4[0], a4[1]; a4[4];
```

a4[5] will result into UB. More information about arrays can be found here.

```
int (*a5)[5](); /* here a4 could be applied indirection  
indexed up to (but not including) 5  
and called */
```

Unfortunately for us, although syntactically possible, the declaration of a5 is forbidden by the current standard.

Section 44.4: Typedef

Typedefs are declarations which have the keyword **typedef** in front and before the type. E.g.:

```
typedef int (*(*t0())())[5];
```

(you can technically put the typedef after the type too - like this int typedef (*(*t0())())[5]; but this is discouraged)

The above declarations declares an identifier for a typedef name. You can use it like this afterwards:

```
t0 pf;
```

Which will have the same effect as writing:

```
int (*(*pf())())[5];
```

As you can see the typedef name "saves" the declaration as a type to use later for other declarations. This way you can save some keystrokes. Also as declaration using **typedef** is still a declaration you are not limited only by the above example:

```
t0 (*pf1);
```

Is the same as:

```
int (*(**pf1())())[5];
```

Section 44.5: Using Global Constants

Headers may be used to declare globally used read-only resources, like string tables for example.

Declare those in a separate header which gets included by any file ("Translation Unit") which wants to make use of them. It's handy to use the same header to declare a related enumeration to identify all string-resources:

resources.h:

```
#ifndef RESOURCES_H  
#define RESOURCES_H  
  
typedef enum { /* Define a type describing the possible valid resource IDs. */  
    RESOURCE_UNDEFINED = -1, /* To be used to initialise any EnumResourceID typed variable to be
```

标记为“未使用”、“不在列表中”、“未定义”，什么鬼。

在应用层面会显示未初始化，而不是在语言层面。

可以说是已初始化的未初始化 :-)

就像指针的 NULL ;-)/

RESOURCE_UNKNOWN = 0, /* 如果应用程序使用某些资源ID,
们没有定义相应的表项时使用，作为回退，
*/

/* 以下标识我们定义的资源：*/

RESOURCE_OK,
RESOURCE_CANCEL,
RESOURCE_ABORT,
/* 在此插入更多内容。 */

RESOURCE_MAX /* 定义的资源最大数量。*/
} EnumResourceID;

extern const char * const resources[RESOURCE_MAX]; /* 声明，向包含此文件的任何人保证
第一个 const 保证字符串不会改变，
在链接时此符号将存在。

第二个 const 保证字符串表项
在初始化时设置后不会突然指向其他位置。*/

#endif

为了实际定义资源，创建了一个相关的 .c 文件，即另一个翻译单元，包含了在相关头文件 (.h) 中声明的实际实例：

resources.c:

#include "resources.h" /* 为确保声明与定义之间的冲突被编译器识别，
件包含到实现定义的翻译单元(.c文件)中。*/

/* 定义资源。遵守 resources.h 中的承诺。*/
const char * const resources[RESOURCE_MAX] = {
 "<unknown>",
 "确定",
 "取消",
 "中止"
};

将声明头文

使用该程序的示例代码如下：

main.c :

```
#include <stdlib.h> /* 用于 EXIT_SUCCESS */  
#include <stdio.h>  
  
#include "resources.h"  
  
int main(void)  
{  
    EnumResourceID resource_id = RESOURCE_UNDEFINED;  
  
    while ((++resource_id) < RESOURCE_MAX)  
    {
```

marked as "not in use", "not in list", "undefined", wtf.
Will say un-initialised on application level, not on language level.

Initialised uninitialized, so to say ;-)

Its like NULL for pointers ;-)/

RESOURCE_UNKNOWN = 0, /* To be used if the application uses some resource ID,
for which we do not have a table entry defined, a fall back in
case we _need_ to display something, but do not find anything
appropriate. */

/* The following identify the resources we have defined: */

RESOURCE_OK,
RESOURCE_CANCEL,
RESOURCE_ABORT,
/* Insert more here. */

RESOURCE_MAX /* The maximum number of resources defined. */
} EnumResourceID;

extern const char * const resources[RESOURCE_MAX]; /* Declare, promise to anybody who includes
this, that at linkage-time this symbol will be around.

The 1st const guarantees the strings will not change,
the 2nd const guarantees the string-table entries
will never suddenly point somewhere else as set during
initialisation. */

#endif

To actually define the resources created a related .c-file, that is another translation unit holding the actual instances
of the what had been declared in the related header (.h) file:

resources.c:

#include "resources.h" /* To make sure clashes between declaration and definition are
recognised by the compiler include the declaring header into
the implementing, defining translation unit (.c file).

/* Define the resources. Keep the promise made in resources.h. */
const char * const resources[RESOURCE_MAX] = {
 "<unknown>",
 "OK",
 "Cancel",
 "Abort"
};

A program using this could look like this:

main.c:

```
#include <stdlib.h> /* for EXIT_SUCCESS */  
#include <stdio.h>  
  
#include "resources.h"  
  
int main(void)  
{  
    EnumResourceID resource_id = RESOURCE_UNDEFINED;  
  
    while ((++resource_id) < RESOURCE_MAX)  
    {
```

```

printf("资源 ID: %d, 资源: '%s'", resource_id, resources[resource_id]);}

return EXIT_SUCCESS;
}

```

使用 GCC 编译上述三个文件，并链接生成程序文件 main，例如使用以下命令：

```
gcc -Wall -Wextra -pedantic -Wconversion -g main.c resources.c -o main
```

(使用这些-Wall-Wextra-pedantic-Wconversion让编译器变得非常严格，这样你在发布代码到Stack Overflow、发布到世界，甚至部署到生产环境之前，就不会遗漏任何东西)

运行创建的程序：

```
$ ./main
```

并得到：

```

资源 ID: 0, 资源: "
资源 ID: 1, 资源: '确定'
资源 ID: 2, 资源: '取消'
资源 ID: 3, 资源: '中止'

```

第44.6节：使用右-左或螺旋规则解析C语言声明

“右-左”规则是一条完全规则的C语言声明解析规则。它在创建声明时也非常有用。

在声明中遇到符号时按顺序读取.....

* 表示 “指针”	- 总是在左侧
[] 表示 “数组”	- 总是在右侧
() 表示 “返回函数”	- 总是在右侧

如何应用该规则

步骤 1

找到标识符。这是你的起点。然后对自己说，“标识符是”。你已经开始了声明。

步骤 2

查看标识符右侧的符号。例如，如果你发现()，那么你就知道这是一个函数的声明。此时你可以说“标识符是返回.....的函数”。或者如果你发现[]，你会说“标识符是.....的数组”。继续向右查看，直到没有符号或者遇到右括号)为止。（如果遇到左括号(，那就是()符号的开始，即使括号中间有内容。下面会详细说明。）

步骤 3

查看标识符左侧的符号。如果它不是我们上面列出的符号之一（比如“int”之类的），就直接说出来。否则，使用上面的表格将其翻译成英文。继续向左查看，直到没有符号或遇到

```

printf("resource ID: %d, resource: '%s'\n", resource_id, resources[resource_id]);
}

return EXIT_SUCCESS;
}

```

Compile the three file above using GCC, and link them to become the program file main for example using this:

```
gcc -Wall -Wextra -pedantic -Wconversion -g main.c resources.c -o main
```

(use these -Wall -Wextra -pedantic -Wconversion to make the compiler really picky, so you don't miss anything before posting the code to SO, will say the world, or even worth deploying it into production)

Run the program created:

```
$ ./main
```

And get:

```

resource ID: 0, resource: ''
resource ID: 1, resource: 'OK'
resource ID: 2, resource: 'Cancel'
resource ID: 3, resource: 'Abort'

```

Section 44.6: Using the right-left or spiral rule to decipher C declaration

The "right-left" rule is a completely regular rule for deciphering C declarations. It can also be useful in creating them.

Read the symbols as you encounter them in the declaration...

* as “pointer to”	- always on the left side
[] as “array of”	- always on the right side
() as “function returning”	- always on the right side

How to apply the rule

STEP 1

Find the identifier. This is your starting point. Then say to yourself, "identifier is." You've started your declaration.

STEP 2

Look at the symbols on the right of the identifier. If, say, you find () there, then you know that this is the declaration for a function. So you would then have "identifier is function returning". Or if you found a [] there, you would say "identifier is array of". Continue right until you run out of symbols OR hit a right parenthesis). (If you hit a left parenthesis (, that's the beginning of a () symbol, even if there is stuff in between the parentheses. More on that below.)

STEP 3

Look at the symbols to the left of the identifier. If it is not one of our symbols above (say, something like "int"), just say it. Otherwise, translate it into English using that table above. Keep going left until you run out of symbols OR hit

左括号(。

现在重复步骤2和3，直到你完成你的声明。

以下是一些示例：

```
int *p[];
```

首先，找到标识符：

```
int *p[ ];  
^
```

"p 是"

现在，向右移动直到遇到符号结束或右括号为止。

```
int *p[ ];  
^^
```

"p 是数组的"

不能再向右移动了（符号结束），所以向左移动并找到：

```
int *p[ ];  
^
```

"p 是指向的指针数组"

继续向左移动并找到：

```
int *p[ ];  
^^^
```

"p 是指向 int 的指针数组".

(或者"p 是一个数组，其中每个元素的类型是指向 int 的指针")

另一个例子：

```
int *(*func())();
```

找出标识符。

```
int *(*func())();  
^^^^^
```

"func 是"

向右移动。

```
int *(*func())();  
^
```

"func 是返回的函数"

a left parenthesis (.

Now repeat steps 2 and 3 until you've formed your declaration.

Here are some examples:

```
int *p[ ];
```

First, find identifier:

```
int *p[ ];  
^
```

"p is"

Now, move right until out of symbols or right parenthesis hit.

```
int *p[ ];  
^^
```

"p is array of"

Can't move right anymore (out of symbols), so move left and find:

```
int *p[ ];  
^
```

"p is array of pointer to"

Keep going left and find:

```
int *p[ ];  
^^^
```

"p is array of pointer to int".

(or "p is an array where each element is of type pointer to int")

Another example:

```
int *(*func())();
```

Find the identifier.

```
int *(*func())();  
^^^^^
```

"func is"

Move right.

```
int *(*func())();  
^
```

"func is function returning"

因为右括号，不能再向右移动，所以向左移动。

```
int *(*func())();  
^
```

"func 是返回指针的函数"

因为左括号，不能再向左移动了，所以继续向右移动。

```
int *(*func())();  
^^
```

"func 是返回函数指针的函数"

因为符号用完了，不能再向右移动了，所以向左移动。

```
int *(*func())();  
^
```

"func 是返回指向函数指针的函数的函数"

最后，继续向左移动，因为右边没有东西了。

```
int *(*func())();  
^^^
```

"func 是返回指向返回 int 的函数指针的函数"。

如你所见，这条规则非常有用。你也可以用它在创建声明时自我检查，并提示你下一步该放置哪个符号以及是否需要括号。

有些声明看起来比实际复杂得多，这是因为数组大小和原型中的参数列表。如果你看到 [3]，读作“大小为 3 的数组”。如果你看到 (char *,int)，读作“期望 (char ,int) 参数并返回...”的函数。。

这是一个有趣的例子：

```
int (*(*fun_one)(char *,double))[9][20];
```

我不会逐步解析这个例子。

**"fun_one 是一个指向函数的指针，该函数期望参数为 (char ,double)，并返回指向大小为 9 的数组中大小为 20 的数组的 int 指针。"

正如你所见，如果去掉数组大小和参数列表，情况就没那么复杂了：

```
int (*(*fun_one)())[][];
```

你可以先这样解析，然后再补充数组大小和参数列表。

最后几点说明：

使用这个规则很可能会写出非法声明，因此需要了解 C 语言中什么是合法的。

例如，如果上面写成：

```
int *((*fun_one)())[][];
```

Can't move right anymore because of the right parenthesis, so move left.

```
int *(*func())();  
^
```

"func is function returning pointer to"

Can't move left anymore because of the left parenthesis, so keep going right.

```
int *(*func())();  
^^
```

"func is function returning pointer to function returning"

Can't move right anymore because we're out of symbols, so go left.

```
int *(*func())();  
^
```

"func is function returning pointer to function returning pointer to"

And finally, keep going left, because there's nothing left on the right.

```
int *(*func())();  
^^^
```

"func is function returning pointer to function returning pointer to int".

As you can see, this rule can be quite useful. You can also use it to sanity check yourself while you are creating declarations, and to give you a hint about where to put the next symbol and whether parentheses are required.

Some declarations look much more complicated than they are due to array sizes and argument lists in prototype form. If you see [3]，that's read as "array (size 3) of...". If you see (char *,int) that's read as *"function expecting (char ,int) and returning...".

Here's a fun one:

```
int (*(*fun_one)(char *,double))[9][20];
```

I won't go through each of the steps to decipher this one.

**"fun_one is pointer to function expecting (char ,double) and returning pointer to array (size 9) of array (size 20) of int."

As you can see, it's not as complicated if you get rid of the array sizes and argument lists:

```
int (*(*fun_one)())[][];
```

You can decipher it that way, and then put in the array sizes and argument lists later.

Some final words:

It is quite possible to make illegal declarations using this rule, so some knowledge of what's legal in C is necessary. For instance, if the above had been:

```
int *((*fun_one)())[][];
```

它将被读作“`fun_one` 是指向返回指针数组的函数的指针”。由于函数不能返回数组，只能返回指向数组的指针，因此该声明是非法的。

非法组合包括：

`[]()` - 不能有函数数组
`()()` - 不能有返回函数的函数
`()[]` - 不能有返回数组的函数

在上述所有情况下，你需要一对括号将左侧的*符号绑定在这些()和[]右侧符号之间，才能使声明合法。

以下是更多示例：

合法

<code>int i;</code>	一个int
<code>int *p;</code>	一个int指针（指向一个int的指针）
<code>int a[];</code>	一个int数组
<code>int f();</code>	一个返回int的函数
<code>int **pp;</code>	一个指向int指针的指针（指向指向int的指针的指针）
<code>int (*pa)[];</code>	一个指向int数组的指针
<code>int (*pf)();</code>	一个指向返回int的函数的指针
<code>int *ap[];</code>	一个int指针数组（指向int的指针数组）
<code>int aa[][];</code>	一个int数组的数组
<code>int *fp();</code>	一个返回int指针的函数
<code>int ***ppp;</code>	一个指向指向int指针的指针的指针
<code>int (**ppa)[];</code>	一个指向指向int数组的指针的指针
<code>int (**ppf)();</code>	一个指向指向返回int的函数的指针的指针
<code>int *(*pap)[];</code>	一个指向int指针数组的指针
<code>int (*paa)[][];</code>	指向整型数组的指针数组的指针
<code>int *(pfp)();</code>	指向返回整型指针的函数的指针
<code>int **app[];</code>	指向整型指针的指针数组
<code>int (*apa)[][];</code>	指向整型数组的指针数组
<code>int (*apf)()();</code>	指向返回整型的函数的指针数组
<code>int *aap[][];</code>	整型指针的二维数组
<code>int aaa[][][];</code>	三维整型数组
<code>int **fpp();</code>	返回指向整型指针的函数
<code>int (*fpa)()[];</code>	返回指向整型数组的指针的函数
<code>int (*fpf)()();</code>	返回指向返回整型的函数的指针的函数

it would have read “`fun_one` is pointer to function returning array of pointer to int”. Since a function cannot return an array, but only a pointer to an array, that declaration is illegal.

Illegal combinations include:

`[]()` - cannot have an array of functions
`()()` - cannot have a function that returns a function
`()[]` - cannot have a function that returns an array

In all the above cases, you would need a set of parentheses to bind a * symbol on the left between these () and [] right-side symbols in order for the declaration to be legal.

Here are some more examples:

Legal

<code>int i;</code>	an int
<code>int *p;</code>	an int pointer (ptr to an int)
<code>int a[];</code>	an array of ints
<code>int f();</code>	a function returning an int
<code>int **pp;</code>	a pointer to an int pointer (ptr to a ptr to an int)
<code>int (*pa)[];</code>	a pointer to an array of ints
<code>int (*pf)();</code>	a pointer to a function returning an int
<code>int *ap[];</code>	an array of int pointers (array of ptrs to ints)
<code>int aa[][];</code>	an array of arrays of ints
<code>int *fp();</code>	a function returning an int pointer
<code>int ***ppp;</code>	a pointer to a pointer to an int pointer
<code>int (**ppa)[];</code>	a pointer to a pointer to an array of ints
<code>int (**ppf)();</code>	a pointer to a pointer to a function returning an int
<code>int *(*pap)[];</code>	a pointer to an array of int pointers
<code>int (*paa)[][];</code>	a pointer to an array of arrays of ints
<code>int *(pfp)();</code>	a pointer to a function returning an int pointer
<code>int **app[];</code>	an array of pointers to int pointers
<code>int (*apa)[][];</code>	an array of pointers to arrays of ints
<code>int (*apf)()();</code>	an array of pointers to functions returning an int
<code>int *aap[][];</code>	an array of arrays of int pointers
<code>int aaa[][][];</code>	an array of arrays of arrays of int
<code>int **fpp();</code>	a function returning a pointer to an int pointer
<code>int (*fpa)()[];</code>	a function returning a pointer to an array of ints
<code>int (*fpf)()();</code>	a function returning a pointer to a function returning an int

Illegal

<code>int af[]();</code>	an array of functions returning an int
<code>int fa()[];</code>	a function returning an array of ints
<code>int ff()();</code>	a function returning a function returning an int
<code>int (*pfa)()[];</code>	a pointer to a function returning an array of ints
<code>int aaf[][]();</code>	an array of arrays of functions returning an int
<code>int (*paf)()[];</code>	a pointer to a an array of functions returning an int
<code>int (*pff)()();</code>	a pointer to a function returning a function returning an int
<code>int *afp[]();</code>	an array of functions returning int pointers
<code>int afa[][]();</code>	an array of functions returning an array of ints
<code>int aff[][]();</code>	an array of functions returning functions returning an int
<code>int *fap()[];</code>	a function returning an array of int pointers
<code>int faa[][][];</code>	a function returning an array of arrays of ints
<code>int faf[][]();</code>	a function returning an array of functions returning an int
<code>int *ffp()();</code>	a function returning a function returning an int pointer

非法

<code>int af[]();</code>	返回整型的函数数组
<code>int fa()[];</code>	返回整型数组的函数
<code>int ff()();</code>	返回返回整型的函数的函数
<code>int (*pfa)()[];</code>	指向返回整型数组的函数的指针
<code>int aaf[][]();</code>	返回整型的函数的二维数组
<code>int (*paf)()[];</code>	指向返回整型的函数数组的指针
<code>int (*pff)()();</code>	指向返回返回整型的函数的函数的指针
<code>int *afp[]();</code>	返回整型指针的函数数组
<code>int afa[][]();</code>	返回整型数组的函数数组
<code>int aff[][]();</code>	返回返回整型的函数的函数数组
<code>int *fap()[];</code>	返回整型指针数组的函数
<code>int faa[][][];</code>	返回整型二维数组的函数
<code>intfaf[][]();</code>	返回返回整型的函数数组的函数
<code>int *ffp()();</code>	返回返回整型指针的函数的函数

来源: http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html

Source: http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html

第45章：结构体填充与打包

默认情况下，C编译器布局结构体，使得每个成员都能快速访问，避免因“未对齐访问”而产生的性能损失，这在RISC机器（如DEC Alpha和部分ARM CPU）上是个问题。

根据CPU架构和编译器的不同，结构体在内存中占用的空间可能比其组成成员大小之和还要大。编译器可以在成员之间或结构体末尾添加填充，但不能在开头添加。

打包覆盖默认的填充。

第45.1节：结构体打包

默认情况下，C语言中的结构体会进行填充。如果你想避免这种行为，必须显式地请求。在GCC中是`__attribute__((__packed__))`。考虑在64位机器上的这个例子：

```
struct foo {
    char *p; /* 8字节 */
    char c; /* 1字节 */
    long x; /* 8字节 */
};
```

该结构体会自动填充以达到8字节对齐，看起来像这样：

```
struct foo {
    char *p; /* 8字节 */
    char c; /* 1字节 */

    char pad[7]; /* 编译器添加的7字节 */

    long x; /* 8字节 */
};
```

因此，`sizeof(struct foo)` 会返回24而不是17。这是因为64位编译器每次从内存中以8字节字长读写数据，很明显当写入`char c`;一个字节时，会先取出完整的8字节（即一个字），只使用其中的第一个字节，其余7个字节保持空闲且不可用于任何读写操作，这就是结构体填充的原因。

结构体打包

但是如果你添加属性`packed`，编译器将不会添加填充：

```
struct __attribute__((__packed__)) foo {
    char *p; /* 8 字节 */
    char c; /* 1 字节 */
    long x; /* 8 字节 */
};
```

现在 `sizeof(struct foo)` 将返回 17。

通常，打包结构用于：

- 节省空间。

Chapter 45: Structure Padding and Packing

By default, C compilers lay out structures so that each member can be accessed fast, without incurring penalties for 'unaligned access, a problem with RISC machines such as the DEC Alpha, and some ARM CPUs.

Depending on the CPU architecture and the compiler, a structure may occupy more space in memory than the sum of the sizes of its component members. The compiler can add padding between members or at the end of the structure, but not at the beginning.

Packing overrides the default padding.

Section 45.1: Packing structures

By default structures are padded in C. If you want to avoid this behaviour, you have to explicitly request it. Under GCC it's `__attribute__((__packed__))`. Consider this example on a 64-bit machine:

```
struct foo {
    char *p; /* 8 bytes */
    char c; /* 1 byte */
    long x; /* 8 bytes */
};
```

The structure will be automatically padded to have 8-byte alignment and will look like this:

```
struct foo {
    char *p; /* 8 bytes */
    char c; /* 1 byte */

    char pad[7]; /* 7 bytes added by compiler */

    long x; /* 8 bytes */
};
```

So `sizeof(struct foo)` will give us 24 instead of 17. This happened because of a 64 bit compiler read/write from/to Memory in 8 bytes of word in each step and obvious when try to write `char c`; a one byte in memory a complete 8 bytes (i.e. word) fetched and consumes only first byte of it and its seven successive of bytes remains empty and not accessible for any read and write operation for structure padding.

Structure packing

But if you add the attribute `packed`, the compiler will not add padding:

```
struct __attribute__((__packed__)) foo {
    char *p; /* 8 bytes */
    char c; /* 1 byte */
    long x; /* 8 bytes */
};
```

Now `sizeof(struct foo)` will return 17.

Generally packed structures are used:

- To save space.

- 格式化数据结构以通过网络传输，而不依赖于网络中每个节点的架构对齐方式。

必须考虑到某些处理器，如 ARM Cortex-M0，不允许非对齐内存访问；在这种情况下，结构体打包可能导致未定义行为并可能导致 CPU 崩溃。

第45.2节：结构体填充

假设这个struct是在32位编译器下定义和编译的：

```
结构体 test_32 {
    整数 a;      // 4 字节
    短整数 b;    // 2 字节
    整数 c;      // 4 字节
} str_32;
```

我们可能会期望这个struct只占用10字节的内存，但通过打印`sizeof(str_32)`我们看到它使用了12字节。

这是因为编译器为了快速访问会对变量进行对齐。一个常见的模式是，当基类型占用N字节（N是2的幂，如1、2、4、8、16—且很少更大）时，变量应当在N字节边界（N字节的倍数）上对齐。

对于显示的结构体，`sizeof(int) == 4`且`sizeof(short) == 2`，常见的布局是：

- int a；存储在偏移量0；大小4。
- short b；存储在偏移量4；大小2。
- 未命名的填充位于偏移量6；大小2。
- int c；存储在偏移量8；大小4。

因此，`struct test_32` 占用12字节内存。在此示例中，没有尾部填充。

编译器会确保任何`struct test_32`变量都从4字节边界开始存储，以便结构体内的成员能够正确对齐，实现快速访问。内存分配函数如`malloc()`、`calloc()`和`realloc()`都必须确保返回的指针对齐良好，适用于任何数据类型，因此动态分配的结构体也会正确对齐。

你可能会遇到一些奇怪的情况，比如在64位Intel x86_64处理器（例如运行macOS Sierra或Mac OS X的Intel Core i7 Mac）上，当以32位模式编译时，编译器将`double`对齐到4字节边界；但在同一硬件上以64位模式编译时，编译器将`double`对齐到8字节边界。

- To format a data structure to transmit over network without depending on each architecture alignment of each node of the network.

It must be taken in consideration that some processors such as the ARM Cortex-M0 do not allow unaligned memory access; in such cases, structure packing can lead to *undefined behaviour* and can crash the CPU.

Section 45.2: Structure padding

Suppose this `struct` is defined and compiled with a 32 bit compiler:

```
struct test_32 {
    int a;      // 4 byte
    short b;    // 2 byte
    int c;      // 4 byte
} str_32;
```

We might expect this `struct` to occupy only 10 bytes of memory, but by printing `sizeof(str_32)` we see it uses 12 bytes.

This happened because the compiler aligns variables for fast access. A common pattern is that when the base type occupies N bytes (where N is a power of 2 such as 1, 2, 4, 8, 16 — and seldom any bigger), the variable should be aligned on an N-byte boundary (a multiple of N bytes).

For the structure shown with `sizeof(int) == 4` and `sizeof(short) == 2`, a common layout is:

- `int a`; stored at offset 0; size 4.
- `short b`; stored at offset 4; size 2.
- unnamed padding at offset 6; size 2.
- `int c`; stored at offset 8; size 4.

Thus `struct test_32` occupies 12 bytes of memory. In this example, there is no trailing padding.

The compiler will ensure that any `struct test_32` variables are stored starting on a 4-byte boundary, so that the members within the structure will be properly aligned for fast access. Memory allocation functions such as `malloc()`, `calloc()` and `realloc()` are required to ensure that the pointer returned is sufficiently well aligned for use with any data type, so dynamically allocated structures will be properly aligned too.

You can end up with odd situations such as on a 64-bit Intel x86_64 processor (e.g. Intel Core i7 — a Mac running macOS Sierra or Mac OS X), where when compiling in 32-bit mode, the compilers place `double` aligned on a 4-byte boundary; but, on the same hardware, when compiling in 64-bit mode, the compilers place `double` aligned on an 8-byte boundary.

第46章：内存管理

名称	描述
大小 (<code>malloc</code> , <code>realloc</code> 和 <code>aligned_malloc</code>)	内存的总大小，单位为字节。对于 <code>aligned_malloc</code> ，大小必须是对齐值的整数倍。
大小 (<code>calloc</code>)	每个元素的大小
元素数量	元素个数
指针	指向之前由 <code>malloc</code> 、 <code>calloc</code> 、 <code>realloc</code> 或 <code>aligned_malloc</code> 分配的内存的指针
对齐	已分配内存的对齐

为了管理动态分配的内存，标准C库提供了函数 `malloc()`、`calloc()`、`realloc()` 和 `free()`。在C99及以后版本中，还有 `aligned_malloc()`。一些系统还提供了 `alloca()`。

第46.1节：内存分配

标准分配

C语言的动态内存分配函数定义在 `<stdlib.h>` 头文件中。如果想动态分配一个对象的内存空间，可以使用以下代码：

```
int *p = malloc(10 * sizeof *p);
if (p == NULL)
{
    perror("malloc() failed");
    return -1;
}
```

这段代码计算了十个 `int` 类型在内存中占用的字节数，然后向 `malloc` 请求这么多字节，并将结果（即刚刚通过 `malloc` 创建的内存块的起始地址）赋值给名为 `p` 的指针。

使用 `sizeof` 来计算请求的内存大小是一种良好习惯，因为 `sizeof` 的结果是实现定义的（除了字符类型，即 `char`、`signed char` 和 `unsigned char`，这些类型的 `sizeof` 定义为始终为1）。

因为 `malloc` 可能无法满足请求，它可能返回空指针。重要的是检查这一点，以防止后续尝试解引用空指针。

使用 `malloc()` 动态分配的内存可以使用 `realloc()` 调整大小，或者在不再需要时，使用 `free()` 释放。

或者，声明 `int array[10];` 也会分配相同大小的内存。但是，如果它在函数内部声明且没有使用 `static` 关键字，它只在声明它的函数及其调用的函数中可用（因为数组会分配在栈上，函数返回时空间会被释放以供重用）。另外，如果在函数内部使用 `static` 定义，或者在任何函数外部定义，则其生命周期为程序的生命周期。指针也可以从函数返回，但 C 语言中的函数不能返回数组。

清零内存

`malloc` 返回的内存可能未初始化为合理的值，应注意使用 `memset` 将内存清零，或立即复制合适的值进去。或者，`calloc` 返回一块所有位都初始化为0的指定大小的内存块。

Chapter 46: Memory management

name	description
<code>size (malloc, realloc and aligned_malloc)</code>	total size of the memory in bytes. For <code>aligned_malloc</code> the size must be a integral multiple of alignment.
<code>size (calloc)</code>	size of each element
<code>nElements</code>	number of elements
<code>ptr</code>	pointer to allocated memory previously returned by <code>malloc</code> , <code>calloc</code> , <code>realloc</code> or <code>aligned_malloc</code>
<code>alignment</code>	alignment of allocated memory

For managing dynamically allocated memory, the standard C library provides the functions `malloc()`, `calloc()`, `realloc()` and `free()`. In C99 and later, there is also `aligned_malloc()`. Some systems also provide `alloca()`.

Section 46.1: Allocating Memory

Standard Allocation

The C dynamic memory allocation functions are defined in the `<stdlib.h>` header. If one wishes to allocate memory space for an object dynamically, the following code can be used:

```
int *p = malloc(10 * sizeof *p);
if (p == NULL)
{
    perror("malloc() failed");
    return -1;
}
```

This computes the number of bytes that ten `int`s occupy in memory, then requests that many bytes from `malloc` and assigns the result (i.e., the starting address of the memory chunk that was just created using `malloc`) to a pointer named `p`.

It is good practice to use `sizeof` to compute the amount of memory to request since the result of `sizeof` is implementation defined (except for *character types*, which are `char`, `signed char` and `unsigned char`, for which `sizeof` is defined to always give 1).

Because `malloc` might not be able to service the request, it might return a null pointer. It is important to check for this to prevent later attempts to dereference the null pointer.

Memory dynamically allocated using `malloc()` may be resized using `realloc()` or, when no longer needed, released using `free()`.

Alternatively, declaring `int array[10];` would allocate the same amount of memory. However, if it is declared inside a function without the keyword `static`, it will only be usable within the function it is declared in and the functions it calls (because the array will be allocated on the stack and the space will be released for reuse when the function returns). Alternatively, if it is defined with `static` inside a function, or if it is defined outside any function, then its lifetime is the lifetime of the program. Pointers can also be returned from a function, however a function in C can not return an array.

Zeroed Memory

The memory returned by `malloc` may not be initialized to a reasonable value, and care should be taken to zero the memory with `memset` or to immediately copy a suitable value into it. Alternatively, `calloc` returns a block of the

这不一定等同于浮点数零或空指针常量的表示。

```
int *p = calloc(10, sizeof *p);
if (p == NULL)
{
    perror("calloc() failed");
    return -1;
}
```

关于calloc的说明：大多数（常用）实现会对calloc()进行性能优化，因此它比先调用malloc()再调用memset()更快，尽管最终效果相同。

对齐内存

版本 ≥ C11

C11 引入了一个新函数 aligned_alloc()，用于分配具有指定对齐方式的空间。如果需要分配的内存必须在某些 malloc() 或 calloc() 无法满足的边界上对齐时，可以使用该函数。malloc() 和 calloc() 函数分配的内存适合任何对象类型（即对齐方式为 alignof(max_align_t)）。但使用 aligned_alloc() 可以请求更大的对齐方式。

```
/* 分配1024字节，按256字节对齐。*/
char *ptr = aligned_alloc(256, 1024);
if (ptr) {
    perror("aligned_alloc()");
    return -1;
}
free(ptr);
```

C11标准规定了两个限制：1) 请求的大小（第二个参数）必须是对齐（第一个参数）的整数倍；2) 对齐的值应为实现支持的有效对齐。未满足任一条件将导致未定义行为。

第46.2节：释放内存

可以通过调用free()释放动态分配的内存。

```
int *p = malloc(10 * sizeof *p); /* 分配内存 */
if (p == NULL)
{
    perror("malloc failed");
    return -1;
}

free(p); /* 释放内存 */
/* 注意，在调用 free(p) 之后，即使使用指针 p 的 *值* 也具有未定义行为，直到对其进行存储一个新值。 */ /* 重新使用/重新赋值指针本身 */

int i = 42;
p = &i; /* 这是有效的，具有定义的行为 */
```

调用 free() 后，指针 p 指向的内存会被回收（由 libc 实现或底层操作系统），因此通过 p 访问已释放的内存块将导致未定义行为。指向已释放内存元素的指针通常称为 dangling pointers（悬空指针），存在安全风险。

此外，C 标准规定，即使访问悬空指针的值也具有未定义行为。

desired size where all bits are initialized to 0. This need not be the same as the representation of floating-point zero or a null pointer constant.

```
int *p = calloc(10, sizeof *p);
if (p == NULL)
{
    perror("calloc() failed");
    return -1;
}
```

A note on calloc: Most (commonly used) implementations will optimise calloc() for performance, so it will be faster than calling malloc(), then memset(), even though the net effect is identical.

Aligned Memory

Version ≥ C11

C11 introduced a new function aligned_alloc() which allocates space with the given alignment. It can be used if the memory to be allocated is needed to be aligned at certain boundaries which can't be satisfied by malloc() or calloc(). malloc() and calloc() functions allocate memory that's suitably aligned for any object type (i.e. the alignment is alignof(max_align_t)). But with aligned_alloc() greater alignments can be requested.

```
/* Allocates 1024 bytes with 256 bytes alignment. */
char *ptr = aligned_alloc(256, 1024);
if (ptr) {
    perror("aligned_alloc()");
    return -1;
}
free(ptr);
```

The C11 standard imposes two restrictions: 1) the size (second argument) requested must be an integral multiple of the alignment (first argument) and 2) the value of alignment should be a valid alignment supported by the implementation. Failure to meet either of them results in undefined behavior.

Section 46.2: Freeing Memory

It is possible to release dynamically allocated memory by calling free().

```
int *p = malloc(10 * sizeof *p); /* allocation of memory */
if (p == NULL)
{
    perror("malloc failed");
    return -1;
}

free(p); /* release of memory */
/* note that after free(p), even using the *value* of the pointer p
   has undefined behavior, until a new value is stored into it. */

/* reusing/re-purposing the pointer itself */
int i = 42;
p = &i; /* This is valid, has defined behaviour */
```

The memory pointed to by p is reclaimed (either by the libc implementation or by the underlying OS) after the call to free(), so accessing that freed memory block via p will lead to undefined behavior. Pointers that reference memory elements that have been freed are commonly called [dangling pointers](#), and present a security risk. Furthermore, the C standard states that even accessing the value of a dangling pointer has undefined behavior.

请注意，指针 p 本身可以如上所示重新赋值。

请注意，您只能对直接由 `malloc()`、`calloc()`、`realloc()` 和 `aligned_alloc()` 函数返回的指针调用 `free()`，或者文档说明内存是以这种方式分配的（如 `strdup()` 是显著的例子）。释放一个指针，

- 是通过对变量使用 `&` 运算符获得的，或者
- 位于已分配块的中间，

这是禁止的。此类错误通常不会被编译器诊断，但会导致程序执行进入未定义状态。

有两种常见策略可以防止此类未定义行为的发生。

第一种且更可取的方法是简单的一—当不再需要时，让p本身停止存在，例如：

```
if (需要某些东西())
{
    int *p = malloc(10 * sizeof *p);
    if (p == NULL)
    {
        perror("malloc failed");
        return -1;
    }

    /* 对 p 执行所需的操作 */

    free(p);
}
```

通过在包含块结束之前（即 `}`）直接调用 `free()`，p 本身将不再存在。编译器会在尝试使用 p 后报编译错误。

第二种方法是在释放指针所指向的内存后，同时使指针本身失效：

```
free(p);
p = NULL; // 你也可以用 0 替换 NULL
```

采用这种方法的理由：

- 在许多平台上，尝试解引用空指针会导致立即崩溃：段错误。
这样，我们至少可以获得一个堆栈跟踪，指向被释放后仍被使用的变量。

如果不将指针设置为NULL，我们就会有悬空指针。程序很可能仍然会崩溃，但会在稍后发生，因为指针所指向的内存会被悄无声息地破坏。这类错误难以追踪，因为它们可能导致调用栈与最初的问题完全无关。

因此，这种方法遵循了快速失败（fail-fast）概念。

- 释放空指针是安全的。C标准规定`free(NULL)`不会产生任何效果：

`free`函数会使ptr指向的空间被释放，也就是说，该空间可供后续分配使用。如果ptr是空指针，则不执行任何操作。否则，如果参数不

Note that the pointer p itself can be re-purposed as shown above.

Please note that you can only call `free()` on pointers that have directly been returned from the `malloc()`, `calloc()`, `realloc()` and `aligned_alloc()` functions, or where documentation tells you the memory has been allocated that way (functions like `strdup()` are notable examples). Freeing a pointer that is,

- obtained by using the `&` operator on a variable, or
- in the middle of an allocated block,

is forbidden. Such an error will usually not be diagnosed by your compiler but will lead the program execution in an undefined state.

There are two common strategies to prevent such instances of undefined behavior.

The first and preferable is simple - have p itself cease to exist when it is no longer needed, for example:

```
if (something_is_needed())
{
    int *p = malloc(10 * sizeof *p);
    if (p == NULL)
    {
        perror("malloc failed");
        return -1;
    }

    /* do whatever is needed with p */

    free(p);
}
```

By calling `free()` directly before the end of the containing block (i.e. the `}`), p itself ceases to exist. The compiler will give a compilation error on any attempt to use p after that.

A second approach is to also invalidate the pointer itself after releasing the memory to which it points:

```
free(p);
p = NULL; // you may also use 0 instead of NULL
```

Arguments for this approach:

- On many platforms, an attempt to dereference a null pointer will cause instant crash: Segmentation fault.
Here, we get at least a stack trace pointing to the variable that was used after being freed.

Without setting pointer to NULL we have dangling pointer. The program will very likely still crash, but later, because the memory to which the pointer points will silently be corrupted. Such bugs are difficult to trace because they can result in a call stack that completely unrelated to the initial problem.

This approach hence follows the [fail-fast concept](#).

- It is safe to free a null pointer. The [C Standard specifies](#) that `free(NULL)` has no effect:

`free` function causes the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a null pointer, no action occurs. Otherwise, if the argument does not

匹配之前由calloc、malloc或realloc函数返回的指针，或者该空间已被调用free或realloc释放，则行为未定义。

- 有时第一种方法无法使用（例如，内存在一个函数中分配，而在完全不同的函数中很久以后才释放）

第46.3节：重新分配内存

在分配内存后，您可能需要扩展或缩小指针的存储空间。函数void *realloc(void *ptr, size_t size)会释放ptr指向的旧对象，并返回一个指向大小为size的新对象的指针。ptr是之前用malloc、calloc或realloc（或空指针）分配的内存块的指针，需重新分配。原内存的最大可能内容会被保留。如果新大小更大，超出旧大小的额外内存未初始化。如果新大小更小，缩小部分的内容将丢失。如果ptr为NULL，函数会分配一个新块并返回其指针。

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p = malloc(10 * sizeof *p);
    if (NULL == p)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    p[0] = 42;
    p[9] = 15;

    /* 重新分配数组到更大的尺寸，将结果存储到一个
     * 临时指针中，以防 realloc() 失败。 */
    {
        int *temporary = realloc(p, 1000000 * sizeof *temporary);

        /* realloc() 失败，原始分配尚未释放。 */
        if (NULL == temporary)
        {
            perror("realloc() failed");
            free(p); /* 清理。 */
            return EXIT_FAILURE;
        }

        p = temporary;
    }

    /* 从这里开始，数组可以使用重新分配的新尺寸，
     * 直到它被释放。 */

    /* p[0] 到 p[9] 的值被保留，因此这将打印：
    42 15
    */
    printf("%d %d", p[0], p[9]); free(p);
}
```

match a pointer earlier returned by the `calloc`, `malloc`, or `realloc` function, or if the space has been deallocated by a call to `free` or `realloc`, the behavior is undefined.

- Sometimes the first approach cannot be used (e.g. memory is allocated in one function, and deallocated much later in a completely different function)

Section 46.3: Reallocating Memory

You may need to expand or shrink your pointer storage space after you have allocated memory to it. The void *realloc(void *ptr, size_t size) function deallocates the old object pointed to by ptr and returns a pointer to an object that has the size specified by size. ptr is the pointer to a memory block previously allocated with malloc, calloc or realloc (or a null pointer) to be reallocated. The maximal possible contents of the original memory is preserved. If the new size is larger, any additional memory beyond the old size are uninitialized. If the new size is shorter, the contents of the shrunken part is lost. If ptr is NULL, a new block is allocated and a pointer to it is returned by the function.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p = malloc(10 * sizeof *p);
    if (NULL == p)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    p[0] = 42;
    p[9] = 15;

    /* Reallocate array to a larger size, storing the result into a
     * temporary pointer in case realloc() fails. */
    {
        int *temporary = realloc(p, 1000000 * sizeof *temporary);

        /* realloc() failed, the original allocation was not free'd yet. */
        if (NULL == temporary)
        {
            perror("realloc() failed");
            free(p); /* Clean up. */
            return EXIT_FAILURE;
        }

        p = temporary;
    }

    /* From here on, array can be used with the new size it was
     * realloc'ed to, until it is free'd. */

    /* The values of p[0] to p[9] are preserved, so this will print:
    42 15
    */
    printf("%d %d\n", p[0], p[9]);
    free(p);
}
```

```
    return EXIT_SUCCESS;  
}
```

重新分配的对象地址可能与*p相同，也可能不同。因此，捕获realloc的返回值非常重要，返回值包含了调用成功时的新地址。

确保将realloc的返回值赋给一个临时变量，而不是原始的p。若调用失败，realloc会返回null，这会覆盖指针，导致数据丢失和内存泄漏。

第46.4节：realloc(ptr, 0) 不等同于 free(ptr)

realloc在概念上等同于对另一个指针执行malloc+memcpy+free操作。

如果请求的空间大小为零，realloc的行为由实现定义。所有接收大小参数为0的内存分配函数行为类似。这些函数可能返回非空指针，但绝不能解引用该指针。

因此，realloc(ptr,0)不等同于free(ptr)。它可能

- 是“懒惰”的实现，仅返回ptr
- 执行free(ptr)，分配一个虚拟元素并返回该元素
- 执行free(ptr)并返回0
- 失败时只返回0，不做其他操作。

因此，特别是后两种情况在应用代码中是无法区分的。

这意味着 realloc(ptr,0) 可能并不会真正释放/取消分配内存，因此它绝不应被用作 free 的替代。

```
    return EXIT_SUCCESS;  
}
```

The reallocated object may or may not have the same address as *p. Therefore it is important to capture the return value from `realloc` which contains the new address if the call is successful.

Make sure you assign the return value of `realloc` to a temporary instead of the original p. `realloc` will return null in case of any failure, which would overwrite the pointer. This would lose your data and create a memory leak.

Section 46.4: realloc(ptr, 0) is not equivalent to free(ptr)

`realloc` is conceptually equivalent to `malloc + memcpy + free` on the other pointer.

If the size of the space requested is zero, the behavior of `realloc` is implementation-defined. This is similar for all memory allocation functions that receive a size parameter of value 0. Such functions may in fact return a non-null pointer, but that must never be dereferenced.

Thus, `realloc(ptr, 0)` is not equivalent to `free(ptr)`. It may

- be a "lazy" implementation and just return ptr
- `free(ptr)`, allocate a dummy element and return that
- `free(ptr)` and return 0
- just return 0 for failure and do nothing else.

So in particular the latter two cases are indistinguishable by application code.

This means `realloc(ptr, 0)` may not really free/deallocate the memory, and thus it should never be used as a replacement for `free`.

第46.5节：可变大小的多维数组

版本 ≥ C99

自C99起，C语言支持可变长度数组（VLA），它们表示边界仅在初始化时已知的数组。虽然你必须小心不要分配过大的VLA（它们可能会破坏你的栈），但使用指向VLA的指针以及在 sizeof 表达式中使用它们是没问题的。

```
double sumAll(size_t n, size_t m, double A[n][m]) {  
    double ret = 0.0;  
    for (size_t i = 0; i < n; ++i)  
        for (size_t j = 0; j < m; ++j)  
            ret += A[i][j];  
    return ret;  
}  
  
int main(int argc, char *argv[argc+1]) {  
    size_t n = argc*10;  
    size_t m = argc*8;  
    double (*matrix)[m] = malloc(sizeof(double[n][m]));  
    // 以某种方式初始化矩阵  
    double res = sumAll(n, m, matrix);printf("result is %g", res);free(matrix);  
}
```

这里 matrix 是指向类型为 `double[m]` 元素的指针， sizeof 表达式与 `double[n][m]` 一起确保它包含 n 个这样的元素的空间。

```
double sumAll(size_t n, size_t m, double A[n][m]) {  
    double ret = 0.0;  
    for (size_t i = 0; i < n; ++i)  
        for (size_t j = 0; j < m; ++j)  
            ret += A[i][j];  
    return ret;  
}  
  
int main(int argc, char *argv[argc+1]) {  
    size_t n = argc*10;  
    size_t m = argc*8;  
    double (*matrix)[m] = malloc(sizeof(double[n][m]));  
    // initialize matrix somehow  
    double res = sumAll(n, m, matrix);  
    printf("result is %g\n", res);  
    free(matrix);  
}
```

Here matrix is a pointer to elements of type `double[m]`, and the sizeof expression with `double[n][m]` ensures that it contains space for n such elements.

所有这些空间都是连续分配的，因此可以通过一次调用 `free` 来释放。

语言中 VLA 的存在也影响了函数头中数组和指针的可能声明。现在，数组参数的 `[]` 中允许使用一般的整数表达式。对于这两个函数，`[]` 中的表达式使用了参数列表中之前声明的参数。对于 `sumAll`，这些是用户代码期望的矩阵长度。对于 C 中所有的数组函数参数，最内层维度都会被重写为指针类型，因此这等价于声明

```
double sumAll(size_t n, size_t m, double (*A)[m]);
```

也就是说，`n` 并不是真正函数接口的一部分，但该信息对文档有用，也可以被边界检查编译器用来警告越界访问。

同样，对于 `main`，表达式 `argc+1` 是 C 标准规定的 `argv` 参数的最小长度。

注意，官方上 C11 中 VLA 支持是可选的，但我们不知道哪个实现了 C11 的编译器不支持它。如果必须，可以用宏 `_STDC_NO_VLA_` 进行测试。

第 46.6 节：`alloca`：在栈上分配内存

注意：这里提到 `alloca` 仅为完整性考虑。它完全不可移植（不被任何常见标准覆盖），且具有许多潜在危险特性，对不了解的人来说不安全。现代 C 代码应使用 可变长度数组（VLA）来替代它。

[手册页](#)

```
#include <alloca.h>
// glibc 版本的 stdlib.h 默认包含 alloca.h

void foo(int size) {
    char *data = alloca(size);
    /*
    函数体;
    */
    // data 会自动释放
}
```

在调用者的栈帧上分配内存，返回指针所引用的空间会在调用函数结束时自动释放。

虽然此函数便于自动内存管理，但请注意请求大内存分配可能导致栈溢出，且不能对用 `alloca` 分配的内存使用 `free`（这可能导致更多的栈溢出问题）。

因此，不建议在循环或递归函数中使用 `alloca`。

并且由于内存函数返回时被释放，不能将该指针作为函数返回值返回（行为将是未定义的）。

总结

- 调用与 `malloc` 相同
- 函数返回时自动释放
- 与 `free`、`realloc` 函数不兼容（未定义行为）
- 指针不能作为函数结果返回（未定义行为）

All this space is allocated contiguously and can thus be deallocated by a single call to `free`.

The presence of VLA in the language also affects the possible declarations of arrays and pointers in function headers. Now, a general integer expression is permitted inside the `[]` of array parameters. For both functions the expressions in `[]` use parameters that have declared before in the parameter list. For `sumAll` these are the lengths that the user code expects for the matrix. As for all array function parameters in C the innermost dimension is rewritten to a pointer type, so this is equivalent to the declaration

```
double sumAll(size_t n, size_t m, double (*A)[m]);
```

也就是说，`n` 并不是真正函数接口的一部分，但该信息对文档有用，也可以被边界检查编译器用来警告越界访问。

Likewise, for `main`, the expression `argc+1` is the minimal length that the C standard prescribes for the `argv` argument.

Note that officially VLA support is optional in C11, but we know of no compiler that implements C11 and that doesn't have them. You could test with the macro `_STDC_NO_VLA_` if you must.

Section 46.6: `alloca`: allocate memory on stack

Caveat: `alloca` is only mentioned here for the sake of completeness. It is entirely non-portable (not covered by any of the common standards) and has a number of potentially dangerous features that make it un-safe for the unaware. Modern C code should replace it with *Variable Length Arrays* (VLA).

[Manual page](#)

```
#include <alloca.h>
// glibc 版本的 stdlib.h 包含 alloca.h 由 default

void foo(int size) {
    char *data = alloca(size);
    /*
    function body;
    */
    // data is automatically freed
}
```

Allocate memory on the stack frame of the caller, the space referenced by the returned pointer is automatically `free`'d when the caller function finishes.

While this function is convenient for automatic memory management, be aware that requesting large allocation could cause a stack overflow, and that you cannot use `free` with memory allocated with `alloca` (which could cause more issue with stack overflow).

For these reason it is not recommended to use `alloca` inside a loop nor a recursive function.

And because the memory is `free`'d upon function return you cannot return the pointer as a function result (the behavior would be undefined).

Summary

- call identical to `malloc`
- automatically free'd upon function return
- incompatible with `free`, `realloc` functions (undefined behavior)
- pointer cannot be returned as a function result (undefined behavior)

- 分配大小受限于栈空间，而栈空间（在大多数机器上）远小于malloc()可用的堆空间
- 避免在同一函数中使用alloca()和VLA（变长数组）
- alloca()的移植性不如malloc()等

建议

- 新代码中不要使用alloca()

版本 ≥ C99

现代替代方案。

```
void foo(int size) {
    char data[size];
    /*
    函数体;
    */
    // data会自动释放
}
```

这在alloca()可用的地方有效，并且在alloca()无效的地方也有效（例如循环内）。它假设使用的是C99实现，或者是未定义__STDC_NO_VLA__的C11实现。

第46.7节：用户定义的内存管理

malloc()通常调用底层操作系统函数来获取内存页。但该函数本身并无特殊之处，可以通过声明一个大型静态数组并从中分配来用纯C语言实现（确保正确对齐存在一定难度，实际上对齐到8字节几乎总是足够的）。

为了实现一个简单的方案，控制块存储在调用返回的指针之前的内存区域。这意味着free()可以通过从返回的指针中减去偏移量并读取控制信息来实现，控制信息通常包括块大小以及允许其放回空闲链表的信息——一个未分配块的链表。

当用户请求分配时，会搜索空闲链表，直到找到一个大小相同或更大的块，然后如有必要进行拆分。如果用户持续进行许多大小和时间间隔不可预测的分配和释放，这可能导致内存碎片（并非所有实际程序都如此，简单方案通常足够小型程序使用）。

```
/* 典型的控制块 */
struct block
{
    size_t size;          /* 块大小 */
    struct block *next;  /* 空闲链表中的下一个块 */
    struct block *prev;  /* 内存中前一个块的反向指针 */
    void *padding;       /* 需要16字节以保证是8的倍数 */
}

static struct block arena[10000]; /* 从这里分配 */
static struct block *firstfree;
```

许多程序需要大量分配大小相同的小对象。这非常容易实现。只需使用带有next指针的块即可。因此，如果需要一个32字节的块：

union block

- allocation size limited by stack space, which (on most machines) is a lot smaller than the heap space available for use by `malloc()`
- avoid using `alloca()` and VLAs (variable length arrays) in a single function
- `alloca()` is not as portable as `malloc()` et al

Recommendation

- Do not use `alloca()` in new code

Version ≥ C99

Modern alternative.

```
void foo(int size) {
    char data[size];
    /*
    function body;
    */
    // data is automatically freed
}
```

This works where `alloca()` does, and works in places where `alloca()` doesn't (inside loops, for example). It does assume either a C99 implementation or a C11 implementation that does not define __STDC_NO_VLA__.

Section 46.7: User-defined memory management

`malloc()` often calls underlying operating system functions to obtain pages of memory. But there is nothing special about the function and it can be implemented in straight C by declaring a large static array and allocating from it (there is a slight difficulty in ensuring correct alignment, in practice aligning to 8 bytes is almost always adequate).

To implement a simple scheme, a control block is stored in the region of memory immediately before the pointer to be returned from the call. This means that `free()` may be implemented by subtracting from the returned pointer and reading off the control information, which is typically the block size plus some information that allows it to be put back in the free list - a linked list of unallocated blocks.

When the user requests an allocation, the free list is searched until a block of identical or larger size to the amount requested is found, then if necessary it is split. This can lead to memory fragmentation if the user is continually making many allocations and frees of unpredictable size and at unpredictable intervals (not all real programs behave like that, the simple scheme is often adequate for small programs).

```
/* typical control block */
struct block
{
    size_t size;          /* size of block */
    struct block *next;  /* next block in free list */
    struct block *prev;  /* back pointer to previous block in memory */
    void *padding;       /* need 16 bytes to make multiple of 8 */
}

static struct block arena[10000]; /* allocate from here */
static struct block *firstfree;
```

Many programs require large numbers of allocations of small objects of the same size. This is very easy to implement. Simply use a block with a next pointer. So if a block of 32 bytes is required:

union block

```

{
    union block * next;
    unsigned char payload[32];
}

static union block arena[100];
static union block * head;
void init(void)
{
    int i;
    for (i = 0; i < 100 - 1; i++)
        arena[i].next = &arena[i + 1];
    arena[i].next = 0; /* Last one, null */
    head = &block[0];
}

void *block_alloc()
{
    void *answer = head;
    if (answer)
        head = head->next;
    return answer;
}

void block_free(void *ptr)
{
    union block *block = ptr;
    block->next = head;
    head = block;
}

```

该方案极其快速且高效，并且可以在一定程度上牺牲清晰度的情况下实现通用化。

```

{
    union block * next;
    unsigned char payload[32];
}

static union block arena[100];
static union block * head;
void init(void)
{
    int i;
    for (i = 0; i < 100 - 1; i++)
        arena[i].next = &arena[i + 1];
    arena[i].next = 0; /* Last one, null */
    head = &block[0];
}

void *block_alloc()
{
    void *answer = head;
    if (answer)
        head = head->next;
    return answer;
}

void block_free(void *ptr)
{
    union block *block = ptr;
    block->next = head;
    head = block;
}

```

This scheme is extremely fast and efficient, and can be made generic with a certain loss of clarity.

第47章：实现定义的行为

第47.1节：负整数的右移

```
int 有符号整数 = -1;  
  
// 右移操作表现出实现定义的行为：  
int 结果 = 有符号整数 >> 1;
```

第47.2节：给整数赋值超出范围的值

```
// 假设 SCHAR_MAX, 即有符号字符能表示的最大值,  
// 是127, 则该赋值的行为是实现定义的：  
有符号字符 整数;  
整数 = 128;
```

第47.3节：分配零字节

```
// 当请求分配的大小为零时, 分配函数具有实现定义的行为。  
void *p = malloc(0);
```

第47.4节：有符号整数的表示

每种有符号整数类型都可以用三种格式中的任意一种表示；具体使用哪一种由实现定义。对于任何给定的至少与int同宽的有符号整数类型，可以在运行时通过该类型中值-1的表示的最低两位来确定所使用的实现，方法如下：

```
enum { sign_magnitude = 1, ones_compl = 2, twos_compl = 3, };  
#define SIGN_REP(T) ((T)-1 & (T)3)  
  
switch (SIGN_REP(long)) {  
    case sign_magnitude: { /* 执行某些操作 */ break; }  
    case ones_compl: { /* 执行其他操作 */ break; }  
    case twos_compl: { /* 执行另外的操作 */ break; }  
    case 0: { _Static_assert(SIGN_REP(long), "错误的符号表示"); }  
}
```

同样的模式适用于较窄类型的表示，但不能用此技术进行测试，因为&的操作数在计算结果之前会受到“通常算术转换”的影响。

Chapter 47: Implementation-defined behaviour

Section 47.1: Right shift of a negative integer

```
int signed_integer = -1;  
  
// The right shift operation exhibits implementation-defined behavior:  
int result = signed_integer >> 1;
```

Section 47.2: Assigning an out-of-range value to an integer

```
// Supposing SCHAR_MAX, the maximum value that can be represented by a signed char, is  
// 127, the behavior of this assignment is implementation-defined:  
signed char integer;  
integer = 128;
```

Section 47.3: Allocating zero bytes

```
// The allocation functions have implementation-defined behavior when the requested size  
// of the allocation is zero.  
void *p = malloc(0);
```

Section 47.4: Representation of signed integers

Each signed integer type may be represented in any one of three formats; it is implementation-defined which one is used. The implementation in use for any given signed integer type at least as wide as int can be determined at runtime from the two lowest-order bits of the representation of value -1 in that type, like so:

```
enum { sign_magnitude = 1, ones_compl = 2, twos_compl = 3, };  
#define SIGN_REP(T) ((T)-1 & (T)3)  
  
switch (SIGN_REP(long)) {  
    case sign_magnitude: { /* do something */ break; }  
    case ones_compl: { /* do otherwise */ break; }  
    case twos_compl: { /* do yet else */ break; }  
    case 0: { _Static_assert(SIGN_REP(long), "bogus sign representation"); }  
}
```

The same pattern applies to the representation of narrower types, but they cannot be tested by this technique because the operands of & are subject to “the usual arithmetic conversions” before the result is computed.

第48章：原子操作

第48.1节：原子操作和运算符

原子变量可以在不同线程之间并发访问而不会产生竞态条件。

```
/* 一个所有线程都可见的全局静态变量 */
static unsigned _Atomic active = ATOMIC_VAR_INIT(0);

int myThread(void* a) {
    ++active;           // 增加活跃的无竞争状态
    // 执行某些操作
    --active;           // 减少活跃的无竞争状态
    return 0;
}
```

允许对基类型进行的所有左值操作（修改对象的操作）都是允许的，并且不会导致不同线程访问它们时发生竞争条件。

- 对原子对象的操作通常比普通算术操作慢几个数量级。
这也包括简单的加载或存储操作。因此，你应该仅在关键任务中使用它们。
- 通常的算术操作和赋值，如 `a = a+1;` 实际上是对 `a` 的三个操作：首先是加载，
然后是加法，最后是存储。这不是无竞争的。只有操作 `a += 1;` 和 `a++;` 是无竞争的。

Chapter 48: Atomics

Section 48.1: atomics and operators

Atomic variables can be accessed concurrently between different threads without creating race conditions.

```
/* a global static variable that is visible by all threads */
static unsigned _Atomic active = ATOMIC_VAR_INIT(0);
```

```
int myThread(void* a) {
    ++active;           // increment active race free
    // do something
    --active;           // decrement active race free
    return 0;
}
```

All lvalue operations (operations that modify the object) that are allowed for the base type are allowed and will not lead to race conditions between different threads that access them.

- Operations on atomic objects are generally orders of magnitude slower than normal arithmetic operations.
This also includes simple load or store operations. So you should only use them for critical tasks.
- Usual arithmetic operations and assignment such as `a = a+1;` are in fact three operations on `a`: first a load, then addition and finally a store. This is *not* race free. Only the operation `a += 1;` and `a++;` are.

第49章：跳转语句

第49.1节：使用return

返回一个值

一个常用的情况：从 main() 返回

```
#include <stdlib.h> /* 用于 EXIT_XXX 宏 */\n\nint main(int argc, char ** argv)\n{\n    if (2 < argc)\n    {\n        return EXIT_FAILURE; /* 代码期望一个参数：\n立即退出，跳过函数其余代码 */\n    }\n\n    /* 执行操作。 */\n\n    return EXIT_SUCCESS;\n}
```

附加说明：

- 对于返回类型为void的函数（不包括void *或相关类型），return语句不应带有任何表达式；即，唯一允许的return语句是return;。
- 对于返回类型非void的函数，return语句不得无表达式出现。
- 对于main()（且仅限于main()），在C99及以后版本中，return语句不是必需的。如果执行当达到终止}时，将隐式返回一个值0。有人认为省略这个return是不好的做法；也有人积极建议省略它。

不返回任何内容

从void函数返回

```
void log(const char * message_to_log)\n{\n    if (NULL == message_to_log)\n    {\n        return; /* 没有要记录的内容，立即返回，跳过日志记录。 */\n    }\n\n    fprintf(stderr, "%s:%d %s", __FILE__, __LINE__, message_to_log); return; /* 可选，\n因为此函数不返回值。 */\n}
```

第49.2节：使用goto跳出嵌套循环

跳出嵌套循环通常需要使用布尔变量，并在循环中检查该变量。假设我们正在遍历 i 和 j，代码可能如下所示

```
size_t i,j;\nfor (i = 0; i < myValue && !breakout_condition; ++i) {\n
```

Chapter 49: Jump Statements

Section 49.1: Using return

Returning a value

One commonly used case: returning from main()

```
#include <stdlib.h> /* for EXIT_XXX macros */\n\nint main(int argc, char ** argv)\n{\n    if (2 < argc)\n    {\n        return EXIT_FAILURE; /* The code expects one argument:\nleave immediately skipping the rest of the function's code */\n    }\n\n    /* Do stuff. */\n\n    return EXIT_SUCCESS;\n}
```

Additional notes:

- For a function having a return type as void (not including void * or related types), the return statement should not have any associated expression; i.e, the only allowed return statement would be return;.
- For a function having a non-void return type, the return statement shall not appear without an expression.
- For main() (and only for main()), an explicit return statement is not required (in C99 or later). If the execution reaches the terminating }, an implicit value of 0 is returned. Some people think omitting this return is bad practice; others actively suggest leaving it out.

Returning nothing

Returning from a void function

```
void log(const char * message_to_log)\n{\n    if (NULL == message_to_log)\n    {\n        return; /* Nothing to log, go home NOW, skip the logging. */\n    }\n\n    fprintf(stderr, "%s:%d %s\n", __FILE__, __LINE__, message_to_log);\n\n    return; /* Optional, as this function does not return a value. */\n}
```

Section 49.2: Using goto to jump out of nested loops

Jumping out of nested loops would usually require use of a boolean variable with a check for this variable in the loops. Supposing we are iterating over i and j, it could look like this

```
size_t i,j;\nfor (i = 0; i < myValue && !breakout_condition; ++i) {\n
```

```

for (j = 0; j < mySecondValue && !breakout_condition; ++j) {
    ... /* 做一些事情，可能会修改 breakout_condition */
    /* 当 breakout_condition == true 时，循环结束 */
}

```

但是 C 语言提供了 `goto` 语句，在这种情况下非常有用。通过将其与循环后声明的标签一起使用，我们可以轻松跳出循环。

```

size_t i,j;
for (i = 0; i < myValue; ++i) {
    for (j = 0; j < mySecondValue; ++j) {
        ...
        if(breakout_condition)
            goto final;
    }
}
final:

```

然而，通常在出现这种需求时，使用 `return` 可能更合适。该结构在结构化编程理论中也被认为是“非结构化”的。

另一个 `goto` 可能有用的情况是跳转到错误处理程序：

```

ptr = malloc(N * x);
if(!ptr)
    goto out_of_memory;

/* 正常处理 */
free(ptr);
return SUCCESS;

out_of_memory:
    free(ptr); /* 无害，且如果有进一步错误时是必要的 */
    return FAILURE;

```

使用 `goto` 保持错误流程与正常程序控制流程分离。然而，从技术角度来看，它也被认为是“非结构化”的。

第49.3节：使用 break 和 continue

在无效输入时立即 `continue` 读取，或在用户请求或文件结束时 `break`：

```

#include <stdlib.h> /* 用于 EXIT_xxx 宏 */
#include <stdio.h> /* 用于 printf() 和 getchar() */
#include <ctype.h> /* 用于 isdigit() */

void flush_input_stream(FILE * fp);

int main(void)
{
    int sum = 0;
    printf("请输入要相加的数字，或输入0退出：");do
    {
        int c = getchar();
        if (EOF == c)

```

```

for (j = 0; j < mySecondValue && !breakout_condition; ++j) {
    ... /* Do something, maybe modifying breakout_condition */
    /* When breakout_condition == true the loops end */
}
}

```

But the C language offers the `goto` clause, which can be useful in this case. By using it with a label declared after the loops, we can easily break out of the loops.

```

size_t i,j;
for (i = 0; i < myValue; ++i) {
    for (j = 0; j < mySecondValue; ++j) {
        ...
        if(breakout_condition)
            goto final;
    }
}
final:

```

However, often when this need comes up a `return` could be better used instead. This construct is also considered "unstructured" in structural programming theory.

Another situation where `goto` might be useful is for jumping to an error-handler:

```

ptr = malloc(N * x);
if(!ptr)
    goto out_of_memory;

/* normal processing */
free(ptr);
return SUCCESS;

out_of_memory:
    free(ptr); /* harmless, and necessary if we have further errors */
    return FAILURE;

```

Use of `goto` keeps error flow separate from normal program control flow. It is however also considered "unstructured" in the technical sense.

Section 49.3: Using break and continue

Immediately `continue` reading on invalid input or `break` on user request or end-of-file:

```

#include <stdlib.h> /* for EXIT_xxx macros */
#include <stdio.h> /* for printf() and getchar() */
#include <ctype.h> /* for isdigit() */

void flush_input_stream(FILE * fp);

int main(void)
{
    int sum = 0;
    printf("Enter digits to be summed up or 0 to exit:\n");

    do
    {
        int c = getchar();
        if (EOF == c)

```

```

{
    printf("读取到文件末尾，正在退出！");
    break;
}

if ('' != c){

flush_input_stream(stdin);
}

if (!isdigit(c))
{
    printf("%c 不是数字！重新开始！", c);continue;

}

if ('0' == c)
{
    printf("请求退出。");break;
}

sum += c - '0';

printf("当前的和是 %d。", sum);} while (1);

return EXIT_SUCCESS;
}

void flush_input_stream(FILE * fp)
{
    size_t i = 0;
    int c;
    while ((c = fgetc(fp)) != '' && c != EOF) /* 读取直到并包括下一个换行符。 */{
        ++i;
    }

    if (0 != i)
    {
        fprintf(stderr, "从输入中清除 %zu 个字符。", i);}
}
}

{
    printf("Read 'end-of-file', exiting!\n");

    break;
}

if ('\n' != c)
{
    flush_input_stream(stdin);
}

if (!isdigit(c))
{
    printf("%c is not a digit! Start over!\n", c);

    continue;
}

if ('0' == c)
{
    printf("Exit requested.\n");

    break;
}

sum += c - '0';

printf("The current sum is %d.\n");
} while (1);

return EXIT_SUCCESS;
}

void flush_input_stream(FILE * fp)
{
    size_t i = 0;
    int c;
    while ((c = fgetc(fp)) != '\n' && c != EOF) /* Pull all until and including the next new-line. */
    {
        ++i;
    }

    if (0 != i)
    {
        fprintf(stderr, "Flushed %zu characters from input.\n", i);
    }
}

```

第50章：创建和包含头文件

在现代C语言中，头文件是必须正确设计和使用的重要工具。它们允许编译器对程序中独立编译的部分进行交叉检查。

头文件声明类型、函数、宏等，这些是使用一组功能的代码所需要的。所有使用这些功能的代码都包含该头文件。所有定义这些功能的代码也包含该头文件。

这使得编译器能够检查使用和定义是否匹配。

第50.1节：简介

在C项目中创建和使用头文件时，需要遵循若干指导原则：

- 署等性

如果一个头文件在一个翻译单元（TU）中被多次包含，不应导致构建失败。

- 自包含性

如果你需要头文件中声明的功能，不应必须显式包含其他头文件。

- 最小性

你不应能从头文件中移除任何信息而不导致构建失败。

- 包含你所使用的（IWYU）

这对C++比C更为重要，但在C中同样重要。如果一个翻译单元中的代码（称为code.c）直接使用了某个头文件（称为"headerA.h"）声明的特性，那么code.c应直接#include "headerA.h"，即使该翻译单元包含了另一个头文件（称为"headerB.h"），而该头文件目前恰好包含了"headerA.h"。

偶尔，可能有充分的理由打破一条或多条这些指导原则，但你应当意识到自己正在打破规则，并且在打破之前了解这样做的后果。

第50.2节：自包含性

现代头文件应当是自包含的，这意味着需要使用其中定义的功能的程序header.h可以包含该头文件（#include "header.h"），而不必担心其他头文件是否需要先被包含。

建议：头文件应当是自包含的。

历史规则

历史上，这一直是一个略有争议的话题。

Chapter 50: Create and include header files

In modern C, header files are crucial tools that must be designed and used correctly. They allow the compiler to cross-check independently compiled parts of a program.

Headers declare types, functions, macros etc that are needed by the consumers of a set of facilities. All the code that uses any of those facilities includes the header. All the code that defines those facilities includes the header. This allows the compiler to check that the uses and definitions match.

Section 50.1: Introduction

There are a number of guidelines to follow when creating and using header files in a C project:

- Idemopotence

If a header file is included multiple times in a translation unit (TU), it should not break builds.

- Self-containment

If you need the facilities declared in a header file, you should not have to include any other headers explicitly.

- Minimality

You should not be able to remove any information from a header without causing builds to fail.

- Include What You Use (IWYU)

Of more concern to C++ than C, but nevertheless important in C too. If the code in a TU (call it code.c) directly uses the features declared by a header (call it "headerA.h"), then code.c should #include "headerA.h" directly, even if the TU includes another header (call it "headerB.h") that happens, at the moment, to include "headerA.h".

Occasionally, there might be good enough reasons to break one or more of these guidelines, but you should both be aware that you are breaking the rule and be aware of the consequences of doing so before you break it.

Section 50.2: Self-containment

Modern headers should be self-contained, which means that a program that needs to use the facilities defined by header.h can include that header (#include "header.h") and not worry about whether other headers need to be included first.

Recommendation: Header files should be self-contained.

Historical rules

Historically, this has been a mildly contentious subject.

在另一个千年以前，AT&T Indian Hill C 风格和编码标准 曾指出：

头文件不应嵌套。因此，头文件的前言应描述为了使该头文件功能完整，需要 `#include` 哪些其他头文件。在极端情况下，当许多头文件需要被包含在多个不同的源文件中时，将所有公共的 `#include` 语句放在一个包含文件中是可以接受的。

这与自包含的原则背道而驰。

现代规则

然而，从那时起，观点倾向于相反的方向。如果源文件需要使用由头文件header.h声明的功能，程序员应该能够写成：

```
#include "header.h"
```

并且（仅在命令行上设置了正确的搜索路径的前提下），任何必要的前置头文件将由 header.h 自动包含，无需在源文件中添加任何其他头文件。

这为源代码提供了更好的模块化。它还避免了代码经过十年或二十年的修改和篡改后，出现“猜测为什么添加了这个头文件”这一难题。

美国国家航空航天局戈达德太空飞行中心（NASA Goddard Space Flight Center, GSFC）的C语言编码标准是较为现代的标准之一—但现在有点难以找到。该标准规定头文件应当是自包含的。它还提供了一种简单的方法来确保头文件是自包含的：头文件的实现文件应将该头文件作为第一个包含的头文件。如果头文件不是自包含的，代码将无法编译。

GSFC给出的理由包括：

§2.1.1 头文件包含的理由

本标准要求单元头文件包含该单元头文件所需的所有其他头文件的`#include`语句。在单元体中首先放置单元头文件的`#include`语句，可以让编译器验证该头文件是否包含所有必需的`#include`语句。

另一种设计方案，本标准不允许，在头文件中不允许有`#include`语句；所有 `#includes` 在主体文件中完成。单元头文件必须包含 `#ifdef` 语句，以检查所需的头文件是否按正确顺序包含。

替代设计的一个优点是，主体文件中的`#include`列表正好是makefile中所需的依赖列表，并且该列表由编译器进行检查。使用标准设计时，必须使用工具来生成依赖列表。然而，所有主流推荐的开发环境都提供了这样的工具。

替代设计的一个主要缺点是，如果单元所需的头文件列表发生变化，使用该单元的每个文件都必须被编辑以更新`#include`语句列表。此外，所需的头文件列表对于一个

Once upon another millennium, the [AT&T Indian Hill C Style and Coding Standards](#) stated:

Header files should not be nested. The prologue for a header file should, therefore, describe what other headers need to be `#included` for the header to be functional. In extreme cases, where a large number of header files are to be included in several different source files, it is acceptable to put all common `#includes` in one include file.

This is the antithesis of self-containment.

Modern rules

However, since then, opinion has tended in the opposite direction. If a source file needs to use the facilities declared by a header header .h, the programmer should be able to write:

```
#include "header.h"
```

and (subject only to having the correct search paths set on the command line), any necessary pre-requisite headers will be included by header.h without needing any further headers added to the source file.

This provides better modularity for the source code. It also protects the source from the “guess why this header was added” conundrum that arises after the code has been modified and hacked for a decade or two.

The [NASA Goddard Space Flight Center \(GSFC\) coding standards for C](#) is one of the more modern standards — but is now a little hard to track down. It states that headers should be self-contained. It also provides a simple way to ensure that headers are self-contained: the implementation file for the header should include the header as the first header. If it is not self-contained, that code will not compile.

The rationale given by GSFC includes:

§2.1.1 Header include rationale

This standard requires a unit's header to contain `#include` statements for all other headers required by the unit header. Placing `#include` for the unit header first in the unit body allows the compiler to verify that the header contains all required `#include` statements.

An alternate design, not permitted by this standard, allows no `#include` statements in headers; all `#includes` are done in the body files. Unit header files then must contain `#ifdef` statements that check that the required headers are included in the proper order.

One advantage of the alternate design is that the `#include` list in the body file is exactly the dependency list needed in a makefile, and this list is checked by the compiler. With the standard design, a tool must be used to generate the dependency list. However, all of the branch recommended development environments provide such a tool.

A major disadvantage of the alternate design is that if a unit's required header list changes, each file that uses that unit must be edited to update the `#include` statement list. Also, the required header list for a

编译器库单元在不同目标上可能不同。

另一种设计的缺点是必须修改编译器库头文件和其他第三方文件，以添加所需的#define语句。

因此，自给自足意味着：

- 如果标题 header.h 需要一个新的嵌套标题 extra.h，您不必检查每个使用 header.h 的源文件来确定是否需要添加 extra.h。
- 如果一个头文件 header.h 不再需要包含特定的头文件 notneeded.h，您不必检查使用 header.h 的每个源文件来确定是否可以安全地移除 notneeded.h（但请参阅“包含你所使用的”原则）。
- 您不必确定包含前置头文件的正确顺序（这需要拓扑排序才能正确完成）。

检查自包含性

请参见链接静态库，了解一个可用于测试头文件幂等性和自包含性的脚本chkhdr。

第50.3节：极小性

头文件是一个关键的一致性检查机制，但它们应尽可能小。特别是，这意味着头文件不应仅因为实现文件需要其他头文件而包含那些头文件。头文件应只包含对所描述服务的使用者必要的头文件。

例如，项目头文件不应包含<stdio.h>，除非某个函数接口使用了类型 FILE*（或仅在<stdio.h>中定义的其他类型之一）。如果接口使用了size_t，最小的满足需求的头文件是<stddef.h>。显然，如果包含了另一个定义了size_t的头文件，就不需要再包含<stddef.h>。

如果头文件是最小化的，那么也能将编译时间保持在最低。

可以设计出专门用于包含许多其他头文件的头文件。从长远来看，这通常不是一个好主意，因为很少有源文件会真正需要所有这些头文件所描述的所有功能。

例如，可以设计一个<standard-c.h>，包含所有标准C头文件—但要小心，因为有些头文件并不总是存在。然而，实际上很少有程序会使用<locale.h>或<tgmath.h>的功能。

- 另见如何在C语言中链接多个实现文件？

第50.4节：符号和杂项

C标准指出，#include <header.h>和#include "header.h"这两种写法之间几乎没有区别。

[#include <header.h>] 在一系列实现定义的位置中搜索由 < 和 > 定界符之间指定的唯一序列标识的头文件，并将该指令替换为该头文件的全部内容。具体这些位置如何指定或头文件如何标识由实现定义。

compiler library unit may be different on different targets.

Another disadvantage of the alternate design is that compiler library header files, and other third party files, must be modified to add the required #ifdef statements.

Thus, self-containment means that:

- If a header header.h needs a new nested header extra.h, you do not have to check every source file that uses header.h to see whether you need to add extra.h.
- If a header header.h no longer needs to include a specific header notneeded.h, you do not have to check every source file that uses header.h to see whether you can safely remove notneeded.h (but see Include what you use).
- You do not have to establish the correct sequence for including the pre-requisite headers (which requires a topological sort to do the job properly).

Checking self-containment

See [Linking against a static library](#) for a script chkhdr that can be used to test idempotence and self-containment of a header file.

Section 50.3: Minimality

Headers are a crucial consistency checking mechanism, but they should be as small as possible. In particular, that means that a header should not include other headers just because the implementation file will need the other headers. A header should contain only those headers necessary for a consumer of the services described.

For example, a project header should not include <stdio.h> unless one of the function interfaces uses the type FILE* (or one of the other types defined solely in <stdio.h>). If an interface uses size_t, the smallest header that suffices is <stddef.h>. Obviously, if another header that defines size_t is included, there is no need to include <stddef.h> too.

If the headers are minimal, then it keeps the compilation time to a minimum too.

It is possible to devise headers whose sole purpose is to include a lot of other headers. These seldom turn out to be a good idea in the long run because few source files will actually need all the facilities described by all the headers. For example, a <standard-c.h> could be devised that includes all the standard C headers — with care since some headers are not always present. However, very few programs actually use the facilities of <locale.h> or <tgmath.h>.

- See also [How to link multiple implementation files in C?](#)

Section 50.4: Notation and Miscellany

The C standard says that there is very little difference between the #include <header.h> and #include "header.h" notations.

[#include <header.h>] searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

`[#include "header.h"]` 会被替换为由 `"..."` 定界符指定的源文件的全部内容。该指定的源文件会以实现定义的方式进行搜索。如果不支持此搜索，或者搜索失败，则该指令会被重新处理，仿佛它是 `[#include <header.h>]`。

`[#include "header.h"]` causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the `"..."` delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read `[#include <header.h>] ...`

因此，双引号形式可能会在比尖括号形式更多的位置查找。标准通过示例规定，标准头文件应使用尖括号包含，尽管使用双引号也能编译成功。同样，像 POSIX 这样的标准也使用尖括号格式——你也应该如此。双引号头文件应保留给项目定义的头文件。对于外部定义的头文件（包括项目依赖的其他项目的头文件），尖括号表示法更为合适。

注意，`#include` 和头文件之间应有一个空格，尽管编译器会接受没有空格的写法。空格成本很低。

许多项目使用如下的表示法：

```
#include <openssl/ssl.h>
#include <sys/stat.h>
#include <linux/kernel.h>
```

你应考虑是否在项目中使用这种命名空间控制（这很可能是个好主意）。应避免使用已有项目使用的名称（特别是，`sys` 和 `linux` 都是不好的选择）。

如果使用此方式，代码应在使用该表示法时保持谨慎和一致。

不要使用 `#include "../include/header.h"` 这种写法。

头文件很少（如果有的话）应定义变量。虽然你会尽量减少全局变量，但如果需要全局变量，应在头文件中声明，在一个合适的源文件中定义，该源文件会包含该头文件以交叉检查声明和定义，所有使用该变量的源文件都会使用该头文件进行声明。

推论：你不会在源文件中声明全局变量—源文件只包含定义。

头文件很少声明`static`函数，显著的例外是`static inline`函数，如果函数需要在多个源文件中使用，则会在头文件中定义。

- 源文件定义全局变量和全局函数。
- 源文件不声明全局变量或函数的存在；它们包含声明变量或函数的头文件。
- 头文件声明全局变量和函数（以及类型和其他支持内容）。
- 头文件不定义变量或任何函数，除了 (`static`) `inline` 函数。

So, the double quoted form may look in more places than the angle-bracketed form. The standard specifies by example that the standard headers should be included in angle-brackets, even though the compilation works if you use double quotes instead. Similarly, standards such as POSIX use the angle-bracketed format — and you should too. Reserve double-quoted headers for headers defined by the project. For externally-defined headers (including headers from other projects your project relies on), the angle-bracket notation is most appropriate.

Note that there should be a space between `#include` and the header, even though the compilers will accept no space there. Spaces are cheap.

A number of projects use a notation such as:

```
#include <openssl/ssl.h>
#include <sys/stat.h>
#include <linux/kernel.h>
```

You should consider whether to use that namespace control in your project (it is quite probably a good idea). You should steer clear of the names used by existing projects (in particular, both `sys` and `linux` would be bad choices).

If you use this, your code should be careful and consistent in the use of the notation.

Do not use `#include "../include/header.h"` notation.

Header files should seldom if ever define variables. Although you will keep global variables to a minimum, if you need a global variable, you will declare it in a header, and define it in one suitable source file, and that source file will include the header to cross-check the declaration and definition, and all source files that use the variable will use the header to declare it.

Corollary: you will not declare global variables in a source file — a source file will only contain definitions.

Header files should seldom declare `static` functions, with the notable exception of `static inline` functions which will be defined in headers if the function is needed in more than one source file.

- Source files define global variables, and global functions.
- Source files do not declare the existence of global variables or functions; they include the header that declares the variable or function.
- Header files declare global variable and functions (and types and other supporting material).
- Header files do not define variables or any functions except (`static`) `inline` functions.

Cross-references

- [Where to document functions in C?](#)
- [List of standard header files in C and C++](#)
- [Is `inline` without `static` or `extern` ever useful in C99?](#)
- [How do I use `extern` to share variables between source files?](#)
- [What are the benefits of a relative path such as `../include/header.h` for a header?](#)
- [Header inclusion optimization](#)
- [Should I include every header?](#)

第50.5节：幂等性

如果某个头文件在一个翻译单元 (TU) 中被包含多次，应该不会有任何编译问题。这称为“幂等性”；你的头文件应该是幂等的。想想如果你必须确保`#include <stdio.h>`只被包含一次，生活会有多困难。

实现幂等性有两种方法：头文件保护和`#pragma once`指令。

头文件保护

头文件保护简单可靠，符合C标准。头文件中的第一个非注释行应该是以下形式：

```
#ifndef UNIQUE_ID_FOR_HEADER  
#define UNIQUE_ID_FOR_HEADER
```

最后一个非注释行应该是`#endif`，后面可以带注释：

```
#endif /* UNIQUE_ID_FOR_HEADER */
```

所有操作代码，包括其他`#include`指令，都应位于这些行之间。

每个名称必须唯一。通常会使用诸如`HEADER_H_INCLUDED`的命名方案。一些较旧的代码使用定义为头文件保护符号的标识（例如`#ifndef BUFSIZ`在`<stdio.h>`中），但它不如唯一名称可靠。

一种选择是使用生成的MD5（或其他）哈希作为头文件保护符号名称。应避免模仿系统头文件使用的方案，这些方案通常使用保留的名称—以一个下划线开头，后跟另一个下划线或大写字母的名称。

`#pragma once`指令

另外，一些编译器支持`#pragma once`指令，其效果与头文件保护符号的三行代码相同。

```
#pragma once
```

支持`#pragma once`的编译器包括微软Visual Studio、GCC和Clang。然而，如果关注可移植性，最好使用头文件保护符号，或者两者同时使用。现代编译器（支持C89或更高版本）要求忽略未识别的`pragma`指令且不作评论（“任何未被实现识别的`pragma`都会被忽略”），但旧版本的GCC对此并不宽容。

第50.6节：包含你所使用的 (Include What You Use, IWWU)

谷歌的[Include What You Use](#)项目，简称IWWU，确保源文件包含代码中使用的所有头文件。

假设源文件`source.c`包含了头文件`arbitrary.h`，而该头文件恰好包含了`freeloader.h`，但源文件也显式且独立地使用了`freeloader.h`中的功能。一开始一切正常。

然后有一天，`arbitrary.h`发生了变化，其客户端不再需要`freeloader.h`的功能。突然，`source.c`无法编译—因为它不符合IWWU标准。由于`source.c`中的代码显式使用了`freeloader.h`的功能，它应该包含它所使用的内容—源文件中也应该有显式的`#include "freeloader.h"`。（幂等性会确保不会出现问题。）

IWWU（包括你所需要的）理念最大化了代码在对接口进行合理更改后仍能继续编译的可能性。显然，如果你的代码调用了随后从公开接口中移除的函数，

Section 50.5: Idempotence

If a particular header file is included more than once in a translation unit (TU), there should not be any compilation problems. This is termed 'idempotence'; your headers should be idempotent. Think how difficult life would be if you had to ensure that `#include <stdio.h>` was only included once.

There are two ways to achieve idempotence: header guards and the `#pragma once` directive.

Header guards

Header guards are simple and reliable and conform to the C standard. The first non-comment lines in a header file should be of the form:

```
#ifndef UNIQUE_ID_FOR_HEADER  
#define UNIQUE_ID_FOR_HEADER
```

The last non-comment line should be `#endif`, optionally with a comment after it:

```
#endif /* UNIQUE_ID_FOR_HEADER */
```

All the operational code, including other `#include` directives, should be between these lines.

Each name must be unique. Often, a name scheme such as `HEADER_H_INCLUDED` is used. Some older code uses a symbol defined as the header guard (e.g. `#ifndef BUFSIZ` in `<stdio.h>`), but it is not as reliable as a unique name.

One option would be to use a generated MD5 (or other) hash for the header guard name. You should avoid emulating the schemes used by system headers which frequently use names reserved to the implementation—names starting with an underscore followed by either another underscore or an upper-case letter.

The `#pragma once` Directive

Alternatively, some compilers support the `#pragma once` directive which has the same effect as the three lines shown for header guards.

```
#pragma once
```

The compilers which support `#pragma once` include MS Visual Studio and GCC and Clang. However, if portability is a concern, it is better to use header guards, or use both. Modern compilers (those supporting C89 or later) are required to ignore, without comment, pragmas that they do not recognize ('Any such pragma that is not recognized by the implementation is ignored') but old versions of GCC were not so indulgent.

Section 50.6: Include What You Use (IWWU)

Google's [Include What You Use](#) project, or IWWU, ensures source files include all headers used in the code.

Suppose a source file `source.c` includes a header `arbitrary.h` which in turn coincidentally includes `freeloader.h`, but the source file also explicitly and independently uses the facilities from `freeloader.h`. All is well to start with. Then one day `arbitrary.h` is changed so its clients no longer need the facilities of `freeloader.h`. Suddenly, `source.c` stops compiling—because it didn't meet the IWWU criteria. Because the code in `source.c` explicitly used the facilities of `freeloader.h`, it should have included what it uses—there should have been an explicit `#include "freeloader.h"` in the source too. (Idempotency would have ensured there wasn't a problem.)

The IWWU philosophy maximizes the probability that code continues to compile even with reasonable changes made to interfaces. Clearly, if your code calls a function that is subsequently removed from the published interface,

无论做多少准备，变更都是不可避免的。这就是为什么在可能的情况下会避免对API进行更改，以及为什么会有跨多个版本的弃用周期等原因。

这在C++中是一个特别的问题，因为标准头文件允许相互包含。源文件文件。cpp 可能包含一个头文件 header1.h，该头文件在某个平台上又包含另一个头文件 header2.h。file.cpp 可能最终也会使用 header2.h 的功能。最初这不会成为问题——代码可以编译，因为 header1.h 包含了 header2.h。在另一个平台上，或者当前平台升级后，header1.h 可能被修改，不再包含 header2.h，那么 file.cpp 就会因此无法编译。

IWYU会发现这个问题，并建议在 file.cpp 中直接包含 header2.h。这将确保代码继续能够编译。类似的考虑也适用于C代码。

no amount of preparation can prevent changes becoming necessary. This is why changes to APIs are avoided when possible, and why there are deprecation cycles over multiple releases, etc.

This is a particular problem in C++ because standard headers are allowed to include each other. Source file file.cpp could include one header header1.h that on one platform includes another header header2.h. file.cpp might turn out to use the facilities of header2.h as well. This wouldn't be a problem initially - the code would compile because header1.h includes header2.h. On another platform, or an upgrade of the current platform, header1.h could be revised so it no longer includes header2.h, and then file.cpp would stop compiling as a result.

IWYU would spot the problem and recommend that header2.h be included directly in file.cpp. This would ensure it continues to compile. Analogous considerations apply to C code too.

第51章：<ctype.h> — 字符分类与转换

第51.1节：介绍

头文件 ctype.h 是标准C库的一部分。它提供了用于分类和转换字符的函数。

所有这些函数都接受一个参数，该参数是一个 int 类型，必须是 EOF 或者可以表示为无符号字符的值。

分类函数的名称前缀为“is”。如果传入的字符满足相关条件，每个函数都会返回一个非零整数值 (TRUE)。如果条件不满足，函数则返回零值 (FALSE)。

这些分类函数的操作如下，假设使用默认的C语言环境 (locale)：

```
int a;
int c = 'A';
a = isalpha(c); /* 检查c是否为字母 (A-Z, a-z)，此处返回非零值。 */
a = isalnum(c); /* 检查c是否为字母或数字 (A-Z, a-z, 0-9)，此处返回非零值。 */
a = iscntrl(c); /* 检查c是否为控制字符 (0x00-0x1F, 0x7F)，此处返回零。 */
a = isdigit(c); /* 检查c是否为数字 (0-9)，此处返回零。 */
a = isgraph(c); /* 检查c是否有图形表示 (除空格外的任何可打印字符)，此处返回非零值。 */
a = islower(c); /* 检查c是否为小写字母 (a-z)，此处返回零。 */
a = isprint(c); /* 检查c是否为任何可打印字符 (包括空格)，此处返回非零值。
*/
a = isupper(c); /* 检查c是否为大写字母 (a-z)，此处返回零。 */
a = ispunct(c); /* 检查c是否为标点符号，此处返回零。 */
a = isspace(c); /* 检查c是否为空白字符，此处返回零。 */
a = isupper(c); /* 检查c是否为大写字母 (A-Z)，此处返回非零值。 */
a = isxdigit(c); /* 检查c是否为十六进制数字 (A-F, a-f, 0-9)，此处返回非零值。 */
版本 ≥ C99
a = isblank(c); /* 检查c是否为空白字符 (空格或制表符)，此处返回非零值。 */
```

有两个转换函数。这些函数以“to”为前缀命名。它们接受与上述相同的参数。但是返回值不是简单的零或非零，而是以某种方式改变后的传入参数。

这些转换函数的操作如下，假设使用默认的C语言环境 (locale)：

```
int a;
int c = 'A';

/* 将c转换为小写字母 (a-z)。
 * 如果无法转换，则返回未更改的值。
 * 返回这里的 'a'。
 */
a = tolower(c);

/* 将 c 转换为大写字母 (A-Z)。
 * 如果无法转换，则返回未更改的值。
 * 返回这里的 'A'。
 */
a = toupper(c);
```

Chapter 51: <ctype.h> – character classification & conversion

Section 51.1: Introduction

The header ctype.h is a part of the standard C library. It provides functions for classifying and converting characters.

All of these functions take one parameter, an int that must be either EOF or representable as an unsigned char.

The names of the classifying functions are prefixed with 'is'. Each returns an integer non-zero value (TRUE) if the character passed to it satisfies the related condition. If the condition is not satisfied then the function returns a zero value (FALSE).

These classifying functions operate as shown, assuming the default C locale:

```
int a;
int c = 'A';
a = isalpha(c); /* Checks if c is alphabetic (A-Z, a-z), returns non-zero here. */
a = isalnum(c); /* Checks if c is alphanumeric (A-Z, a-z, 0-9), returns non-zero here. */
a = iscntrl(c); /* Checks if c is a control character (0x00-0x1F, 0x7F), returns zero here. */
a = isdigit(c); /* Checks if c is a digit (0-9), returns zero here. */
a = isgraph(c); /* Checks if c has a graphical representation (any printing character except space), returns non-zero here. */
a = islower(c); /* Checks if c is a lower-case letter (a-z), returns zero here. */
a = isprint(c); /* Checks if c is any printable character (including space), returns non-zero here.
*/
a = isupper(c); /* Checks if c is an upper-case letter (a-z), returns zero here. */
a = ispunct(c); /* Checks if c is a punctuation character, returns zero here. */
a = isspace(c); /* Checks if c is a white-space character, returns zero here. */
a = isupper(c); /* Checks if c is an upper-case letter (A-Z), returns non-zero here. */
a = isxdigit(c); /* Checks if c is a hexadecimal digit (A-F, a-f, 0-9), returns non-zero here. */
Version ≥ C99
a = isblank(c); /* Checks if c is a blank character (space or tab), returns non-zero here. */
```

There are two conversion functions. These are named using the prefix 'to'. These functions take the same argument as those above. However the return value is not a simple zero or non-zero but the passed argument changed in some manner.

These conversion functions operate as shown, assuming the default C locale:

```
int a;
int c = 'A';

/* Converts c to a lower-case letter (a-z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'a' here.
 */
a = tolower(c);

/* Converts c to an upper-case letter (A-Z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'A' here.
 */
a = toupper(c);
```

以下信息引用自 cplusplus.com, 映射了原始的127字符ASCII集如何被各个分类类型函数识别 (a • 表示该函数对该字符返回非零值)

ASCII 值	字符	iscntrl	isblank	isspace	isupper	islower	isalpha	isdigit	isxdigit	isalnum	ispunct	isgraph	isprint
0x00 .. 0x08	NUL, (其他控制码) •												
0x09	制表符 ('')	•	•	•									
0x0A .. 0x0D	(空白控制 码: '\f', '\v', '\n', '\r')	•	•										
0x0E .. 0x1F	(其他控制码)	•											
0x20	空格 (' ')		•	•				•					
0x21 .. 0x2F	!"#\$%&'()*+,-./			•	•	•	•	•	•	•	•	•	•
0x30 .. 0x39	0123456789		•	•	•	•	•	•	•	•	•	•	•
0x3a .. 0x40	::<=>?@			•	•	•	•	•	•	•	•	•	•
0x41 .. 0x46	ABCDEF		•	•	•	•	•	•	•	•	•	•	•
0x47 .. 0x5A	GHIJKLMNOPQRSTUVWXYZ		•	•	•	•	•	•	•	•	•	•	•
0x5B .. 0x60	[]^_`			•	•	•	•	•	•	•	•	•	•
0x61 .. 0x66	abcdef		•	•	•	•	•	•	•	•	•	•	•
0x67 .. 0x7A	ghijklmnopqrstuvwxyz		•	•	•	•	•	•	•	•	•	•	•
0x7B .. 0x7E	{~-bar			•	•	•	•	•	•	•	•	•	•
0x7F	(DEL)	•											

第51.2节：从流中分类读取的字符

```
#include <ctype.h>
#include <stdio.h>

typedef struct {
    size_t space;
    size_t alnum;
    size_t punct;
} chartypes;

chartypes classify(FILE *f) {
    chartypes types = { 0, 0, 0 };
    int ch;

    while ((ch = fgetc(f)) != EOF) {
        types.space += !isspace(ch);
        types.alnum += !isalnum(ch);
        types.punct += !ispunct(ch);
    }

    return types;
}
```

classify函数从流中读取字符，并统计空格、字母数字和

The below information is quoted from cplusplus.com mapping how the original 127-character ASCII set is considered by each of the classifying type functions (a • indicates that the function returns non-zero for that character)

ASCII values	characters	iscntrl	isblank	isspace	isupper	islower	isalpha	isdigit	isxdigit	isalnum	ispunct	isgraph	isprint
0x00 .. 0x08	NUL, (other control codes) •												
0x09	tab ('\t')		•	•	•								
0x0A .. 0x0D	(white-space control codes: '\f', '\v', '\n', '\r')		•	•	•								
0x0E .. 0x1F	(other control codes)	•											
0x20	space (' ')			•									
0x21 .. 0x2F	!"#\$%&'()*+,-./			•	•	•	•	•	•	•	•	•	•
0x30 .. 0x39	0123456789			•	•	•	•	•	•	•	•	•	•
0x3a .. 0x40	::<=>?@			•	•	•	•	•	•	•	•	•	•
0x41 .. 0x46	ABCDEF				•			•	•	•	•	•	•
0x47 .. 0x5A	GHIJKLMNOPQRSTUVWXYZ					•	•	•	•	•	•	•	•
0x5B .. 0x60	[]^_`					•	•						
0x61 .. 0x66	abcdef					•	•	•	•	•	•	•	•
0x67 .. 0x7A	ghijklmnopqrstuvwxyz					•	•	•	•	•	•	•	•
0x7B .. 0x7E	{~-bar					•	•	•	•	•	•	•	•
0x7F	(DEL)	•											

Section 51.2: Classifying characters read from a stream

```
#include <ctype.h>
#include <stdio.h>

typedef struct {
    size_t space;
    size_t alnum;
    size_t punct;
} chartypes;

chartypes classify(FILE *f) {
    chartypes types = { 0, 0, 0 };
    int ch;

    while ((ch = fgetc(f)) != EOF) {
        types.space += !isspace(ch);
        types.alnum += !isalnum(ch);
        types.punct += !ispunct(ch);
    }

    return types;
}
```

The classify function reads characters from a stream and counts the number of spaces, alphanumeric and

标点符号字符的数量。它避免了若干陷阱。

- 从流中读取字符时，结果保存为int类型，否则会在读取EOF（文件结束标志）和具有相同位模式的字符之间产生歧义。
- 字符分类函数（例如`isspace`）期望其参数要么是可表示为`unsigned char`的值，要么是EOF宏的值。由于这正是`fgetc`返回的内容，因此这里无需进行转换。
- 字符分类函数的返回值仅区分零（表示false）和非零（表示true）。为了统计出现次数，需要将该值转换为1或0，这通过双重否定!!来实现。

第51.3节：从字符串中分类字符

```
#include <ctype.h>
#include <stddef.h>

typedef struct {
    size_t space;
    size_t alnum;
    size_t punct;
} chartypes;

chartypes classify(const char *s) {
    chartypes types = { 0, 0, 0 };
    const char *p;
    for (p = s; p != '\0'; p++) {
        types.space += !!isspace((unsigned char)*p);
        types.alnum += !!isalnum((unsigned char)*p);
        types.punct += !!ispunct((unsigned char)*p);
    }
    return types;
}
```

`classify`函数检查字符串中的所有字符，并统计空格、字母数字和标点符号字符的数量。它避免了若干陷阱。

- 字符分类函数（例如`isspace`）期望其参数要么是可表示为无符号字符（`unsigned char`）的值，要么是EOF宏的值。
- 表达式*p的类型是`char`，因此必须进行转换以符合上述要求。
- `char`类型被定义为等同于`signed char`或`unsigned char`中的一种。
- 当`char`等同于`unsigned char`时，没有问题，因为`char`类型的所有可能值都可以表示为`unsigned char`。
- 当`char`等同于有符号字符（`signed char`）时，必须先将其转换为无符号字符（`unsigned char`），然后再传递给字符分类函数。尽管由于这种转换字符的值可能会改变，但这正是这些函数所期望的。
- 字符分类函数的返回值仅区分零（表示false）和非零（表示true）。为了统计出现次数，需要将该值转换为1或0，这通过双重否定!!来实现。

punctuation characters. It avoids several pitfalls.

- When reading a character from a stream, the result is saved as an `int`, since otherwise there would be an ambiguity between reading EOF (the end-of-file marker) and a character that has the same bit pattern.
- The character classification functions (e.g. `isspace`) expect their argument to be *either representable as an unsigned char, or the value of the EOF macro*. Since this is exactly what the `fgetc` returns, there is no need for conversion here.
- The return value of the character classification functions only distinguishes between zero (meaning `false`) and nonzero (meaning `true`). For counting the number of occurrences, this value needs to be converted to a 1 or 0, which is done by the double negation, !!.

Section 51.3: Classifying characters from a string

```
#include <ctype.h>
#include <stddef.h>

typedef struct {
    size_t space;
    size_t alnum;
    size_t punct;
} chartypes;

chartypes classify(const char *s) {
    chartypes types = { 0, 0, 0 };
    const char *p;
    for (p = s; p != '\0'; p++) {
        types.space += !!isspace((unsigned char)*p);
        types.alnum += !!isalnum((unsigned char)*p);
        types.punct += !!ispunct((unsigned char)*p);
    }
    return types;
}
```

The `classify` function examines all characters from a string and counts the number of spaces, alphanumeric and punctuation characters. It avoids several pitfalls.

- The character classification functions (e.g. `isspace`) expect their argument to be *either representable as an unsigned char, or the value of the EOF macro*.
- The expression `*p` is of type `char` and must therefore be converted to match the above wording.
- The `char` type is defined to be equivalent to either `signed char` or `unsigned char`.
- When `char` is equivalent to `unsigned char`, there is no problem, since every possible value of the `char` type is representable as `unsigned char`.
- When `char` is equivalent to `signed char`, it must be converted to `unsigned char` before being passed to the character classification functions. And although the value of the character may change because of this conversion, this is exactly what these functions expect.
- The return value of the character classification functions only distinguishes between zero (meaning `false`) and nonzero (meaning `true`). For counting the number of occurrences, this value needs to be converted to a 1 or 0, which is done by the double negation, !!.

第52章：副作用

第52.1节：前置/后置自增/自减运算符

在C语言中，有两个一元运算符——'++'和'--'，它们是常见的混淆来源。运算符++称为自增运算符，运算符--称为自减运算符。它们都可以以前缀形式或后缀形式使用。前缀形式的++运算符语法是++操作数，后缀形式的语法是操作数++。使用前缀形式时，操作数先增加1，然后操作数的新值用于表达式的计算。考虑以下示例：

```
int n, x = 5;
n = ++x; /* x先自增1(x=6), 结果赋值给n(6) */
/* 这是两条语句的简写形式： */
/* x = x + 1; */
/* n = x; */
```

使用后缀形式时，表达式中先使用操作数当前的值，然后操作数的值增加1。考虑以下示例：

```
int n, x = 5;
n = x++; /* 先将x的值(5)赋给n(5), 然后x自增1；x变为(6) */
/* 这是两条语句的简写形式： */
/* n = x; */
/* x = x + 1; */
```

递减运算符--的工作原理也可以类似地理解。

以下代码演示了每个操作的作用

```
int main()
{
    int a, b, x = 42;
a = ++x; /* a 和 x 是 43 */
b = x++; /* b 是 43, x 是 44 */
a = x--; /* a 是 44, x 是 43 */
b = --x; /* b 和 x 是 42 */

    return 0;
}
```

由上可见，后置运算符返回变量的当前值，然后修改它，而前置运算符先修改变量，然后返回修改后的值。

在所有版本的 C 语言中，前置和后置运算符的求值顺序未定义，因此以下代码可能返回意外的结果：

```
int main()
{
    int a, x = 42;
a = x++ + x; /* 错误 */
a = x + x; /* 正确 */
++x;

    int ar[10];
x = 0;
ar[x] = x++; /* 错误 */
```

Chapter 52: Side Effects

Section 52.1: Pre/Post Increment/Decrement operators

In C, there are two unary operators - '++' and '--' that are very common source of confusion. The operator ++ is called the *increment operator* and the operator -- is called the *decrement operator*. Both of them can be used used in either *prefix form* or *postfix form*. The syntax for prefix form for ++ operator is ++operand and the syntax for postfix form is operand++. When used in the prefix form, the operand is incremented first by 1 and the resultant value of the operand is used in the evaluation of the expression. Consider the following example:

```
int n, x = 5;
n = ++x; /* x is incremented by 1(x=6), and result is assigned to n(6) */
/* this is a short form for two statements: */
/* x = x + 1; */
/* n = x; */
```

When used in the postfix form, the operand's current value is used in the expression and then the value of the operand is incremented by 1. Consider the following example:

```
int n, x = 5;
n = x++; /* value of x(5) is assigned first to n(5), and then x is incremented by 1; x(6) */
/* this is a short form for two statements: */
/* n = x; */
/* x = x + 1; */
```

The working of the decrement operator -- can be understood similarly.

The following code demonstrates what each one does

```
int main()
{
    int a, b, x = 42;
a = ++x; /* a and x are 43 */
b = x++; /* b is 43, x is 44 */
a = x--; /* a is 44, x is 43 */
b = --x; /* b and x are 42 */

    return 0;
}
```

From the above it is clear that post operators return the current value of a variable and *then* modify it, but pre operators modify the variable and *then* return the modified value.

In all versions of C, the order of evaluation of pre and post operators are not defined, hence the following code can return unexpected outputs:

```
int main()
{
    int a, x = 42;
a = x++ + x; /* wrong */
a = x + x; /* right */
++x;

    int ar[10];
x = 0;
ar[x] = x++; /* wrong */
```

```
ar[x++] = x; /* 错误 */
ar[x] = x; /* 正确 */
++x;
return 0;
}
```

请注意，当单独在语句中使用时，使用前置运算符优于后置运算符也是一种良好习惯。请参见上述代码。

还要注意，当调用函数时，所有对参数的副作用必须在函数运行之前完成。

```
int foo(int x)
{
    return x;
}

int main()
{
    int a = 42;
    int b = foo(a++); /* 这将返回43，尽管看起来应该返回42 */
    return 0;
}
```

```
ar[x++] = x; /* wrong */
ar[x] = x; /* right */
++x;
return 0;
}
```

Note that it is also good practice to use pre over post operators when used alone in a statement. Look at the above code for this.

Note also, that when a function is called, all side effects on arguments must take place before the function runs.

```
int foo(int x)
{
    return x;
}

int main()
{
    int a = 42;
    int b = foo(a++); /* This returns 43, even if it seems like it should return 42 */
    return 0;
}
```

第53章：多字符字符序列

第53.1节：三字符序列 (Trigraphs)

符号 [] { } ^ \ | ~ # 在 C 程序中经常使用，但在 1980 年代末，存在一些代码集（例如斯堪的纳维亚国家使用的 ISO 646 变体）中，这些符号的 ASCII 字符位置被用于国家语言变体字符（例如，英国用 £ 替代 #；丹麦用 Æ Å æ ø 代替 { } { } | \；EBCDIC 中没有 ~）。这导致在使用这些代码集的机器上编写 C 代码变得困难。

为了解决这个问题，C 标准建议使用三字符组合来生成一个称为三字符序列 (trigraph) 的单一字符。三字符序列是由三个字符组成的序列，其中前两个字符是问号。

以下是一个简单的例子，使用三字符序列代替#、{和}：

```
??=include <stdio.h>

int main()
??<
    printf("Hello World!");??>
```

这将被 C 预处理器通过将三字符序列替换为它们的单字符等价物来转换，就好像代码是这样写的：

```
#include <stdio.h>

int main()
{
    printf("Hello World!");}
```

三字符序列等价物

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

请注意，三字符序列存在问题，例如，??/ 是一个反斜杠，可能影响注释中续行的含义，并且必须在字符串和字符面量中识别（例如，'??/??/' 是一个单一字符，即反斜杠）。

第53.2节：有向图

版本 ≥ C99

1994年，五个三字符组合的更易读替代方案被提出。这些替代方案只使用两个字符，被称为双字符组合。与三字符组合不同，双字符组合是记号。如果双字符组合出现在其他记号中（例如字符串字面量或字符常量），则不会被视为双字符组合，而保持原样。

Chapter 53: Multi-Character Character Sequence

Section 53.1: Trigraphs

The symbols [] { } ^ \ | ~ # are frequently used in C programs, but in the late 1980s, there were code sets in use (ISO 646 variants, for example, in Scandinavian countries) where the ASCII character positions for these were used for national language variant characters (e.g. £ for # in the UK; Æ Å æ ø for { } { } | \ in Denmark; there was no ~ in EBCDIC). This meant that it was hard to write C code on machines that used these sets.

To solve this problem, the C standard suggested the use of combinations of three characters to produce a single character called a trigraph. A trigraph is a sequence of three characters, the first two of which are question marks.

The following is a simple example that uses trigraph sequences instead of #, { and }:

```
??=include <stdio.h>

int main()
??<
    printf("Hello World!\n");
??>
```

This will be changed by the C preprocessor by replacing the trigraphs with their single-character equivalents as if the code had been written:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
```

Trigraph Equivalent

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

Note that trigraphs are problematic because, for example, ??/ is a backslash and can affect the meaning of continuation lines in comments, and have to be recognized inside strings and character literals (e.g. '??/??/' is a single character, a backslash).

Section 53.2: Digraphs

Version ≥ C99

In 1994 more readable alternatives to five of the trigraphs were supplied. These use only two characters and are known as digraphs. Unlike trigraphs, digraphs are tokens. If a digraph occurs in another token (e.g. string literals or

如果双字符组合出现在其他记号中（例如字符串字面量或字符常量），则不会被视为双字符组合，而保持原样。

下面展示了处理双字符组合序列前后的区别。

```
#include <stdio.h>

int main()
<%
    printf("Hello %> World!"); /* 注意字符串中包含双字符组合 */%>
```

这将被视为与以下内容相同：

```
#include <stdio.h>

int main()
{
    printf("Hello %> World!"); /* 注意字符串中未改变的双字符组合。 */}
```

双字符组合等价物

<:	[
:>]
{	{
%>	}
#	#

character constants) then it will not be treated as a digraph, but remain as it is.

The following shows the difference before and after processing the digraphs sequence.

```
#include <stdio.h>

int main()
<%
    printf("Hello %> World!\n"); /* Note that the string contains a digraph */
%>
```

Which will be treated the same as:

```
#include <stdio.h>

int main()
{
    printf("Hello %> World!\n"); /* Note the unchanged digraph within the string. */
}
```

Digraph Equivalent

<:	[
:>]
<%	{
%>	}
%:	#

第54章：约束条件

第54.1节：同一作用域内重复的变量名

根据C标准表达的约束条件的一个例子是在同一作用域内声明两个同名变量，例如：

```
void foo(int bar)
{
    int var;
    double var;
}
```

这段代码违反了约束条件，必须在编译时产生诊断信息。这非常有用，因为相比未定义行为，开发者会在程序运行前被告知该问题，避免程序可能产生的任何异常行为。

因此，约束往往是那些在编译时容易检测到的错误，例如此类错误，而导致未定义行为但在编译时难以或不可能检测到的问题则不属于约束。

1) 准确措辞：

版本 = C99

如果标识符没有链接，则在相同作用域和相同命名空间中，该标识符（在声明符或类型说明符中）的声明不得超过一个，标签除外，如6.7.2.3节所规定。

第54.2节：一元算术运算符

一元+和-运算符仅可用于算术类型，因此例如如果尝试将它们用于结构体，程序将产生诊断信息，例如：

```
struct foo
{
    bool bar;
};

void baz(void)
{
    struct foo testStruct;
    -testStruct; /* 这违反了约束，因此必须产生诊断 */
}
```

Chapter 54: Constraints

Section 54.1: Duplicate variable names in the same scope

An example of a constraint as expressed in the C standard is having two variables of the same name declared in a scope), for example:

```
void foo(int bar)
{
    int var;
    double var;
}
```

This code breaches the constraint and must produce a diagnostic message at compile time. This is very useful as compared to undefined behavior as the developer will be informed of the issue before the program is run, potentially doing anything.

Constraints thus tend to be errors which are easily detectable at compile time such as this, issues which result in undefined behavior but would be difficult or impossible to detect at compile time are thus not constraints.

1) exact wording:

Version = C99

If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except for tags as specified in 6.7.2.3.

Section 54.2: Unary arithmetic operators

The unary + and - operators are only usable on arithmetic types, therefore if for example one tries to use them on a struct the program will produce a diagnostic eg:

```
struct foo
{
    bool bar;
};

void baz(void)
{
    struct foo testStruct;
    -testStruct; /* This breaks the constraint so must produce a diagnostic */
}
```

第55章：内联

第55.1节：内联在多个源文件中使用的函数

对于经常被调用的小函数，函数调用相关的开销可能占该函数总执行时间的很大一部分。因此，提高性能的一种方法是消除这部分开销。

在这个例子中，我们使用了四个函数（加上main()）分布在三个源文件中。其中两个函数（plusfive()和timestwo()）分别被位于“source1.c”和“source2.c”中的另外两个函数调用。包含main()是为了让我们有一个可运行的示例。

main.c :

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int main(void) {
    int start = 3;
    int intermediate = complicated1(start);printf("第一个结果是 %d", intermediate);
    intermediate = complicated2(start);printf("第二个结果是 %d", intermediate);
    return 0;
}
```

source1.c :

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated1(int input) {
    int tmp = timestwo(input);
    tmp = plusfive(tmp);
    return tmp;
}
```

source2.c :

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated2(int input) {
    int tmp = plusfive(input);
    tmp = timestwo(tmp);
    return tmp;
}
```

headerfile.h:

```
#ifndef HEADERFILE_H
#define HEADERFILE_H

int complicated1(int input);
int complicated2(int input);

inline int timestwo(int input) {
    return input * 2;
}
inline int plusfive(int input) {
```

Chapter 55: Inlining

Section 55.1: Inlining functions used in more than one source file

For small functions that get called often, the overhead associated with the function call can be a significant fraction of the total execution time of that function. One way of improving performance, then, is to eliminate the overhead.

In this example we use four functions (plus main()) in three source files. Two of those (plusfive() and timestwo()) each get called by the other two located in "source1.c" and "source2.c". The main() is included so we have a working example.

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int main(void) {
    int start = 3;
    int intermediate = complicated1(start);
    printf("First result is %d\n", intermediate);
    intermediate = complicated2(start);
    printf("Second result is %d\n", intermediate);
    return 0;
}
```

source1.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated1(int input) {
    int tmp = timestwo(input);
    tmp = plusfive(tmp);
    return tmp;
}
```

source2.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated2(int input) {
    int tmp = plusfive(input);
    tmp = timestwo(tmp);
    return tmp;
}
```

headerfile.h:

```
#ifndef HEADERFILE_H
#define HEADERFILE_H

int complicated1(int input);
int complicated2(int input);

inline int timestwo(int input) {
    return input * 2;
}
inline int plusfive(int input) {
```

```
return input + 5;  
}  
  
#endif
```

函数 `timestwo` 和 `plusfive` 被 `complicated1` 和 `complicated2` 两个不同的“翻译单元”或源文件调用。为了以这种方式使用它们，我们必须在头文件中定义它们。

假设使用 `gcc`，编译命令如下：

```
cc -O2 -std=c99 -c -o main.o main.c  
cc -O2 -std=c99 -c -o source1.o source1.c  
cc -O2 -std=c99 -c -o source2.o source2.c  
cc main.o source1.o source2.o -o main
```

我们使用 `-O2` 优化选项，因为某些编译器在未开启优化时不会进行内联。

`inline` 关键字的作用是该函数符号不会被输出到目标文件中。

否则，在最后一行链接目标文件生成最终可执行文件时会发生错误。如果没有 `inline`，两个 `.o` 文件中都会定义相同的符号，就会出现“符号重复定义”错误。

在实际需要该符号的情况下，这种做法的缺点是根本不会生成该符号。

有两种方法可以解决这个问题。第一种是在恰好一个.c文件中为内联函数添加额外的`extern`声明。因此，在`source1.c`中添加以下内容：

```
extern int timestwo(int input);  
extern int plusfive(int input);
```

另一种方法是用`static inline`来定义函数，而不是`inline`。这种方法的缺点是最终可能会在使用该头文件生成的每个目标文件中都产生该函数的副本。

```
return input + 5;  
}  
  
#endif
```

Functions `timestwo` and `plusfive` get called by both `complicated1` and `complicated2`, which are in different “translation units”, or source files. In order to use them in this way, we have to define them in the header.

Compile like this, assuming `gcc`:

```
cc -O2 -std=c99 -c -o main.o main.c  
cc -O2 -std=c99 -c -o source1.o source1.c  
cc -O2 -std=c99 -c -o source2.o source2.c  
cc main.o source1.o source2.o -o main
```

We use the `-O2` optimization option because some compilers don't inline without optimization turned on.

The effect of the `inline` keyword is that the function symbol in question is not emitted into the object file. Otherwise an error would occur in the last line, where we are linking the object files to form the final executable. If we would not have `inline`, the same symbol would be defined in both `.o` files, and a “multiply defined symbol” error would occur.

In situations where the symbol is actually needed, this has the disadvantage that the symbol is not produced at all. There are two possibilities to deal with that. The first is to add an extra `extern` declaration of the inlined functions in exactly one of the `.c` files. So add the following to `source1.c`:

```
extern int timestwo(int input);  
extern int plusfive(int input);
```

The other possibility is to define the function with `static inline` instead of `inline`. This method has the drawback that eventually a copy of the function in question may be produced in **every** object file that is produced with this header.

第56章：联合体

第56.1节：使用联合体重新解释值

一些C语言实现允许代码先写入联合体类型的一个成员，然后读取另一个成员，以执行某种重新解释转换（将新类型解析为旧类型的位表示）。

然而，重要的是要注意，这并不被当前或过去的C标准允许，并且会导致未定义行为，尽管如此，这仍是许多编译器提供的一个非常常见的扩展（如果你打算这样做，请查阅你的编译器文档）。

这种技术的一个现实例子是“快速逆平方根”算法，该算法依赖于IEEE 754浮点数的实现细节，比使用浮点运算更快地计算逆平方根。该算法可以通过指针转换（这非常危险且违反严格别名规则）或通过联合体（这仍是未定义行为，但在许多编译器中有效）来实现：

```
union floatToInt
{
    int32_t intMember;
    float floatMember; /* 浮点数必须是32位IEEE 754格式才能正常工作 */
};

float inverseSquareRoot(float input)
{
    union floatToInt x;
    int32_t i;
    float f;
    x.floatMember = input; /* 赋值给浮点成员 */
    i = x.intMember; /* 从整数成员读取 */
    i = 0x5f3759df - (i >> 1);
    x.intMember = i; /* 赋值给整数成员 */
    f = x.floatMember; /* 从浮点成员读取 */
    f = f * (1.5f - input * 0.5f * f * f);
    return f * (1.5f - input * 0.5f * f * f);
}
```

由于相比使用浮点运算速度更快，这种技术过去在计算机图形和游戏中被广泛使用，它是一种折衷方案，牺牲了一些精度且非常不具可移植性，以换取速度。

第56.2节：写入联合体的一个成员并从另一个成员读取

联合体的成员共享同一块内存空间。这意味着写入一个成员会覆盖所有其他成员的数据，从一个成员读取会得到与从所有其他成员读取相同的数据。然而，由于联合体成员可能具有不同的类型和大小，读取的数据可能会被不同地解释，详见

<http://stackoverflow.com/documentation/c/1119/structs-and-unions/9399/using-unions-to-reinterpret-values>

下面的简单示例演示了一个包含两个成员的联合体，两个成员类型相同。它显示写入成员m_1会导致从成员m_2读取到写入的值，写入成员m_2会导致从成员m_1读取到写入的值。

```
#include <stdio.h>
```

Chapter 56: Unions

Section 56.1: Using unions to reinterpret values

Some C implementations permit code to write to one member of a union type then read from another in order to perform a sort of reinterpreting cast (parsing the new type as the bit representation of the old one).

It is important to note however, this is not permitted by the C standard current or past and will result in undefined behavior, none the less is is a very common extension offered by compilers (so check your compiler docs if you plan to do this).

One real life example of this technique is the "Fast Inverse Square Root" algorithm which relies on implementation details of IEEE 754 floating point numbers to perform an inverse square root more quickly than using floating point operations, this algorithm can be performed either through pointer casting (which is very dangerous and breaks the strict aliasing rule) or through a union (which is still undefined behavior but works in many compilers):

```
union floatToInt
{
    int32_t intMember;
    float floatMember; /* Float must be 32 bits IEEE 754 for this to work */
};

float inverseSquareRoot(float input)
{
    union floatToInt x;
    int32_t i;
    float f;
    x.floatMember = input; /* Assign to the float member */
    i = x.intMember; /* Read back from the integer member */
    i = 0x5f3759df - (i >> 1);
    x.intMember = i; /* Assign to the integer member */
    f = x.floatMember; /* Read back from the float member */
    f = f * (1.5f - input * 0.5f * f * f);
    return f * (1.5f - input * 0.5f * f * f);
}
```

This technique was widely used in computer graphics and games in the past due to its greater speed compared to using floating point operations, and is very much a compromise, losing some accuracy and being very non portable in exchange for speed.

Section 56.2: Writing to one union member and reading from another

The members of a union share the same space in memory. This means that writing to one member overwrites the data in all other members and that reading from one member results in the same data as reading from all other members. However, because union members can have differing types and sizes, the data that is read can be interpreted differently, see

<http://stackoverflow.com/documentation/c/1119/structs-and-unions/9399/using-unions-to-reinterpret-values>

The simple example below demonstrates a union with two members, both of the same type. It shows that writing to member m_1 results in the written value being read from member m_2 and writing to member m_2 results in the written value being read from member m_1.

```
#include <stdio.h>
```

```

union my_union /* 定义联合体 */
{
    int m_1;
    int m_2;
};

int main (void)
{
    union my_union u;          /* 声明联合体 */
    u.m_1 = 1;                /* 写入 m_1 */printf("u.m_2: %i\n", u.m_2); /* 读取 m_2 */
    u.m_2 = 2;                /* 写入 m_2 */printf("u.m_1: %i\n", u.m_1); /* 读取 m_1 */return 0;
}

```

结果

```

u.m_2: 1
u.m_1: 2

```

第56.3节：结构体和联合体的区别

这说明联合体成员共享内存，而结构体成员不共享内存。

```

#include <stdio.h>
#include <string.h>

union My_Union
{
    int 变量_1;
    int 变量_2;
};

结构体 我的结构体
{
    int 变量_1;
    int 变量_2;
};

int main (void)
{
    联合体 我的联合体 u;
    结构体 我的结构体 s;
    u.变量_1 = 1;
    u.变量_2 = 2;
    s.变量_1 = 1;
    s.变量_2 = 2;
    printf ("u.变量_1: %i", u.变量_1);printf ("u.变量_2: %i",
    u.变量_2);printf ("s.变量_1: %i", s.变量_1);printf ("s.变量
    _2: %i", s.变量_2);printf ("sizeof (union My_Union): %i"
    , sizeof (union My_Union));printf ("sizeof (struct My_S
    truct): %i", sizeof (struct My_Struct));return 0;
}

```

```

union my_union /* Define union */
{
    int m_1;
    int m_2;
};

int main (void)
{
    union my_union u;          /* Declare union */
    u.m_1 = 1;                /* Write to m_1 */
    printf("u.m_2: %i\n", u.m_2); /* Read from m_2 */
    u.m_2 = 2;                /* Write to m_2 */
    printf("u.m_1: %i\n", u.m_1); /* Read from m_1 */
    return 0;
}

```

Result

```

u.m_2: 1
u.m_1: 2

```

Section 56.3: Difference between struct and union

This illustrates that union members shares memory and that struct members does not share memory.

```

#include <stdio.h>
#include <string.h>

union My_Union
{
    int variable_1;
    int variable_2;
};

struct My_Struct
{
    int variable_1;
    int variable_2;
};

int main (void)
{
    union My_Union u;
    struct My_Struct s;
    u.variable_1 = 1;
    u.variable_2 = 2;
    s.variable_1 = 1;
    s.variable_2 = 2;
    printf ("u.variable_1: %i\n", u.variable_1);
    printf ("u.variable_2: %i\n", u.variable_2);
    printf ("s.variable_1: %i\n", s.variable_1);
    printf ("s.variable_2: %i\n", s.variable_2);
    printf ("sizeof (union My_Union): %i\n", sizeof (union My_Union));
    printf ("sizeof (struct My_Struct): %i\n", sizeof (struct My_Struct));
    return 0;
}

```

第57章：线程（本地）

第57.1节：由单个线程初始化

在大多数情况下，所有被多个线程访问的数据应在创建线程之前初始化。这确保所有线程以清晰的状态开始，且不会发生“竞争条件”。

如果这不可行，可以使用once_flag和call_once

```
#include <threads.h>
#include <stdlib.h>

// 本示例的用户数据
double const* Big = 0;

// 保护Big的标志，必须是全局和/或静态的
static once_flag onceBig = ONCE_INIT;

void destroyBig(void) {
    free((void*)Big);
}

void initBig(void) {
    // 赋值给无const限定的临时变量
    double* b = malloc(largeNum);
    if (!b) {
        perror("allocation failed for Big");
        exit(EXIT_FAILURE);
    }
    // 现在初始化并存储Big
    initializeBigWithSophisticatedValues(largeNum, b);
    Big = b;
    // 确保在退出或快速退出时释放空间
    atexit(destroyBig);
    at_quick_exit(destroyBig);
}

// 依赖于 Big 的用户线程函数
int myThreadFunc(void* a) {
    call_once(&onceBig, initBig);
    // 从这里开始只使用 Big
    ...
    return 0;
}
```

once_flag 用于协调可能想要初始化相同数据 Big 的不同线程。对 call_once 的调用保证了

- initBig 只被调用一次
- call_once 会阻塞，直到 initBig 被调用，无论是由同一线程还是其他线程调用。

除了分配之外，这种只调用一次的函数中通常还会进行线程控制数据结构的动态初始化，比如 mtx_t 或 cnd_t，这些结构不能静态初始化，分别使用 mtx_init 或 cnd_init。

第57.2节：启动多个线程

```
#include <stdio.h>
```

Chapter 57: Threads (native)

Section 57.1: Initialization by one thread

In most cases all data that is accessed by several threads should be initialized before the threads are created. This ensures that all threads start with a clear state and no *race condition* occurs.

If this is not possible once_flag and call_once can be used

```
#include <threads.h>
#include <stdlib.h>

// the user data for this example
double const* Big = 0;

// the flag to protect big, must be global and/or static
static once_flag onceBig = ONCE_INIT;

void destroyBig(void) {
    free((void*)Big);
}

void initBig(void) {
    // assign to temporary with no const qualification
    double* b = malloc(largeNum);
    if (!b) {
        perror("allocation failed for Big");
        exit(EXIT_FAILURE);
    }
    // now initialize and store Big
    initializeBigWithSophisticatedValues(largeNum, b);
    Big = b;
    // ensure that the space is freed on exit or quick_exit
    atexit(destroyBig);
    at_quick_exit(destroyBig);
}

// the user thread function that relies on Big
int myThreadFunc(void* a) {
    call_once(&onceBig, initBig);
    // only use Big from here on
    ...
    return 0;
}
```

The once_flag is used to coordinate different threads that might want to initialize the same data Big. The call to call_once guarantees that

- initBig is called exactly once
- call_once blocks until such a call to initBig has been made, either by the same or another thread.

Besides allocation, a typical thing to do in such a once-called function is a dynamic initialization of a thread control data structures such as mtx_t or cnd_t that can't be initialized statically, using mtx_init or cnd_init, respectively.

Section 57.2: Start several threads

```
#include <stdio.h>
```

```

#include <threads.h>
#include <stdlib.h>

struct my_thread_data {
    double factor;
};

int my_thread_func(void* a) {
    struct my_thread_data* d = a;
    // 对 d 进行某些操作
    printf("we found %g", d->factor); // 返回成功或错误代码
    return d->factor > 1.0;
}

int main(int argc, char* argv[argc+1]) {
    unsigned n = 4;
    if (argc > 1) n = strtoull(argv[1], 0, 0);
    // 为线程的参数预留空间
    struct my_thread_data D[n]; // 不能初始化
    for (unsigned i = 0; i < n; ++i) {
        D[i] = (struct my_thread_data){ .factor = 0.5*i, };
    }
    // 为线程的 ID 预留空间
    thrd_t id[4];
    // 启动线程
    for (unsigned i = 0; i < n; ++i) {
        thrd_create(&id[i], my_thread_func, &D[i]);
    }
    // 等待所有线程完成，但忽略它们的返回值
    for (unsigned i = 0; i < n; ++i) {
        thrd_join(id[i], 0);
    }
    return EXIT_SUCCESS;
}

```

```

#include <threads.h>
#include <stdlib.h>

struct my_thread_data {
    double factor;
};

int my_thread_func(void* a) {
    struct my_thread_data* d = a;
    // do something with d
    printf("we found %g\n", d->factor);
    // return an success or error code
    return d->factor > 1.0;
}

int main(int argc, char* argv[argc+1]) {
    unsigned n = 4;
    if (argc > 1) n = strtoull(argv[1], 0, 0);
    // reserve space for the arguments for the threads
    struct my_thread_data D[n]; // can't be initialized
    for (unsigned i = 0; i < n; ++i) {
        D[i] = (struct my_thread_data){ .factor = 0.5*i, };
    }
    // reserve space for the ID's of the threads
    thrd_t id[4];
    // launch the threads
    for (unsigned i = 0; i < n; ++i) {
        thrd_create(&id[i], my_thread_func, &D[i]);
    }
    // Wait that all threads have finished, but throw away their
    // return values
    for (unsigned i = 0; i < n; ++i) {
        thrd_join(id[i], 0);
    }
    return EXIT_SUCCESS;
}

```

第58章：多线程

C11 标准中有一个线程库, <threads.h>, 但目前还没有已知的编译器实现它。因此, 要在 C 语言中使用多线程, 必须使用平台特定的实现, 例如使用 pthread.h 头文件的 POSIX 线程库 (通常称为 pthreads)。

第58.1节：C11 线程简单示例

```
#include <threads.h>
#include <stdio.h>

int run(void *arg)
{
    printf("Hello world of C11 threads.");
    return 0;
}

int main(int argc, const char *argv[])
{
    thrd_t 线程;
    int result;

    thrd_create(&线程, run, NULL);

    thrd_join(&线程, &result);

    printf("线程返回 %d 结束", result);
}
```

Chapter 58: Multithreading

In C11 there is a standard thread library, <threads.h>, but no known compiler that yet implements it. Thus, to use multithreading in C you must use platform specific implementations such as the POSIX threads library (often referred to as pthreads) using the pthread.h header.

Section 58.1: C11 Threads simple example

```
#include <threads.h>
#include <stdio.h>

int run(void *arg)
{
    printf("Hello world of C11 threads.");
    return 0;
}

int main(int argc, const char *argv[])
{
    thrd_t thread;
    int result;

    thrd_create(&thread, run, NULL);

    thrd_join(&thread, &result);

    printf("Thread return %d at the end\n", result);
}
```

第59章：进程间通信(IPC)

进程间通信(IPC)机制允许不同的独立进程相互通信。标准C语言不提供任何IPC机制。因此，所有此类机制均由宿主操作系统定义。POSIX定义了一套广泛的IPC机制；Windows定义了另一套；其他系统则定义了它们自己的变体。

第59.1节：信号量

信号量用于同步两个或多个进程之间的操作。POSIX定义了两套不同的信号量函数：

1. "System V IPC" — [semctl\(\)](#), [semop\(\)](#), [semget\(\)](#).
2. "POSIX信号量" — [sem_close\(\)](#), [sem_destroy\(\)](#), [sem_getvalue\(\)](#), [sem_init\(\)](#), [sem_open\(\)](#), [sem_post\(\)](#),
[sem_trywait\(\)](#), [sem_unlink\(\)](#).

本节描述的是System V IPC信号量，之所以称为System V，是因为它们起源于Unix System V。

首先，你需要包含所需的头文件。旧版本的POSIX要求#include <sys/types.h>；现代POSIX和大多数系统则不需要。

```
#include <sys/sem.h>
```

然后，你需要在父进程和子进程中都定义一个键。

```
#define KEY 0x1111
```

这个键在两个程序中必须相同，否则它们将不会引用相同的IPC结构。有一些方法可以生成一个约定的键，而无需硬编码其值。

接下来，根据您的编译器，您可能需要也可能不需要执行此步骤：声明一个用于信号量操作的联合体。

```
union semun {  
    int val;  
    struct semid_ds *buf;  
    unsigned short *array;  
};
```

接下来，定义您的try (semwait) 和raise (semsignal) 结构体。名称P和V来源于荷兰语

```
struct sembuf p = { 0, -1, SEM_UNDO}; # semwait  
struct sembuf v = { 0, +1, SEM_UNDO}; # semsignal
```

现在，首先获取您的IPC信号量的ID。

```
int id;  
// 第二个参数是信号量的数量  
// 第三个参数是模式 (IPC_CREAT在需要时创建信号量集)  
if ((id = semget(KEY, 1, 0666 | IPC_CREAT) < 0) {  
    /* 错误处理代码 */  
}
```

Chapter 59: Interprocess Communication (IPC)

Inter-process communication (IPC) mechanisms allow different independent processes to communicate with each other. Standard C does not provide any IPC mechanisms. Therefore, all such mechanisms are defined by the host operating system. POSIX defines an extensive set of IPC mechanisms; Windows defines another set; and other systems define their own variants.

Section 59.1: Semaphores

Semaphores are used to synchronize operations between two or more processes. POSIX defines two different sets of semaphore functions:

1. 'System V IPC' — [semctl\(\)](#), [semop\(\)](#), [semget\(\)](#).
2. 'POSIX Semaphores' — [sem_close\(\)](#), [sem_destroy\(\)](#), [sem_getvalue\(\)](#), [sem_init\(\)](#), [sem_open\(\)](#), [sem_post\(\)](#),
[sem_trywait\(\)](#), [sem_unlink\(\)](#).

This section describes the System V IPC semaphores, so called because they originated with Unix System V.

First, you'll need to include the required headers. Old versions of POSIX required #include <sys/types.h>; modern POSIX and most systems do not require it.

```
#include <sys/sem.h>
```

Then, you'll need to define a key in both the parent as well as the child.

```
#define KEY 0x1111
```

This key needs to be the same in both programs or they will not refer to the same IPC structure. There are ways to generate an agreed key without hard-coding its value.

Next, depending on your compiler, you may or may not need to do this step: declare a union for the purpose of semaphore operations.

```
union semun {  
    int val;  
    struct semid_ds *buf;  
    unsigned short *array;  
};
```

Next, define your try (semwait) and raise (semsignal) structures. The names P and V originate from Dutch

```
struct sembuf p = { 0, -1, SEM_UNDO}; # semwait  
struct sembuf v = { 0, +1, SEM_UNDO}; # semsignal
```

Now, start by getting the id for your IPC semaphore.

```
int id;  
// 2nd argument is number of semaphores  
// 3rd argument is the mode (IPC_CREAT creates the semaphore set if needed)  
if ((id = semget(KEY, 1, 0666 | IPC_CREAT) < 0) {  
    /* error handling code */  
}
```

在父进程中，将信号量初始化为计数器为1。

```
union semun u;
u.val = 1;
if (semctl(id, 0, SETVAL, u) < 0) { // SETVAL 是一个宏，用于指定将信号量的值设置为联合体 u 指定的值
    /* 错误处理代码 */
}
```

现在，你可以根据需要递减或递增信号量。在临界区开始时，使用 `semop()` 函数递减计数器：

```
if (semop(id, &p, 1) < 0) {
    /* 错误处理代码 */
}
```

要递增信号量，使用 `&v` 替代 `&p`：

```
if (semop(id, &v, 1) < 0) {
    /* 错误处理代码 */
}
```

注意，每个函数成功时返回 0，失败时返回 -1。不检查这些返回状态可能会导致严重的问题。

示例 1.1：线程竞速

下面的程序将会创建一个子进程，父进程和子进程都会尝试在终端上打印字符，但没有任何同步机制。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main()
{
    int pid;
    pid = fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork");
        exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefghijklmnopqrstuvwxyz";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
}
```

In the parent, initialise the semaphore to have a counter of 1.

```
union semun u;
u.val = 1;
if (semctl(id, 0, SETVAL, u) < 0) { // SETVAL is a macro to specify that you're setting the value of
    /* error handling code */
}
```

Now, you can decrement or increment the semaphore as you need. At the start of your critical section, you decrement the counter using the `semop()` function:

```
if (semop(id, &p, 1) < 0) {
    /* error handling code */
}
```

To increment the semaphore, you use `&v` instead of `&p`:

```
if (semop(id, &v, 1) < 0) {
    /* error handling code */
}
```

Note that every function returns 0 on success and -1 on failure. Not checking these return statuses can cause devastating problems.

Example 1.1: Racing with Threads

The below program will have a process fork a child and both parent and child attempt to print characters onto the terminal without any synchronization.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main()
{
    int pid;
    pid = fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork");
        exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefghijklmnopqrstuvwxyz";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
}
```

```

else
{
    char *s = "ABCDEFGH";
    int l = strlen(s);
    for(int i = 0; i < l; ++i)
    {
        putchar(s[i]);
        fflush(stdout);
        sleep(rand() % 2);
        putchar(s[i]);
        fflush(stdout);
        sleep(rand() % 2);
    }
}

```

输出 (第一次运行) :

```
aABAaBCbCbDDcEEcddeFFGGHHefffghh
```

(第二次运行) :

```
aabbccAABddBCeeCffgDDghEEhFFGGHH
```

编译并运行此程序，每次应得到不同的输出。

示例 1.2：使用信号量避免竞态条件

将示例 1.1 修改为使用信号量，我们得到：

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY 0x1111

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

struct sembuf p = { 0, -1, SEM_UNDO };
struct sembuf v = { 0, +1, SEM_UNDO };

int main()
{
    int id = semget(KEY, 1, 0666 | IPC_CREAT);
    if(id < 0)
    {
        perror("semget");
        exit(11);
    }
    union semun u;
    u.val = 1;
    if(semctl(id, 0, SETVAL, u) < 0)
    {

```

```

else
{
    char *s = "ABCDEFGH";
    int l = strlen(s);
    for(int i = 0; i < l; ++i)
    {
        putchar(s[i]);
        fflush(stdout);
        sleep(rand() % 2);
        putchar(s[i]);
        fflush(stdout);
        sleep(rand() % 2);
    }
}

```

Output (1st run):

```
aABAaBCbCbDDcEEcddeFFGGHHefffghh
```

(2nd run):

```
aabbccAABddBCeeCffgDDghEEhFFGGHH
```

Compiling and running this program should give you a different output each time .

Example 1.2: Avoid Racing with Semaphores

Modifying Example 1.1 to use semaphores, we have:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY 0x1111

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

struct sembuf p = { 0, -1, SEM_UNDO };
struct sembuf v = { 0, +1, SEM_UNDO };

int main()
{
    int id = semget(KEY, 1, 0666 | IPC_CREAT);
    if(id < 0)
    {
        perror("semget");
        exit(11);
    }
    union semun u;
    u.val = 1;
    if(semctl(id, 0, SETVAL, u) < 0)
    {

```

```

        perror("semctl"); exit(12);
    }
    int pid;
    pid = fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefg";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(13);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(14);
            }
        }

        sleep(rand() % 2);
    }
    else
    {
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(15);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(16);
            }
        }

        sleep(rand() % 2);
    }
}

```

输出：

aabbAABBCCccdeeDDffEEFFGGHHgggh

```

        perror("semctl"); exit(12);
    }
    int pid;
    pid = fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefg";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(13);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(14);
            }
        }

        sleep(rand() % 2);
    }
    else
    {
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(15);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(16);
            }
        }

        sleep(rand() % 2);
    }
}

```

Output:

aabbAABBCCccdeeDDffEEFFGGHHgggh

编译并运行此程序每次都会得到相同的输出。

Compiling and running this program will give you the same output each time.

第60章：测试框架

许多开发者使用单元测试来检查他们的软件是否按预期工作。单元测试检查较大软件中的小单元，并确保输出符合预期。测试框架通过提供初始化/清理服务和协调测试，使单元测试更容易进行。

有许多适用于C语言的单元测试框架。例如，Unity是一个纯C语言框架。人们也经常使用C++测试框架来测试C代码；同样也有许多C++测试框架。

第60.1节：Unity测试框架

Unity是一个xUnit风格的C语言单元测试框架。它完全用C语言编写，具有可移植、快速、简单、表达力强且可扩展的特点。它特别设计为对嵌入式系统的单元测试也非常有用。

一个简单的测试用例，用于检查函数的返回值，可能如下所示

```
void test_FunctionUnderTest_should_ReturnFive(void)
{
    TEST_ASSERT_EQUAL_INT( 5, FunctionUnderTest() );
}
```

一个完整的测试文件可能如下所示：

```
#include "unity.h"
#include "UnitUnderTest.h" /* 待测试单元。 */

void setUp (void) /* 在每个测试前运行，将单元初始化调用放在这里。 */
void tearDown (void) /* 在每个测试后运行，将单元清理调用放在这里。 */

void test_TheFirst(void)
{
TEST_IGNORE_MESSAGE("Hello world!"); /* 忽略此测试但打印一条消息。 */
}

int main (void)
{
UNITY_BEGIN();
RUN_TEST(test_TheFirst); /* 运行测试。 */
    return UNITY_END();
}
```

Unity 附带了一些示例项目、makefile 和一些 Ruby rake 脚本，帮助更轻松地创建较长的测试文件。

第60.2节：CMocka

CMocka 是一个优雅的 C 语言单元测试框架，支持模拟对象。它仅依赖标准 C 库，适用于多种计算平台（包括嵌入式）和不同编译器。它提供了模拟测试教程、API 文档和各种示例。

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

void null_test_success (void ** state) {}
```

Chapter 60: Testing frameworks

Many developers use unit tests to check that their software works as expected. Unit tests check small units of larger pieces of software, and ensure that the outputs match expectations. Testing frameworks make unit testing easier by providing set-up/tear-down services and coordinating the tests.

There are many unit testing frameworks available for C. For example, Unity is a pure C framework. People quite often use C++ testing frameworks to test C code; there are many C++ test frameworks too.

Section 60.1: Unity Test Framework

Unity is an xUnit-style test framework for unit testing C. It is written completely in C and is portable, quick, simple, expressive and extensible. It is designed to especially be also useful for unit testing for embedded systems.

A simple test case that checks the return value of a function, might look as follows

```
void test_FunctionUnderTest_should_ReturnFive(void)
{
    TEST_ASSERT_EQUAL_INT( 5, FunctionUnderTest() );
}
```

A full test file might look like:

```
#include "unity.h"
#include "UnitUnderTest.h" /* The unit to be tested. */

void setUp (void) /* Is run before every test, put unit init calls here. */
void tearDown (void) /* Is run after every test, put unit clean-up calls here. */

void test_TheFirst(void)
{
    TEST_IGNORE_MESSAGE("Hello world!"); /* Ignore this test but print a message. */
}

int main (void)
{
    UNITY_BEGIN();
    RUN_TEST(test_TheFirst); /* Run the test. */
    return UNITY_END();
}
```

Unity comes with some example projects, makefiles and some Ruby rake scripts that help make creating longer test files a bit easier.

Section 60.2: CMocka

CMocka is an elegant unit testing framework for C with support for mock objects. It only requires the standard C library, works on a range of computing platforms (including embedded) and with different compilers. It has a [tutorial](#) on testing with mocks, [API documentation](#), and a variety of [examples](#).

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

void null_test_success (void ** state) {}
```

```

void null_test_fail (void ** state)
{
    assert_true (0);
}

/* 这些函数将用于初始化
并在每次测试运行后清理资源 */
int setup (void ** state)
{
    return 0;
}

int teardown (void ** state)
{
    return 0;
}

int main (void)
{
    const struct CMUnitTests tests [] =
    {
cmocka_unit_test (null_test_success),
        cmocka_unit_test (null_test_fail),
    };

    /* 如果不需要 setup 和 teardown 函数,
则可以传入 NULL */

    int count_fail_tests =
cmocka_run_group_tests (tests, setup, teardown);

    return count_fail_tests;
}

```

第60.3节：CppUTest

[CppUTest](#) 是一个 [xUnit](#) 风格的C和C++单元测试框架。它用C++编写，旨在实现可移植性和设计上的简洁。它支持内存泄漏检测、构建模拟对象，并能与 Google Test 一起运行测试。附带用于Visual Studio和Eclipse CDT的辅助脚本和示例项目。

```

#include <CppUTest/CommandLineTestRunner.h>
#include <CppUTest/TestHarness.h>

TEST_GROUP(Foo_Group) {}

TEST(Foo_Group, Foo_TestOne) {}

/* 测试运行器可以提供选项,
例如启用彩色输出, 运行特定的
测试或测试组等。
该函数将返回失败测试的数量。 */

int main(int argc, char ** argv)
{
    RUN_ALL_TESTS(argc, argv);
}

```

一个测试组可以有一个 `setup()` 和一个 `teardown()` 方法。`setup` 方法在每个测试之前调用，且

```

void null_test_fail (void ** state)
{
    assert_true (0);
}

/* These functions will be used to initialize
and clean resources up after each test run */
int setup (void ** state)
{
    return 0;
}

int teardown (void ** state)
{
    return 0;
}

int main (void)
{
    const struct CMUnitTests tests [] =
    {
        cmocka_unit_test (null_test_success),
        cmocka_unit_test (null_test_fail),
    };

    /* If setup and teardown functions are not
needed, then NULL may be passed instead */

    int count_fail_tests =
        cmocka_run_group_tests (tests, setup, teardown);

    return count_fail_tests;
}

```

Section 60.3: CppUTest

[CppUTest](#) is an [xUnit](#)-style framework for unit testing C and C++. It is written in C++ and aims for portability and simplicity in design. It has support for memory leak detection, building mocks, and running its tests along with the Google Test. Comes with helper scripts and sample projects for Visual Studio and Eclipse CDT.

```

#include <CppUTest/CommandLineTestRunner.h>
#include <CppUTest/TestHarness.h>

TEST_GROUP(Foo_Group) {}

TEST(Foo_Group, Foo_TestOne) {}

/* Test runner may be provided options, such
as to enable colored output, to run only a
specific test or a group of tests, etc. This
will return the number of failed tests. */

int main(int argc, char ** argv)
{
    RUN_ALL_TESTS(argc, argv);
}

```

A test group may have a `setup()` and a `teardown()` method. The `setup` method is called prior to each test and the

`teardown()` 方法随后被调用。两者都是可选的，可以独立省略。其他方法和变量也可以在组内声明，并且对该组的所有测试都可用。

```
TEST_GROUP(Foo_Group)
{
    size_t data_bytes = 128;
    void * data;

    void setup()
    {
        data = malloc(data_bytes);
    }

    void teardown()
    {
        free(data);
    }

    void clear()
    {
        memset(data, 0, data_bytes);
    }
}
```

`teardown()` method is called after. Both are optional and either may be omitted independently. Other methods and variables may also be declared inside a group and will be available to all tests of that group.

```
TEST_GROUP(Foo_Group)
{
    size_t data_bytes = 128;
    void * data;

    void setup()
    {
        data = malloc(data_bytes);
    }

    void teardown()
    {
        free(data);
    }

    void clear()
    {
        memset(data, 0, data_bytes);
    }
}
```

第61章：Valgrind

第61.1节：字节丢失——忘记释放

这是一个调用了 `malloc` 但没有调用 `free` 的程序：

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    char *s;

    s = malloc(26); // 罪魁祸首

    return 0;
}
```

如果没有额外参数，valgrind 不会检测到这个错误。

但是如果我们开启 `--leak-check=yes` 或者 `--tool=memcheck`，它会报警并显示导致这些内存泄漏的代码行（前提是程序是以调试模式编译的）：

```
$ valgrind -q --leak-check=yes ./missing_free
==4776== 26 字节在 1 个块中被确定丢失, 丢失记录 1/1
==4776== 位于 0x4024F20: malloc (vg_replace_malloc.c:236)
==4776== 位于 0x80483F8: main (missing_free.c:9)
==4776==
```

如果程序不是以调试模式编译（例如 GCC 中没有使用 `-g` 标志），它仍然会告诉我们泄漏发生在哪个相关函数中，但不会显示具体代码行。

这让我们可以回去查看那行代码分配了哪个内存块，并尝试向前追踪为什么没有释放它。

第61.2节：使用Valgrind时最常见的错误

Valgrind 会在每行末尾以下格式提供错误发生的行号
(文件.c:行号)。Valgrind 中的错误总结如下：

```
错误总结: 1 个错误来自 1 个上下文 (抑制: 0 来自 0)
```

最常见的错误包括：

1. 非法读/写错误

```
==8451== 无效的 2 字节读取
==8451== 在 0x4E7381D: getenv (getenv.c:84)
==8451== 由 0x4EB1559: __libc_message (libc_fatal.c:80)
==8451== 由 0x4F5256B: __fortify_fail (fortify_fail.c:37)
==8451== 由 0x4F5250F: __stack_chk_fail (stack_chk_fail.c:28)
==8451== 由 0x40059C: main (valg.c:10)
==8451== 地址 0x700000007 既不是栈内存, 也不是 malloc 分配的内存, 或 (最近) 释放的内存
```

Chapter 61: Valgrind

Section 61.1: Bytes lost -- Forgetting to free

Here is a program that calls `malloc` but not `free`:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    char *s;

    s = malloc(26); // the culprit

    return 0;
}
```

With no extra arguments, valgrind will not look for this error.

But if we turn on `--leak-check=yes` or `--tool=memcheck`, it will complain and display the lines responsible for those memory leaks if the program was compiled in debug mode:

```
$ valgrind -q --leak-check=yes ./missing_free
==4776== 26 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4776== at 0x4024F20: malloc (vg_replace_malloc.c:236)
==4776== by 0x80483F8: main (missing_free.c:9)
==4776==
```

If the program is not compiled in debug mode (for example with the `-g` flag in GCC) it will still show us where the leak happened in terms of the relevant function, but not the lines.

This lets us go back and look at what block was allocated in that line and try to trace forward to see why it wasn't freed.

Section 61.2: Most common errors encountered while using Valgrind

Valgrind provides you with the *lines at which the error occurred* at the end of each line in the format
(file.c:line_no). Errors in valgrind are summarised in the following way:

```
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

The most common errors include:

1. Illegal read/write errors

```
==8451== Invalid read of size 2
==8451== at 0x4E7381D: getenv (getenv.c:84)
==8451== by 0x4EB1559: __libc_message (libc_fatal.c:80)
==8451== by 0x4F5256B: __fortify_fail (fortify_fail.c:37)
==8451== by 0x4F5250F: __stack_chk_fail (stack_chk_fail.c:28)
==8451== by 0x40059C: main (valg.c:10)
==8451== Address 0x700000007 is not stack'd, malloc'd or (recently) free'd
```

当代码开始访问不属于程序的内存时，就会发生这种情况。访问的内存大小也能帮助你判断使用的是哪个变量。

2. 未初始化变量的使用

```
--8795== 8个上下文中的第5个错误：  
--8795== 条件跳转或移动依赖于未初始化的值  
--8795==   发生在 0x4E881AF: vfprintf (vfprintf.c:1631)  
--8795==   由 0x4E8F898: printf (printf.c:33) 调用  
--8795==   由 0x400548: main (valg.c:7) 调用
```

根据错误信息，在valg.c的main函数第7行，调用printf()时传递了一个未初始化的变量给printf。

3. 非法释放内存

```
--8954== 无效的 free() / delete / delete[] / realloc()  
--8954==   发生在 0x4C2EDEB: free (位于 /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)  
--8954==   由 0x4005A8: main (valg.c:10) 调用  
--8954== 地址 0x5203040 位于大小为240的已释放内存块内偏移0字节处  
--8954==   发生在 0x4C2EDEB: free (位于 /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)  
--8954==   由 0x40059C: main (valg.c:9) 调用  
--8954== 该内存块分配于  
--8954==   发生在 0x4C2DB8F: malloc (位于 /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)  
--8954==   由 0x40058C: main (valg.c:7) 调用
```

根据valgrind，代码在valg.c的main函数第10行非法释放了内存（第二次释放），而该内存已在第9行释放，且该内存块本身是在第7行分配的。

第61.3节：运行Valgrind

```
valgrind ./my-program arg1 arg2 < test-input
```

这将运行你的程序并生成一份关于所有分配和释放的报告。它还会警告你一些常见错误，比如使用未初始化的内存、解引用指向异常位置的指针、写出通过 malloc 分配的内存块末尾，或者未能释放内存块。

第61.4节：添加标志

你还可以开启更多测试，例如：

```
valgrind -q --tool=memcheck --leak-check=yes ./my-program arg1 arg2 < test-input
```

有关（众多）选项的更多信息，请参见 valgrind --help，或访问文档 <http://valgrind.org/> 了解输出含义的详细信息。

This happens when the code starts to access memory which does not belong to the program. The size of the memory accessed also gives you an indication of what variable was used.

2. Use of Uninitialized Variables

```
--8795== 1 errors in context 5 of 8:  
--8795== Conditional jump or move depends on uninitialized value(s)  
--8795==   at 0x4E881AF: vfprintf (vfprintf.c:1631)  
--8795==   by 0x4E8F898: printf (printf.c:33)  
--8795==   by 0x400548: main (valg.c:7)
```

According to the error, at line 7 of the main of valg.c, the call to `printf()` passed an uninitialized variable to `printf`.

3. Illegal freeing of Memory

```
--8954== Invalid free() / delete / delete[] / realloc()  
--8954==   at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)  
--8954==   by 0x4005A8: main (valg.c:10)  
--8954== Address 0x5203040 is 0 bytes inside a block of size 240 free'd  
--8954==   at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)  
--8954==   by 0x40059C: main (valg.c:9)  
--8954== Block was alloc'd at  
--8954==   at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)  
--8954==   by 0x40058C: main (valg.c:7)
```

According to valgrind, the code freed the memory illegally (a second time) at line 10 of valg.c, whereas it was already freed at line 9, and the block itself was allocated memory at line 7.

Section 61.3: Running Valgrind

```
valgrind ./my-program arg1 arg2 < test-input
```

This will run your program and produce a report of any allocations and de-allocations it did. It will also warn you about common errors like using uninitialized memory, dereferencing pointers to strange places, writing off the end of blocks allocated using malloc, or failing to free blocks.

Section 61.4: Adding flags

You can also turn on more tests, such as:

```
valgrind -q --tool=memcheck --leak-check=yes ./my-program arg1 arg2 < test-input
```

See valgrind --help for more information about the (many) options, or look at the documentation at <http://valgrind.org/> for detailed information about what the output means.

第62章：常见的C语言编程习惯和开发者实践

第62.1节：比较字面量和变量

假设你正在比较 value 和某个变量

```
if ( i == 2) //错误的写法
{
doSomething;
}
```

现在假设你把==错写成了=。那你就得花不少时间去找出问题所在。

```
if( 2 == i) //正确写法
{
doSomething;
}
```

那么，如果不小心漏写了等号，编译器会报错“尝试将值赋给字面量。”

这不会在比较两个变量时保护你，但每一点帮助都很重要。

[更多信息请见这里。](#)

第62.2节：不要将函数的参数列表留空——使用void

假设你正在创建一个调用时不需要参数的函数，你面临着如何在函数原型和函数定义中定义参数列表的困境。

- 你可以选择在原型和定义中都保持参数列表为空。这样，它们看起来就像你需要的函数调用语句。
- 你在某处读到，关键字**void**的用途之一（仅有几个用途）是定义不接受任何参数的函数的参数列表。所以，这也是一种选择。

那么，哪种选择才是正确的呢？

回答：使用关键字 void

一般建议：如果一种语言为特定用途提供了某些特性，最好在代码中使用它们。例如，使用enum而不是#define宏（这是另一个例子）。

C11 标准第 6.7.6.3 节“函数声明符”，第 10 段指出：

作为列表中唯一项的无名 void 类型参数的特殊情况，表示该函数没有参数。

同一节第 14 段显示了唯一的区别：

... 作为函数定义一部分的函数声明符中的空列表表示该函数没有参数。

Chapter 62: Common C programming idioms and developer practices

Section 62.1: Comparing literal and variable

Suppose you are comparing value with some variable

```
if ( i == 2) //Bad-way
{
    doSomething;
}
```

Now suppose you have mistaken == with =. Then it will take your sweet time to figure it out.

```
if( 2 == i) //Good-way
{
    doSomething;
}
```

Then, if an equal sign is accidentally left out, the compiler will complain about an “attempted assignment to literal.” This won’t protect you when comparing two variables, but every little bit helps.

[See here](#) for more info.

Section 62.2: Do not leave the parameter list of a function blank — use void

Suppose you are creating a function that requires no arguments when it is called and you are faced with the dilemma of how you should define the parameter list in the function prototype and the function definition.

- You have the choice of keeping the parameter list empty for both prototype and definition. Thereby, they look just like the function call statement you will need.
- You read somewhere that one of the uses of keyword **void** (there are only a few of them), is to define the parameter list of functions that do not accept any arguments in their call. So, this is also a choice.

So, which is the correct choice?

ANSWER: using the keyword **void**

GENERAL ADVICE: If a language provides certain feature to use for a special purpose, you are better off using that in your code. For example, using **enums** instead of #define macros (that’s for another example).

C11 section 6.7.6.3 "Function declarators", paragraph 10, states:

The special case of an unnamed parameter of type void as the only item in the list specifies that the function has no parameters.

Paragraph 14 of that same section shows the only difference:

... An empty list in a function declarator that is part of a definition of that function specifies that the

不作为函数定义一部分的函数声明符中的空列表表示未提供有关参数数量或类型的信息。

K&R (第 72-73 页) 对上述内容的简化解释：

此外，如果函数声明不包含参数，如
double atof();，这也被理解为对atof的参数不做任何假设；所有参数检查都被关闭。空参数列表的这种特殊含义旨在允许旧的 C 程序能用新编译器编译。但在新程序中使用它是不好的做法。如果函数有参数，应声明参数；如果没有参数，应使用void。

所以你的函数原型应该是这样的：

```
int foo(void);
```

函数定义应该是这样的：

```
int foo(void)
{
    ...
    <statements>
    ...
    return 1;
}
```

使用上述声明而不是int foo()这种声明（即不使用关键字void）的一个优点是，编译器可以检测出如果你用错误的语句调用函数，比如foo(42)时的错误。如果参数列表留空，这种函数调用语句不会引起任何错误。错误会悄无声息地通过，代码仍然会执行。

这也意味着你应该这样定义main()函数：

```
int main(void)
{
    ...
    <statements>
    ...
    return 0;
}
```

注意，虽然用空参数列表定义的函数不接受参数，但它并不提供函数的原型，因此如果随后用带参数的方式调用该函数，编译器不会报错。

例如：

```
#include <stdio.h>

static void parameterless()
{
    printf("%s called", __func__);
}

int main(void)
{
    parameterless(3, "arguments", "provided");
}
```

function has no parameters. The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.

A simplified explanation provided by K&R (pgs- 72-73) for the above stuff:

Furthermore, if a function declaration does not include arguments, as in
double atof();, that too is taken to mean that nothing is to be assumed about the arguments of `atof`; all parameter checking is turned off. This special meaning of the empty argument list is intended to permit older C programs to compile with new compilers. But it's a bad idea to use it with new programs. If the function takes arguments, declare them; if it takes no arguments, use `void`.

So this is how your function prototype should look:

```
int foo(void);
```

And this is how the function definition should be:

```
int foo(void)
{
    ...
    <statements>
    ...
    return 1;
}
```

One advantage of using the above, over `int foo()` type of declaration (ie. without using the keyword `void`), is that the compiler can detect the error if you call your function using an erroneous statement like `foo(42)`. This kind of a function call statement would not cause any errors if you leave the parameter list blank. The error would pass silently, undetected and the code would still execute.

This also means that you should define the `main()` function like this:

```
int main(void)
{
    ...
    <statements>
    ...
    return 0;
}
```

Note that even though a function defined with an empty parameter list takes no arguments, it does not provide a prototype for the function, so the compiler will not complain if the function is subsequently called with arguments. For example:

```
#include <stdio.h>

static void parameterless()
{
    printf("%s called\n", __func__);
}

int main(void)
{
    parameterless(3, "arguments", "provided");
}
```

```
    return 0;  
}
```

如果该代码保存在文件proto79.c中，可以在Unix上使用GCC（演示中使用的是macOS Sierra 10.12.5上的7.1.0版本）这样编译：

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -pedantic proto79.c -o proto79  
$
```

如果使用更严格的选项编译，会出现错误：

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-style-definition -pedantic proto79.c -o proto79  
proto79.c:3:13: 错误：函数声明不是原型 [-Werror=strict-prototypes]  
static void parameterless()  
          ^~~~~~  
proto79.c: 在函数 'parameterless' 中：  
proto79.c:3:13: 错误：旧式 函数定义 [-Werror=old-style-definition]  
cc1: 所有警告均视为错误  
$
```

如果你给函数的正式原型是static void parameterless(void)，那么编译时会报错：

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-style-definition -pedantic proto79.c -o proto79  
proto79.c: 在函数 'main' 中：  
proto79.c:10:5: 错误：传递给函数 'parameterless' 的参数过多  
    parameterless(3, "arguments", "provided");  
          ^~~~~~  
proto79.c:3:13: 提示：在此处声明  
static void parameterless(void)  
          ^~~~~~  
$
```

教训一 始终确保你有函数原型，并确保编译器在你不遵守规则时提醒你。

```
    return 0;  
}
```

If that code is saved in the file proto79.c, it can be compiled on Unix with GCC (version 7.1.0 on macOS Sierra 10.12.5 used for demonstration) like this:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -pedantic proto79.c -o proto79  
$
```

If you compile with more stringent options, you get errors:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-style-definition -pedantic proto79.c -o proto79  
proto79.c:3:13: error: function declaration isn't a prototype [-Werror=strict-prototypes]  
static void parameterless()  
          ^~~~~~  
proto79.c: In function 'parameterless':  
proto79.c:3:13: error: old-style function definition [-Werror=old-style-definition]  
cc1: all warnings being treated as errors  
$
```

If you give the function the formal prototype static void parameterless(void), then the compilation gives errors:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-style-definition -pedantic proto79.c -o proto79  
proto79.c: In function 'main':  
proto79.c:10:5: error: too many arguments to function 'parameterless'  
    parameterless(3, "arguments", "provided");  
          ^~~~~~  
proto79.c:3:13: note: declared here  
static void parameterless(void)  
          ^~~~~~  
$
```

Moral — always make sure you have prototypes, and make sure your compiler tells you when you are not obeying the rules.

第63章：常见陷阱

本节讨论C程序员应注意并避免的一些常见错误。有关一些意外问题及其原因的更多信息，请参见未定义行为（Undefined behavior）。

第63.1节：算术运算中混合有符号和无符号整数

通常不建议在算术运算中混合使用有符号和无符号整数。例如，下面的示例输出结果会是什么？

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1000;
    signed int b = -1;

    if (a > b) puts("a 大于 b");
    else puts("a 小于或等于 b");

    return 0;
}
```

由于1000大于-1，你会期望输出为 a 大于 b，然而情况并非如此。

不同整型之间的算术运算是在一个由所谓的通常算术转换（参见语言规范6.3.1.8）定义的公共类型内进行的。

在这种情况下，“公共类型”是无符号整型，因为正如通常算术转换中所述，_____

714 否则，如果具有无符号整型类型的操作数的等级大于或等于另一个操作数类型的等级，则具有有符号整型类型的操作数将被转换为具有无符号整型类型的操作数的类型。

这意味着整型操作数b将在比较之前被转换为无符号整型。

当-1被转换为无符号整型时，结果是无符号整型的最大可能值，该值大于1000，这意味着 a > b为假。

第63.2节：宏是简单的字符串替换

宏是简单的字符串替换。（严格来说，它们处理的是预处理标记，而非任意字符串。）

```
#include <stdio.h>

#define SQUARE(x) x*x

int main(void) {
    printf("%d", SQUARE(1+2));return
    0;
}
```

Chapter 63: Common pitfalls

This section discusses some of the common mistakes that a C programmer should be aware of and should avoid making. For more on some unexpected problems and their causes, please see Undefined behavior

Section 63.1: Mixing signed and unsigned integers in arithmetic operations

It is usually not a good idea to mix `signed` and `unsigned` integers in arithmetic operations. For example, what will be output of following example?

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1000;
    signed int b = -1;

    if (a > b) puts("a is more than b");
    else puts("a is less or equal than b");

    return 0;
}
```

Since 1000 is more than -1 you would expect the output to be `a is more than b`, however that will not be the case.

Arithmetic operations between different integral types are performed within a common type defined by the so called usual arithmetic conversions (see the language specification, 6.3.1.8).

In this case the "common type" is `unsigned int`, Because, as stated in [Usual arithmetic conversions](#),

714 Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

This means that `int` operand `b` will get converted to `unsigned int` before the comparison.

When -1 is converted to an `unsigned int` the result is the maximal possible `unsigned int` value, which is greater than 1000, meaning that `a > b` is false.

Section 63.2: Macros are simple string replacements

Macros are simple string replacements. (Strictly speaking, they work with preprocessing tokens, not arbitrary strings.)

```
#include <stdio.h>

#define SQUARE(x) x*x

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

你可能期望这段代码打印9 (3×3)，但实际上会打印5，因为宏会被展开为
 $1+2*1+2$ 。

你应该用括号将参数和整个宏表达式括起来，以避免这个问题。

```
#include <stdio.h>

#define SQUARE(x) ((x)*(x))

int main(void) {
    printf("%d", SQUARE(1+2));return
    0;
}
```

另一个问题是宏的参数不保证只被计算一次；它们可能根本不被计算，或者被计算多次。

```
#include <stdio.h>

#define MIN(x, y) ((x) <= (y) ? (x) : (y))

int main(void) {
    int a = 0;
    printf("%d", MIN(a++, 10));printf("a
    = %d", a);
    return 0;
}
```

在这段代码中，宏将被展开为 $((a++) \leq (10) ? (a++) : (10))$ 。由于 $a++(0)$ 小于10， $a++$ 将被计算两次，这会导致a的值和从MIN返回的结果与你预期的不同。

这可以通过使用函数来避免，但请注意，函数定义会固定类型，而宏在类型上可以（过于）灵活。

```
#include <stdio.h>

int min(int x, int y) {
    return x <= y ? x : y;
}

int main(void) {
    int a = 0;
    printf("%d", min(a++, 10));printf("a
    = %d", a);
    return 0;
}
```

现在双重计算的问题已解决，但这个min函数无法处理double类型的数据，例如会发生截断。

宏指令可以分为两种类型：

```
#define OBJECT_LIKE_MACRO    后跟预处理器标记的“替换列表”
#define FUNCTION_LIKE_MACRO(with, arguments) 后跟替换列表
```

区分这两种宏类型的关键在于#define后标识符后面的字符：如果是
/paren（左括号），则是函数式宏；否则是对象式宏。如果打算编写函数式

You may expect this code to print 9 (3×3), but actually 5 will be printed because the macro will be expanded to
 $1+2*1+2$.

You should wrap the arguments and the whole macro expression in parentheses to avoid this problem.

```
#include <stdio.h>

#define SQUARE(x) ((x)*(x))

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

Another problem is that the arguments of a macro are not guaranteed to be evaluated once; they may not be evaluated at all, or may be evaluated multiple times.

```
#include <stdio.h>

#define MIN(x, y) ((x) <= (y) ? (x) : (y))

int main(void) {
    int a = 0;
    printf("%d\n", MIN(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

In this code, the macro will be expanded to $((a++) \leq (10) ? (a++) : (10))$. Since $a++(0)$ is smaller than 10, $a++$ will be evaluated twice and it will make the value of a and what is returned from MIN differ from what you may expect.

This can be avoided by using functions, but note that the types will be fixed by the function definition, whereas macros can be (too) flexible with types.

```
#include <stdio.h>

int min(int x, int y) {
    return x <= y ? x : y;
}

int main(void) {
    int a = 0;
    printf("%d\n", min(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

Now the problem of double-evaluation is fixed, but this min function cannot deal with double data without truncating, for example.

Macro directives can be of two types:

```
#define OBJECT_LIKE_MACRO    followed by a "replacement list" of preprocessor tokens
#define FUNCTION_LIKE_MACRO(with, arguments) followed by a replacement list
```

What distinguishes these two types of macros is the character that follows the identifier after #define: if it's an /paren, it is a function-like macro; otherwise, it's an object-like macro. If the intention is to write a function-like

宏，宏名结尾和(之间不能有任何空白。详见this以获取详细说明。

版本 ≥ C99

在C99或更高版本中，你可以使用static inline int min(int x, int y) { ... }。

版本 ≥ C11

在C11中，你可以为min编写一个“类型通用”的表达式。

```
#include <stdio.h>

#define min(x, y) _Generic((x),
    long double: min_ld,
    unsigned long long: min_ull,
    default: min_i \
    )(x, y)

#define gen_min(suffix, type) \
static inline type min_##suffix(type x, type y) { return (x < y) ? x : y; }

gen_min(ld, long double)
gen_min(ull, unsigned long long)
gen_min(i, int)

int main(void)
{
    unsigned long long ull1 = 50ULL;
    unsigned long long ull2 = 37ULL;
    printf("min(%llu, %llu) = %llu\n", ull1, ull2, min(ull1, ull2));long double ld1 = 3.1
41592653L;
    long double ld2 = 3.141592652L;
    printf("min(.10Lf, .10Lf) = %.10Lf\n", ld1, ld2, min(ld1, ld2));int i1 = 3141653;

    int i2 = 3141652;
    printf("min(%d, %d) = %d\n", i1, i2, min(i1, i2));return 0;
}
```

该通用表达式可以扩展到更多类型，如double、float、long long、unsigned long、long、
unsigned —并编写相应的gen_min宏调用。

第63.3节：忘记将realloc的返回值复制到临时变量中

如果 realloc失败，它会返回 NULL。如果你将原始缓冲区的值赋给 realloc的返回值，且它
返回 NULL，那么原始缓冲区（旧指针）就会丢失，导致内存泄漏。解决方法是先复制到
临时指针，如果该临时指针不为NULL，则再复制到真实缓冲区。

```
char *buf, *tmp;

buf = malloc(...);
...

/* 错误示范 */
if ((buf = realloc(buf, 16)) == NULL)
    perror("realloc");
```

macro，there must not be any white space between the end of the name of the macro and (. Check [this](#) for a detailed explanation.

Version ≥ C99

In C99 or later, you could use static inline int min(int x, int y) { ... }.

Version ≥ C11

In C11, you could write a 'type-generic' expression for min.

```
#include <stdio.h>

#define min(x, y) _Generic((x),
    long double: min_ld,
    unsigned long long: min_ull,
    default: min_i \
    )(x, y)

#define gen_min(suffix, type) \
static inline type min_##suffix(type x, type y) { return (x < y) ? x : y; }

gen_min(ld, long double)
gen_min(ull, unsigned long long)
gen_min(i, int)

int main(void)
{
    unsigned long long ull1 = 50ULL;
    unsigned long long ull2 = 37ULL;
    printf("min(%llu, %llu) = %llu\n", ull1, ull2, min(ull1, ull2));
    long double ld1 = 3.141592653L;
    long double ld2 = 3.141592652L;
    printf("min(.10Lf, .10Lf) = %.10Lf\n", ld1, ld2, min(ld1, ld2));
    int i1 = 3141653;
    int i2 = 3141652;
    printf("min(%d, %d) = %d\n", i1, i2, min(i1, i2));
    return 0;
}
```

The generic expression could be extended with more types such as double, float, long long, unsigned — and appropriate gen_min macro invocations written.

Section 63.3: Forgetting to copy the return value of realloc into a temporary

If realloc fails, it returns NULL. If you assign the value of the original buffer to realloc's return value, and if it returns NULL, then the original buffer (the old pointer) is lost, resulting in a [memory leak](#). The solution is to copy into a temporary pointer, and if that temporary is not NULL, then copy into the real buffer.

```
char *buf, *tmp;

buf = malloc(...);
...

/* WRONG */
if ((buf = realloc(buf, 16)) == NULL)
    perror("realloc");
```

```
/* 正确示范 */
if ((tmp = realloc(buf, 16)) != NULL)
    buf = tmp;
else
    perror("realloc");
```

第63.4节：忘记为\0分配额外的一个字节

当你将字符串复制到malloc分配的缓冲区时，务必记得在strlen的结果上加1。

```
char *dest = malloc(strlen(src)); /* 错误 */
char *dest = malloc(strlen(src) + 1); /* 正确 */

strcpy(dest, src);
```

这是因为strlen不包括结尾的\0字符长度。如果采用上面所示的错误方法，调用strcpy时，程序将导致未定义行为。

这同样适用于从stdin或其他来源读取已知最大长度字符串的情况。例如

```
#define MAX_INPUT_LEN 42

char buffer[MAX_INPUT_LEN]; /* 错误 */
char buffer[MAX_INPUT_LEN + 1]; /* 正确 */

scanf("%42s", buffer); /* 确保缓冲区不会溢出 */
```

第63.5节：误解数组衰减

在使用多维数组、指针数组等代码中，一个常见的问题是Type**和类型[M][N]是根本不同的类型：

```
#include <stdio.h>

void print_strings(char **strings, size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"示例 1", "示例 2", "示例 3", "示例 4"};
    print_strings(s, 4);
    return 0;
}
```

示例编译器输出：

```
file1.c: 在 函数 'main' 中：
file1.c:13:23: 错误: 传递参数 1 给 'print_strings' 时指针类型不兼容 [-W incompatible-pointer-types]
print_strings(strings, 4);
               ^
file1.c:3:10: 注释: 期望类型为 'char **' 但参数类型为 'char (*)[20]'
```

```
/* RIGHT */
if ((tmp = realloc(buf, 16)) != NULL)
    buf = tmp;
else
    perror("realloc");
```

Section 63.4: Forgetting to allocate one extra byte for \0

When you are copying a string into a `malloced` buffer, always remember to add 1 to `strlen`.

```
char *dest = malloc(strlen(src)); /* WRONG */
char *dest = malloc(strlen(src) + 1); /* RIGHT */

strcpy(dest, src);
```

This is because `strlen` does not include the trailing `\0` in the length. If you take the `WRONG` (as shown above) approach, upon calling `strcpy`, your program would invoke undefined behaviour.

It also applies to situations when you are reading a string of known maximum length from `stdin` or some other source. For example

```
#define MAX_INPUT_LEN 42

char buffer[MAX_INPUT_LEN]; /* WRONG */
char buffer[MAX_INPUT_LEN + 1]; /* RIGHT */

scanf("%42s", buffer); /* Ensure that the buffer is not overflowed */
```

Section 63.5: Misunderstanding array decay

A common problem in code that uses multidimensional arrays, arrays of pointers, etc. is the fact that `Type**` and `Type[M][N]` are fundamentally different types:

```
#include <stdio.h>

void print_strings(char **strings, size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(s, 4);
    return 0;
}
```

Sample compiler output:

```
file1.c: In function 'main':
file1.c:13:23: error: passing argument 1 of 'print_strings' from incompatible pointer type [-W incompatible-pointer-types]
print_strings(strings, 4);
               ^
file1.c:3:10: note: expected 'char **' but argument is of type 'char (*)[20]'
```

```
void print_strings(char **strings, size_t n)
```

错误提示指出，main函数中的 s 数组被传递给了函数 print_strings，而该函数期望接收的指针类型与实际传入的不同。提示中还说明了 print_strings 期望的类型以及从 main 传入的类型。

问题的根源在于所谓的数组衰变。当类型为 `char[4][20]` (4个包含20个字符的数组的数组) 的 s 被传递给函数时，它会变成指向其第一个元素的指针，就好像你写了 `&s[0]` 一样，其类型为 `char (*)[20]` (指向一个包含20个字符数组的指针)。这种情况适用于任何数组，包括指针数组、三维数组 (数组的数组的数组) 以及指向数组的指针数组。下面的表格展示了数组衰变时发生的情况。类型描述的变化部分用高亮标示以便说明：

衰变前	衰变后
<code>char [20]</code>	<code>char *</code>
<code>char [4][20]</code>	<code>char (*)[20]</code>
<code>字符 *[4]</code>	<code>字符 **</code>
<code>字符 [3][4][20]</code>	<code>字符 (*)[4][20]</code>
<code>字符 (*[4])[20]</code>	<code>字符 (**)[20]</code>

注释：(20个字符的) 数组
 (4个包含20个字符数组的) 数组
 (4个指向1个字符的指针) 数组
 (3个包含4个包含20个字符的数组的数组) 数组
 (4个指向1个包含20个字符的数组的指针) 数组

注释：指向 (1个字符) 的指针
 指向 (1个包含20个字符的数组) 的指针
 指向 (1个指向1个字符的指针) 的指针
 指向 (1个包含4个包含20个字符的数组的数组) 的指针
 指向 (1个指针指向1个包含20个字符的数组的指针)

如果数组可以衰变为指针，那么可以说指针可以被视为至少包含1个元素的数组。一个例外是空指针，它指向无效地址，因此不是数组。

数组衰变只发生一次。如果数组已经衰变为指针，那么它现在是指针，而不是数组。即使你有一个指向数组的指针，也要记住该指针可以被视为至少包含一个元素的数组，所以数组衰变已经发生。

换句话说，指向数组的指针 (`char (*)[20]`) 永远不会变成指向指针的指针 (`char **`)。要修正 print_strings 函数，只需让它接收正确的类型：

```
void print_strings(char (*strings)[20], size_t n)
/* 或者 */
void print_strings(char strings[][20], size_t n)
```

当你希望 print_strings 函数对任何字符数组通用时，会出现问题：如果字符数是30而不是20呢？或者是50呢？答案是在数组参数之前添加另一个参数：

```
#include <stdio.h>

/*
* 注意参数顺序的调整以及参数名称的变化
* 相较于之前的定义：
*   n (字符串数量)
*   => scount (字符串计数)
*
* 当然，您也可以使用以下强烈推荐的形式之一
* 作为 `strings` 参数：
*
*   字符串数组 strings[scount][ccount]
*   字符串数组 strings[][][ccount]
*/
void print_strings(size_t scount, size_t ccount, char (*strings)[ccount])
{
    size_t i;
    for (i = 0; i < scount; i++)
```

```
void print_strings(char **strings, size_t n)
```

The error states that the s array in the main function is passed to the function print_strings, which expects a different pointer type than it received. It also includes a note expressing the type that is expected by print_strings and the type that was passed to it from main.

The problem is due to something called *array decay*. What happens when s with its type `char [4][20]` (array of 4 arrays of 20 chars) is passed to the function is it turns into a pointer to its first element as if you had written `&s[0]`, which has the type `char (*)[20]` (pointer to 1 array of 20 chars). This occurs for any array, including an array of pointers, an array of arrays of arrays (3-D arrays), and an array of pointers to an array. Below is a table illustrating what happens when an array decays. Changes in the type description are highlighted to illustrate what happens:

Before Decay	After Decay
<code>char [20]</code>	<code>array of (20 chars)</code>
<code>char [4][20]</code>	<code>array of (4 arrays of 20 chars)</code>
<code>char *[4]</code>	<code>array of (4 pointers to 1 char)</code>
<code>char [3][4][20]</code>	<code>array of (3 arrays of 4 arrays of 20 chars)</code>
<code>char (*[4])[20]</code>	<code>array of (4 pointers to 1 array of 20 chars)</code>

注释：(20个字符的) 数组
 (4个包含20个字符数组的) 数组
 (4个指向1个字符的指针) 数组
 (3个包含4个包含20个字符的数组的数组) 数组
 (4个指向1个包含20个字符的数组的指针) 数组

注释：指向 (1个字符) 的指针
 指向 (1个包含20个字符的数组) 的指针
 指向 (1个指向1个字符的指针) 的指针
 指向 (1个包含4个包含20个字符的数组的数组) 的指针
 指向 (1个指针指向1个包含20个字符的数组的指针)

If an array can decay to a pointer, then it can be said that a pointer may be considered an array of at least 1 element. An exception to this is a null pointer, which points to nothing and is consequently not an array.

Array decay only happens once. If an array has decayed to a pointer, it is now a pointer, not an array. Even if you have a pointer to an array, remember that the pointer might be considered an array of at least one element, so array decay has already occurred.

In other words, a pointer to an array (`char (*)[20]`) will never become a pointer to a pointer (`char **`). To fix the print_strings function, simply make it receive the correct type:

```
void print_strings(char (*strings)[20], size_t n)
/* OR */
void print_strings(char strings[][20], size_t n)
```

A problem arises when you want the print_strings function to be generic for any array of chars: what if there are 30 chars instead of 20? Or 50? The answer is to add another parameter before the array parameter:

```
#include <stdio.h>

/*
* Note the rearranged parameters and the change in the parameter name
* from the previous definitions:
*   n (number of strings)
*   => scount (string count)
*
* Of course, you could also use one of the following highly recommended forms
* for the `strings` parameter instead:
*
*   char strings[scount][ccount]
*   char strings[][][ccount]
*/
void print_strings(size_t scount, size_t ccount, char (*strings)[ccount])
{
    size_t i;
    for (i = 0; i < scount; i++)
```

```

    puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"示例 1", "示例 2", "示例 3", "示例 4"};
    print_strings(4, 20, s);
    return 0;
}

```

编译时不会产生错误，并且输出符合预期：

```

示例 1
示例 2
示例 3
示例 4

```

第63.6节：忘记释放内存（内存泄漏）

编程的最佳实践是释放任何由你自己的代码直接分配的内存，或者通过调用内部或外部函数（例如库API如`strdup()`）隐式分配的内存。未能释放内存可能导致内存泄漏，内存泄漏会积累成大量无法被你的程序（或系统）使用的浪费内存，可能导致程序崩溃或未定义行为。如果泄漏在循环或递归函数中反复发生，问题更容易出现。程序运行时间越长，发生故障的风险越大。有时问题会立即显现；有时则可能在数小时甚至数年持续运行后才出现。内存耗尽的失败可能是灾难性的，具体取决于具体情况。

以下无限循环是一个内存泄漏的示例，该泄漏最终会耗尽可用内存，原因是调用了`getline()`函数，该函数隐式分配新内存，但未释放该内存。

```

#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    /* 下面的循环会尽可能快地泄漏内存 */
    for(;;) {
        getline(&line, &size, stdin); /* 隐式分配了新内存 */

        /* <执行其他操作> */

        line = NULL;
    }

    return 0;
}

```

相比之下，下面的代码也使用了`getline()`函数，但这次正确释放了分配的内存，避免了内存泄漏。

```

#include <stdlib.h>
#include <stdio.h>

```

```

    puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(4, 20, s);
    return 0;
}

```

Compiling it produces no errors and results in the expected output:

```

Example 1
Example 2
Example 3
Example 4

```

Section 63.6: Forgetting to free memory (memory leaks)

A programming best practice is to free any memory that has been allocated directly by your own code, or implicitly by calling an internal or external function, such as a library API like `strdup()`. Failing to free memory can introduce a memory leak, which could accumulate into a substantial amount of wasted memory that is unavailable to your program (or the system), possibly leading to crashes or undefined behavior. Problems are more likely to occur if the leak is incurred repeatedly in a loop or recursive function. The risk of program failure increases the longer a leaking program runs. Sometimes problems appear instantly; other times problems won't be seen for hours or even years of constant operation. Memory exhaustion failures can be catastrophic, depending on the circumstances.

The following infinite loop is an example of a leak that will eventually exhaust available memory by calling `getline()`, a function that implicitly allocates new memory, without freeing that memory.

```

#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    /* The loop below leaks memory as fast as it can */
    for(;;) {
        getline(&line, &size, stdin); /* New memory implicitly allocated */

        /* <do whatever> */

        line = NULL;
    }

    return 0;
}

```

In contrast, the code below also uses the `getline()` function, but this time, the allocated memory is correctly freed, avoiding a leak.

```

#include <stdlib.h>
#include <stdio.h>

```

```

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    for(;;) {
        if (getline(&line, &size, stdin) < 0) {
            free(line);
        }
        /* 处理失败情况，例如设置标志、跳出循环和/或退出 */
    }
    /* <执行其他操作> */

    free(line);
    line = NULL;
}

return 0;
}

```

内存泄漏并不总是会带来明显的后果，也不一定是功能性问题。虽然“最佳实践”要求在关键点和条件下严格释放内存，以减少内存占用和降低内存耗尽的风险，但也存在例外。例如，如果程序的运行时间和范围有限，分配失败的风险可能被认为微不足道。在这种情况下，绕过显式释放内存可能是可以接受的。例如，大多数现代操作系统会在程序终止时自动释放程序占用的所有内存，无论是由于程序失败、调用系统函数 `exit()`、进程终止，还是达到 `main()` 的末尾。在程序即将终止时显式释放内存实际上可能是多余的，甚至会带来性能损失。

如果内存不足，分配可能会失败，且应在调用栈的适当层级处理失败。上面展示的 `getline()` 是一个有趣的用例，因为它是一个库函数，不仅会分配内存且由调用者负责释放，而且可能因多种原因失败，所有这些都必须考虑。因此，使用 C API 时，务必阅读 文档 (man 页)，特别注意错误情况和内存使用，并明确哪个软件层负责释放返回的内存。

另一种常见的内存处理做法是在释放指针所指向的内存后，立即将该指针设置为 `NULL`，这样可以随时检测指针的有效性（例如检查是否为 `NULL` / 非 `NULL`），因为访问已释放的内存可能导致严重问题，如读取垃圾数据（读操作）、数据损坏（写操作）和/或程序崩溃。在大多数现代操作系统中，释放地址为 0 (`NULL`) 的内存是无操作 (NOP，即无害的)，这是 C 标准的要求——因此将指针设为 `NULL` 后，如果该指针被传递给 `free()`，就不会有重复释放内存的风险。请记住，重复释放内存可能导致非常耗时、混乱且难以诊断的故障。

第63.7节：复制过多

```

char buf[8]; /* 小缓冲区，容易溢出 */printf("你叫什么名字
? ");
scanf("%s", buf); /* 错误 */
scanf("%7s", buf); /* 正确 */

```

如果用户输入的字符串长度超过7个字符（减去1个用于空终止符），缓冲区buf后面的内存将会被覆盖。

```

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    for(;;) {
        if (getline(&line, &size, stdin) < 0) {
            free(line);
            line = NULL;
        }
        /* Handle failure such as setting flag, breaking out of loop and/or exiting */
    }
    /* <do whatever> */

    free(line);
    line = NULL;
}

return 0;
}

```

Leaking memory doesn't always have tangible consequences and isn't necessarily a functional problem. While "best practice" dictates rigorously freeing memory at strategic points and conditions, to reduce memory footprint and lower risk of memory exhaustion, there can be exceptions. For example, if a program is bounded in duration and scope, the risk of allocation failure might be considered too small to worry about. In that case, bypassing explicit deallocation might be considered acceptable. For example, most modern operating systems automatically free all memory consumed by a program when it terminates, whether it is due to program failure, a system call to `exit()`, process termination, or reaching end of `main()`. Explicitly freeing memory at the point of imminent program termination could actually be redundant or introduce a performance penalty.

Allocation can fail if insufficient memory is available, and handling failures should be accounted for at appropriate levels of the call stack. `getline()`, shown above is an interesting use-case because it is a library function that not only allocates memory it leaves to the caller to free, but can fail for a number of reasons, all of which must be taken into account. Therefore, it is essential when using a C API, to read the [documentation \(man page\)](#) and pay particular attention to error conditions and memory usage, and be aware which software layer bears the burden of freeing returned memory.

Another common memory handling practice is to consistently set memory pointers to `NULL` immediately after the memory referenced by those pointers is freed, so those pointers can be tested for validity at any time (e.g. checked for `NULL` / non-`NULL`), because accessing freed memory can lead to severe problems such as getting garbage data (read operation), or data corruption (write operation) and/or a program crash. In most modern operating systems, freeing memory location 0 (`NULL`) is a NOP (e.g. it is harmless), as required by the C standard — so by setting a pointer to `NULL`, there is no risk of double-freeing memory if the pointer is passed to `free()`. Keep in mind that double-freeing memory can lead to very time consuming, confusing, and difficult to diagnose failures.

Section 63.7: Copying too much

```

char buf[8]; /* tiny buffer, easy to overflow */

printf("What is your name?\n");
scanf("%s", buf); /* WRONG */
scanf("%7s", buf); /* RIGHT */

```

If the user enters a string longer than 7 characters (-1 for the null terminator), memory behind the buffer buf will

这会导致未定义行为。恶意黑客经常利用这一点来覆盖返回地址，并将其更改为黑客恶意代码的地址。

第63.8节：在比较时错误地写成=而不是==

=运算符用于赋值。

==运算符用于比较。

应当小心不要混淆这两者。有时会错误地写成

```
/* 将y赋值给x */
if (x = y) {
    /* 逻辑 */
}
```

而实际上想要的是：

```
/* 比较 x 是否等于 y */
if (x == y) {
    /* 逻辑 */
}
```

前者将 y 的值赋给 x 并检查该值是否非零，而不是进行比较，这等同于：

```
if ((x = y) != 0) {
    /* 逻辑 */
}
```

有时测试赋值结果是有意为之且常用的，因为它避免了代码重复和对第一次特殊处理。比较

```
while ((c = getopt_long(argc, argv, short_options, long_options, &option_index)) != -1) {
    switch (c) {
        ...
    }
}
```

与

```
c = getopt_long(argc, argv, short_options, long_options, &option_index);
while (c != -1) {
    switch (c) {
        ...
    }
}
c = getopt_long(argc, argv, short_options, long_options, &option_index);
```

现代编译器会识别这种模式，当赋值在括号内时不会发出警告，如上所示，但对于其他用法可能会警告。例如：

```
if (x = y)      /* 警告 */
if ((x = y))   /* 无警告 */
```

be overwritten. This results in undefined behavior. Malicious hackers often exploit this in order to overwrite the return address, and change it to the address of the hacker's malicious code.

Section 63.8: Mistakenly writing = instead of == when comparing

The = operator is used for assignment.

The == operator is used for comparison.

One should be careful not to mix the two. Sometimes one mistakenly writes

```
/* assign y to x */
if (x = y) {
    /* logic */
}
```

when what was really wanted is:

```
/* compare if x is equal to y */
if (x == y) {
    /* logic */
}
```

The former assigns value of y to x and checks if that value is non zero, instead of doing comparison, which is equivalent to:

```
if ((x = y) != 0) {
    /* logic */
}
```

There are times when testing the result of an assignment is intended and is commonly used, because it avoids having to duplicate code and having to treat the first time specially. Compare

```
while ((c = getopt_long(argc, argv, short_options, long_options, &option_index)) != -1) {
    switch (c) {
        ...
    }
}
```

versus

```
c = getopt_long(argc, argv, short_options, long_options, &option_index);
while (c != -1) {
    switch (c) {
        ...
    }
}
c = getopt_long(argc, argv, short_options, long_options, &option_index);
```

Modern compilers will recognise this pattern and do not warn when the assignment is inside parenthesis like above, but may warn for other usages. For example:

```
if (x = y)      /* warning */
if ((x = y))   /* no warning */
```

```
if ((x = y) != 0) /* 无警告；明确 */
```

一些程序员采用将常量放在运算符左侧的策略（通常称为Yoda条件）。因为常量是右值，这种条件写法如果使用了错误的运算符，编译器会报错。

```
if (5 = y) /* 错误 */
```

```
if (5 == y) /* 无错误 */
```

然而，这严重降低了代码的可读性，如果程序员遵循良好的C语言编码规范，这种做法并非必要，而且在比较两个变量时无效，因此并非通用解决方案。

此外，许多现代编译器在代码使用Yoda条件时可能会发出警告。

第63.9节：典型的

`scanf()`调用中换行符未被消费

当该程序

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;

    scanf("%d", &num);
    fgets(str, sizeof(str), stdin); if ((lf = str
        chr(str, '') != NULL) *lf = '\0'; printf("%d \"%s\"", num, str);

    return 0;
}
```

使用此输入执行

```
42
life
```

输出将是42 ""而不是预期的42 "life"。

这是因为42后面的换行符未被`scanf()`调用时消费，而被`fgets()`在读取`life`之前消费了。然后，`fgets()`在读取`life`之前停止读取。

为避免此问题，一种在已知行最大长度时有用的方法——例如在线评测系统中解决问题时——是避免直接使用`scanf()`，而通过`fgets()`读取所有行。

你可以使用`sscanf()`来解析读取的行。

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char line_buffer[128] = "", str[128], *lf;

    fgets(line_buffer, sizeof(line_buffer), stdin);
    sscanf(line_buffer, "%d", &num);
```

```
if ((x = y) != 0) /* no warning; explicit */
```

Some programmers use the strategy of putting the constant to the left of the operator (commonly called [Yoda conditions](#)). Because constants are rvalues, this style of condition will cause the compiler to throw an error if the wrong operator was used.

```
if (5 = y) /* Error */
```

```
if (5 == y) /* No error */
```

However, this severely reduces the readability of the code and is not considered necessary if the programmer follows good C coding practices, and doesn't help when comparing two variables so it isn't a universal solution. Furthermore, many modern compilers may give warnings when code is written with Yoda conditions.

Section 63.9: Newline character is not consumed in typical `scanf()` call

When this program

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;

    scanf("%d", &num);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

is executed with this input

```
42
life
```

the output will be 42 "" instead of expected 42 "life".

This is because a newline character after 42 is not consumed in the call of `scanf()` and it is consumed by `fgets()` before it reads `life`. Then, `fgets()` stop reading before reading `life`.

To avoid this problem, one way that is useful when the maximum length of a line is known -- when solving problems in online judge system, for example -- is avoiding using `scanf()` directly and reading all lines via `fgets()`. You can use `sscanf()` to parse the lines read.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char line_buffer[128] = "", str[128], *lf;

    fgets(line_buffer, sizeof(line_buffer), stdin);
    sscanf(line_buffer, "%d", &num);
```

```
fgets(str, sizeof(str), stdin);if ((lf = str
chr(str, '') != NULL) *lf = '\0';printf("%d \"%s\"", num, str);
return 0;
}
```

另一种方法是在使用scanf()之后并且在使用fgets()之前，读取直到遇到换行符为止。

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;
    int c;

    scanf("%d", &num);
    while ((c = getchar()) != ' ' && c != EOF);fgets(str, size
of(str), stdin);

    if ((lf = strchr(str, '')) != NULL) *lf = '\0';printf("%d \"%s\"", num, str);
    return 0;
}
```

第63.10节：在#define中添加分号

在C预处理器中很容易混淆，把它当作C语言本身的一部分，但这是错误的，因为预处理器只是一个文本替换机制。例如，如果你写

```
/* 错误示范 */
#define MAX 100;
int arr[MAX];
```

代码会展开为

```
int arr[100];
```

这会导致语法错误。解决方法是从#define行中去掉分号。几乎总是#define以分号结尾是错误的。

第63.11节：不谨慎使用分号

注意分号。以下示例

```
if (x > a);
a = x;
```

实际上意味着：

```
if (x > a) {}
a = x;
```

这意味着无论如何都会将 x 赋值给 a，这可能不是你最初想要的。

有时，缺少分号也会导致不易察觉的问题：

```
fgets(str, sizeof(str), stdin);

if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
printf("%d \"%s\"\n", num, str);
return 0;
}
```

Another way is to read until you hit a newline character after using `scanf()` and before using `fgets()`.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;
    int c;

    scanf("%d", &num);
    while ((c = getchar()) != '\n' && c != EOF);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Section 63.10: Adding a semicolon to a #define

It is easy to get confused in the C preprocessor, and treat it as part of C itself, but that is a mistake because the preprocessor is just a text substitution mechanism. For example, if you write

```
/* WRONG */
#define MAX 100;
int arr[MAX];
```

the code expands to

```
int arr[100];
```

which is a syntax error. The remedy is to remove the semicolon from the `#define` line. It is almost invariably a mistake to end a `#define` with a semicolon.

Section 63.11: Incautious use of semicolons

Be careful with semicolons. Following example

```
if (x > a);
a = x;
```

actually means:

```
if (x > a) {}
a = x;
```

which means x will be assigned to a in any case, which might not be what you wanted originally.

Sometimes, missing a semicolon will also cause an unnoticeable problem:

```
if (i < 0)
    return;
day = date[0];
hour = date[1];
minute = date[2];
```

return 后面的分号缺失，因此会返回 day=date[0]。

避免此类问题的一种技巧是，在多行条件语句和循环中始终使用大括号。例如：

```
if (x > a) {
    a = x;
}
```

第63.12节：链接时的未定义引用错误

编译过程中最常见的错误之一发生在链接阶段。错误信息类似如下：

```
$ gcc undefined_reference.c
/tmp/ccoXhwF0.o: 在函数 `main' 中：
undefined_reference.c:(.text+0x15): 未定义的引用 `foo'
collect2: 错误: ld 返回了 1 退出 状态
$
```

那么我们来看一下导致此错误的代码：

```
int foo(void);

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

这里我们看到对 foo 的声明 (int foo();)，但没有对它的定义（实际函数）。所以我们向编译器提供了函数头，但没有任何地方定义该函数，编译阶段通过，但链接器报出未定义引用错误。

要修复我们这个小程序中的错误，只需为 foo 添加一个定义：

```
/* foo 的声明 */
int foo(void);

/* foo 的定义 */
int foo(void)
{
    return 5;
}

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

```
if (i < 0)
    return;
day = date[0];
hour = date[1];
minute = date[2];
```

The semicolon behind return is missed, so day=date[0] will be returned.

One technique to avoid this and similar problems is to always use braces on multi-line conditionals and loops. For example:

```
if (x > a) {
    a = x;
}
```

Section 63.12: Undefined reference errors when linking

One of the most common errors in compilation happens during the linking stage. The error looks similar to this:

```
$ gcc undefined_reference.c
/tmp/ccoXhwF0.o: In function `main':
undefined_reference.c:(.text+0x15): undefined reference to `foo'
collect2: error: ld returned 1 exit status
$
```

So let's look at the code that generated this error:

```
int foo(void);

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

We see here a *declaration* of foo (int foo();) but no *definition* of it (actual function). So we provided the compiler with the function header, but there was no such function defined anywhere, so the compilation stage passes but the linker exits with an Undefined reference error.

To fix this error in our small program we would only have to add a *definition* for foo:

```
/* Declaration of foo */
int foo(void);

/* Definition of foo */
int foo(void)
{
    return 5;
}

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

现在这段代码可以编译了。另一种情况是，`foo()` 的源代码在一个单独的源文件 `foo.c` 中（并且有一个头文件 `foo.h` 用于声明 `foo()`，且该头文件被包含在 `foo.c` 和 `undefined_reference.c` 中）。那么解决方法是链接来自 `foo.c` 和 `undefined_reference.c` 的两个目标文件，或者编译这两个源文件：

```
$ gcc -c undefined_reference.c
$ gcc -c foo.c
$ gcc -o working_program undefined_reference.o foo.o
$
```

或者：

```
$ gcc -o working_program undefined_reference.c foo.c
$
```

一个更复杂的情况是涉及库文件，比如下面的代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv)
{
    double first;
    double second;
    double power;

    if (argc != 3)
    {
        fprintf(stderr, "用法: %s <分母> <分子>", argv[0]); return EXIT_FAILURE;
    }

    /* 将用户输入转换为数字，这里应进行额外的错误检查 */
    first = strtod(argv[1], NULL);
    second = strtod(argv[2], NULL);

    /* 使用 libm 中的 pow() 函数 - 除非此代码与 libm 一起编译，否则会导致
     * 链接错误！ */
    power = pow(first, second);

    printf("%f 的 %f 次方 = %f", first, second, power); return EXIT_SUCCESS;
}
```

代码语法正确，`pow()` 的声明来自 `#include <math.h>`，因此我们尝试编译和链接但出现如下错误：

```
$ gcc no_library_in_link.c -o no_library_in_link
/tmp/ccduQQqA.o: 在函数 `main' 中:
no_library_in_link.c:(.text+0x8b): 未定义的引用 `pow'
collect2: 错误: ld 返回了 1 退出状态
$
```

这是因为在链接阶段未找到`pow()`函数的定义。为了解决这个问题，我们必须指定要链接名为`libm`的数学库，通过指定`-lm`标志来实现。（注意，有些平台

Now this code will compile. An alternative situation arises where the source for `foo()` is in a separate source file `foo.c` (and there's a header `foo.h` to declare `foo()` that is included in both `foo.c` and `undefined_reference.c`). Then the fix is to link both the object file from `foo.c` and `undefined_reference.c`, or to compile both the source files:

```
$ gcc -c undefined_reference.c
$ gcc -c foo.c
$ gcc -o working_program undefined_reference.o foo.o
$
```

Or:

```
$ gcc -o working_program undefined_reference.c foo.c
$
```

A more complex case is where libraries are involved, like in the code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv)
{
    double first;
    double second;
    double power;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <denom> <nom>\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Translate user input to numbers, extra error checking
     * should be done here. */
    first = strtod(argv[1], NULL);
    second = strtod(argv[2], NULL);

    /* Use function pow() from libm - this will cause a linkage
     * error unless this code is compiled against libm! */
    power = pow(first, second);

    printf("%f to the power of %f = %f\n", first, second, power);

    return EXIT_SUCCESS;
}
```

The code is syntactically correct, declaration for `pow()` exists from `#include <math.h>`, so we try to compile and link but get an error like this:

```
$ gcc no_library_in_link.c -o no_library_in_link
/tmp/ccduQQqA.o: In function `main':
no_library_in_link.c:(.text+0x8b): undefined reference to `pow'
collect2: error: ld returned 1 exit status
$
```

This happens because the *definition* for `pow()` wasn't found during the linking stage. To fix this we have to specify we want to link against the math library called `libm` by specifying the `-lm` flag. (Note that there are platforms such

在 macOS 等系统中不需要 `-lmath`, 但当出现未定义引用时, 就需要该库。)

所以我们再次运行编译阶段, 这次在源文件或目标文件之后指定库:

```
$ gcc no_library_in_link.c -lmath -o library_in_link_cmd  
$ ./library_in_link_cmd 2 4  
2.000000 的 4.000000 次方 = 16.000000  
$
```

成功运行!

第 63.13 节：检查逻辑表达式是否为 'true'

原始的 C 标准没有内置布尔类型, 因此 `bool`、`true` 和 `false` 没有固有含义, 通常由程序员定义。通常 `true` 定义为 1, `false` 定义为 0。

版本 \geq C99

C99 添加了内置类型 `_Bool` 和头文件 `<stdbool.h>`, 定义了 `bool` (扩展为 `_Bool`)、`false` 和 `true`。它也允许重新定义 `bool`、`true` 和 `false`, 但指出这是一个过时的特性。

更重要的是, 逻辑表达式将任何计算结果为零的视为 `false`, 任何非零计算结果视为 `true`。例如:

```
/* 如果最高有效位被设置, 则返回 'true' */  
bool isUpperBitSet(uint8_t bitField)  
{  
    if ((bitField & 0x80) == true) /* 只有当 true 是 0x80 且 bitField 设置了该位时比较才成功 */  
    {  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}
```

在上述示例中, 函数试图检查高位是否被设置, 如果是则返回 `true`。然而, 通过显式地与 `true` 比较, `if` 语句只有在 `(bitField & 0x80)` 的值等于定义的 `true` (通常是 1, 极少是 `0x80`) 时才会成功。应显式检查预期的情况:

```
/* 如果最高有效位被设置, 则返回 'true' */  
bool isUpperBitSet(uint8_t bitField)  
{  
    if ((bitField & 0x80) == 0x80) /* 显式测试我们预期的情况 */  
    {  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}
```

或者将任何非零值视为 `true`。

as macOS where `-lmath` is not needed, but when you get the undefined reference, the library is needed.)

So we run the compilation stage again, this time specifying the library (after the source or object files):

```
$ gcc no_library_in_link.c -lmath -o library_in_link_cmd  
$ ./library_in_link_cmd 2 4  
2.000000 to the power of 4.000000 = 16.000000  
$
```

And it works!

Section 63.13: Checking logical expression against 'true'

The original C standard had no intrinsic Boolean type, so `bool`, `true` and `false` had no inherent meaning and were often defined by programmers. Typically `true` would be defined as 1 and `false` would be defined as 0.

Version \geq C99

C99 adds the built-in type `_Bool` and the header `<stdbool.h>` which defines `bool` (expanding to `_Bool`), `false` and `true`. It also allows you to redefine `bool`, `true` and `false`, but notes that this is an obsolescent feature.

More importantly, logical expressions treat anything that evaluates to zero as `false` and any non-zero evaluation as `true`. For example:

```
/* Return 'true' if the most significant bit is set */  
bool isUpperBitSet(uint8_t bitField)  
{  
    if ((bitField & 0x80) == true) /* Comparison only succeeds if true is 0x80 and bitField has  
that bit set */  
    {  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}
```

In the above example, the function is trying to check if the upper bit is set and return `true` if it is. However, by explicitly checking against `true`, the `if` statement will only succeed if `(bitfield & 0x80)` evaluates to whatever `true` is defined as, which is typically 1 and very seldom `0x80`. Either explicitly check against the case you expect:

```
/* Return 'true' if the most significant bit is set */  
bool isUpperBitSet(uint8_t bitField)  
{  
    if ((bitField & 0x80) == 0x80) /* Explicitly test for the case we expect */  
    {  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}
```

Or evaluate any non-zero value as `true`.

```

/* 如果最高有效位被设置，则返回 'true' */
bool isUpperBitSet(uint8_t bitField)
{
    /* 如果高位被设置，结果是 0x80, if 语句会将其视为 true */
    if (bitField & 0x80)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

第 63.14 节：指针运算中的额外缩放

在指针运算中，要加或减的整数不是被解释为地址的变化，而是要移动的元素数量。

```

#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + sizeof(int) * 2; /* 错误 */printf("%d %d", *ptr, *ptr2);
    return 0;
}

```

这段代码在计算赋给 `ptr2` 的指针时进行了额外的缩放。如果 `sizeof(int)` 是 4，这在现代32位环境中很常见，那么该表达式表示“`array[0]` 后的第8个元素”，这是越界的，并且会引发未定义行为。

如果想让 `ptr2` 指向 `array[0]` 后的第2个元素，应该直接加 2。

```

#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + 2;
    printf("%d %d", *ptr, *ptr2); /* 将打印 "1 3" */return 0;
}

```

使用加法运算符进行显式指针运算可能会令人困惑，因此使用数组下标可能更好。

```

#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = &ptr[2];
    printf("%d %d", *ptr, *ptr2); /* 将打印 "1 3" */return 0;
}

```

`E1[E2]` 与 `(*((E1)+(E2)))` 是相同的 (N1570 6.5.2.1, 第2段)，且 `&(E1[E2])` 等价于 `((E1)+(E2))`

```

/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    /* If upper bit is set, result is 0x80 which the if will evaluate as true */
    if (bitField & 0x80)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

Section 63.14: Doing extra scaling in pointer arithmetic

In pointer arithmetic, the integer to be added or subtracted to pointer is interpreted not as change of *address* but as number of *elements* to move.

```

#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + sizeof(int) * 2; /* wrong */
    printf("%d %d\n", *ptr, *ptr2);
    return 0;
}

```

This code does extra scaling in calculating pointer assigned to `ptr2`. If `sizeof(int)` is 4, which is typical in modern 32-bit environments, the expression stands for "8 elements after `array[0]`", which is out-of-range, and it invokes *undefined behavior*.

To have `ptr2` point at what is 2 elements after `array[0]`, you should simply add 2.

```

#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + 2;
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}

```

Explicit pointer arithmetic using additive operators may be confusing, so using array subscripting may be better.

```

#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = &ptr[2];
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}

```

`E1[E2]` is identical to `(*((E1)+(E2)))` (N1570 6.5.2.1, paragraph 2), and `&(E1[E2])` is equivalent to `((E1)+(E2))`

或者, 如果更喜欢指针运算, 将指针强制转换为不同的数据类型可以实现字节寻址。但要小心: 字节序可能成为问题, 且强制转换为除“指向字符”的指针以外的类型会导致严格别名规则问题。

```
#include <stdio.h>

int main(void) {
    int 数组[3] = {1,2,3}; // 分配了4字节 * 3
    unsigned char *指针 = (unsigned char *) 数组; // 无符号字符只占1字节
    /*
     * 现在对指针的任何指针运算都将匹配
     * 内存中的字节。指针可以被视为
     * 声明为: unsigned char 指针[12];
     */
    return 0;
}
```

第63.15节：多行注释不能嵌套

在C语言中, 多行注释 /* 和 */ 不能嵌套。

如果你使用这种注释风格标注一段代码或函数:

```
/*
 * max(): 查找数组中最大的整数并返回它。
 * 如果数组长度小于1, 结果未定义。
 * arr: 要搜索的整数数组。
 * num: arr中整数的数量。
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
max = arr[i];
    return max;
}
```

你将无法轻易地将其注释掉:

```
//尝试注释掉这段代码块...
/*
 * max(): 查找数组中最大的整数并返回它。
 * 如果数组长度小于1, 结果未定义。
 * arr: 要搜索的整数数组。
 * num: arr中整数的数量。
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
max = arr[i];
    return max;
}
```

Alternatively, if pointer arithmetic is preferred, casting the pointer to address a different data type can allow byte addressing. Be careful though: [endianness](#) can become an issue, and casting to types other than 'pointer to character' leads to [strict aliasing problems](#).

```
#include <stdio.h>

int main(void) {
    int array[3] = {1,2,3}; // 4 bytes * 3 allocated
    unsigned char *ptr = (unsigned char *) array; // unsigned chars only take 1 byte
    /*
     * Now any pointer arithmetic on ptr will match
     * bytes in memory. ptr can be treated like it
     * was declared as: unsigned char ptr[12];
     */
    return 0;
}
```

Section 63.15: Multi-line comments cannot be nested

In C, multi-line comments, /* and */, do not nest.

If you annotate a block of code or function using this style of comment:

```
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

You will not be able to comment it out easily:

```
//Trying to comment out the block...
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

```
}
```

```
//导致下面这一行出错...
*/
```

一种解决方案是使用 C99 风格的注释：

```
// max(): 查找数组中最大的整数并返回它。
// 如果数组长度小于1，结果未定义。
// arr: 要搜索的整数数组。
// num : arr 中整数的数量。
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
max = arr[i];
    return max;
}
```

现在整个代码块可以很容易地被注释掉：

```
/*
// max(): 查找数组中最大的整数并返回它。
// 如果数组长度小于1，结果未定义。
// arr: 要搜索的整数数组。
// num : arr 中整数的数量。
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
max = arr[i];
    return max;
}

*/
```

另一种解决方案是不使用注释语法来禁用代码，而是使用`#ifdef`或`#ifndef`预处理指令。这些指令可以嵌套，使你可以自由地以你喜欢的风格注释代码。

```
#define DISABLE_MAX /* 删除或注释此行以启用 max() 代码块 */

#ifndef DISABLE_MAX
/*
 * max(): 查找数组中最大的整数并返回它。
 * 如果数组长度小于1，结果未定义。
 * arr: 要搜索的整数数组。
 * num: arr 中整数的数量。
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
max = arr[i];
    return max;
}
```

```
}
```

```
//Causes an error on the line below...
*/
```

One solution is to use C99 style comments:

```
// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

Now the entire block can be commented out easily:

```
/*
// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

*/
```

Another solution is to avoid disabling code using comment syntax, using `#ifdef` or `#ifndef` preprocessor directives instead. These directives do nest, leaving you free to comment your code in the style you prefer.

```
#define DISABLE_MAX /* Remove or comment this line to enable max() code block */

#ifndef DISABLE_MAX
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

```
#endif
```

有些指南甚至建议代码段绝对不应被注释，如果需要临时禁用代码，可以使用`#if 0`指令。

参见 `#if 0` 来屏蔽代码段。

第63.16节：忽略库函数的返回值

C标准库中的几乎每个函数在成功时都会返回某些值，出错时返回其他值。例如，`malloc`在成功时会返回指向分配内存块的指针，如果函数未能分配请求的内存块，则返回空指针。因此，你应该始终检查返回值以便更容易调试。

这是错误的：

```
char* x = malloc(100000000000UL * sizeof *x);
/* more code */
scanf("%s", x); /* 这可能会引发未定义行为，如果运气不好会导致段错误，除非你的系统有大量内存 */
```

这是正确的：

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char* x = malloc(100000000000UL * sizeof *x);
    if (x == NULL) {
        perror("malloc() failed");
        exit(EXIT_FAILURE);
    }

    if (scanf("%s", x) != 1) {fprintf(stderr, "无法读取字符串");free(x);
        exit(EXIT_FAILURE);
    }

    /* 对 x 进行操作。 */

    /* 清理工作。 */
    free(x);

    return EXIT_SUCCESS;
}
```

这样你可以立即知道错误的原因，否则你可能会花费数小时在完全错误的地方寻找错误。

第63.17节：比较浮点数

浮点类型（`float`、`double` 和 `long double`）由于精度有限且以二进制格式表示数值，无法精确表示某些数字。就像我们在十进制中有循环小数（例如 $1/3$ ），在二进制中也存在无法有限表示的分数（例如 $1/3$ ，更重要的是 $1/10$ ）。不要直接比较浮点值；应使用一个误差范围（delta）。

```
#endif
```

Some guides go so far as to recommend that code sections must *never* be commented and that if code is to be temporarily disabled one could resort to using an `#if 0` directive.

See `#if 0` to block out code sections.

Section 63.16: Ignoring return values of library functions

Almost every function in C standard library returns something on success, and something else on error. For example, `malloc` will return a pointer to the memory block allocated by the function on success, and, if the function failed to allocate the requested block of memory, a null pointer. So you should always check the return value for easier debugging.

This is bad:

```
char* x = malloc(100000000000UL * sizeof *x);
/* more code */
scanf("%s", x); /* This might invoke undefined behaviour and if lucky causes a segmentation violation, unless your system has a lot of memory */
```

This is good:

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char* x = malloc(100000000000UL * sizeof *x);
    if (x == NULL) {
        perror("malloc() failed");
        exit(EXIT_FAILURE);
    }

    if (scanf("%s", x) != 1) {
        fprintf(stderr, "could not read string\n");
        free(x);
        exit(EXIT_FAILURE);
    }

    /* Do stuff with x. */

    /* Clean up. */
    free(x);

    return EXIT_SUCCESS;
}
```

This way you know right away the cause of error, otherwise you might spend hours looking for a bug in a completely wrong place.

Section 63.17: Comparing floating point numbers

Floating point types (`float`, `double` and `long double`) cannot precisely represent some numbers because they have finite precision and represent the values in a binary format. Just like we have repeating decimals in base 10 for fractions such as $1/3$, there are fractions that cannot be represented finitely in binary too (such as $1/3$, but also, more importantly, $1/10$). Do not directly compare floating point values; use a delta instead.

```

#include <float.h> // 用于 DBL_EPSILON 和 FLT_EPSILON
#include <math.h> // 用于 fabs()

int main(void)
{
    double a = 0.1; // 不精确：(二进制) 0.000110...

    // 可能为假或真
    if (a + a + a + a + a + a + a + a == 1.0) {printf("10
        * 0.1 确实是 1.0。这在一般情况下并不保证。");}
    else {printf("10 * 0.1 几乎等于 1.0。");}

    // 使用一个很小的增量值。
    if (fabs(a + a + a + a + a + a + a + a - 1.0) < 0.000001) {
        // C99 5.2.4.2.2p8 保证 double 类型至少有 10 位小数的精度
        // 的精度。
        printf("10 * 0.1 几乎等于 1.0。");}
    else {printf("10 * 0.1 确实是 1.0。");}
}

return 0;
}

```

另一个例子：

```

gcc -O3 -g -I./inc -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes
-Wold-style-definition rd11.c -o rd11 -L./lib -lsq
#include <stdio.h>
#include <math.h>

static inline double rel_diff(double a, double b)
{
    return fabs(a - b) / fmax(fabs(a), fabs(b));
}

int main(void)
{
    double d1 = 3.14159265358979;
    double d2 = 355.0 / 113.0;

    double epsilon = 1.0;
    for (int i = 0; i < 10; i++)
    {
        if (rel_diff(d1, d2) < epsilon)
            printf("%d:%.10f <=> %.10f 在容差范围内 %.10f (相对差 %.4E)",
1, d2, epsilon, rel_diff(d1, d2));else
            printf("%d:%.10f <=> %.10f 超出容差范围 %.10f (相对差 %.4E)",
1, d2, epsilon, rel_diff(d1, d2));epsilon /= 10.0;
    }
    return 0;
}

```

输出：

```

0:3.1415926536 <=> 3.1415929204 在容差范围内 1.0000000000 (相对差 8.4914E-08)
1:3.1415926536 <=> 3.1415929204 在容差范围内 0.1000000000 (相对差 8.4914E-08)
2:3.1415926536 <=> 3.1415929204 在容差范围内 0.0100000000 (相对差 8.4914E-08)
3:3.1415926536 <=> 3.1415929204 在容差范围内 0.0010000000 (相对差 8.4914E-08)
4:3.1415926536 <=> 3.1415929204 在容差范围内 0.0001000000 (相对差 8.4914E-08)

```

```

#include <float.h> // for DBL_EPSILON and FLT_EPSILON
#include <math.h> // for fabs()

int main(void)
{
    double a = 0.1; // imprecise: (binary) 0.000110...

    // may be false or true
    if (a + a + a + a + a + a + a + a == 1.0) {
        printf("10 * 0.1 is indeed 1.0. This is not guaranteed in the general case.\n");
    }

    // Using a small delta value.
    if (fabs(a + a + a + a + a + a + a + a - 1.0) < 0.000001) {
        // C99 5.2.4.2.2p8 guarantees at least 10 decimal digits
        // of precision for the double type.
        printf("10 * 0.1 is almost 1.0.\n");
    }

    return 0;
}

```

Another example:

```

gcc -O3 -g -I./inc -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes
-Wold-style-definition rd11.c -o rd11 -L./lib -lsq
#include <stdio.h>
#include <math.h>

static inline double rel_diff(double a, double b)
{
    return fabs(a - b) / fmax(fabs(a), fabs(b));
}

int main(void)
{
    double d1 = 3.14159265358979;
    double d2 = 355.0 / 113.0;

    double epsilon = 1.0;
    for (int i = 0; i < 10; i++)
    {
        if (rel_diff(d1, d2) < epsilon)
            printf("%d:%.10f <=> %.10f within tolerance %.10f (rel diff %.4E)\n",
i, d1, d2, epsilon, rel_diff(d1, d2));
        else
            printf("%d:%.10f <=> %.10f out of tolerance %.10f (rel diff %.4E)\n",
i, d1, d2, epsilon, rel_diff(d1, d2));
        epsilon /= 10.0;
    }
    return 0;
}

```

Output:

```

0:3.1415926536 <=> 3.1415929204 within tolerance 1.0000000000 (rel diff 8.4914E-08)
1:3.1415926536 <=> 3.1415929204 within tolerance 0.1000000000 (rel diff 8.4914E-08)
2:3.1415926536 <=> 3.1415929204 within tolerance 0.0100000000 (rel diff 8.4914E-08)
3:3.1415926536 <=> 3.1415929204 within tolerance 0.0010000000 (rel diff 8.4914E-08)
4:3.1415926536 <=> 3.1415929204 within tolerance 0.0001000000 (rel diff 8.4914E-08)

```

```
5:3.1415926536 <=> 3.1415929204 在容差范围内 0.0000100000 (相对差 8.4914E-08)
6:3.1415926536 <=> 3.1415929204 在容差范围内 0.0000010000 (相对差 8.4914E-08)
7:3.1415926536 <=> 3.1415929204 在容差范围内 0.0000001000 (相对差 8.4914E-08)
8:3.1415926536 <=> 3.1415929204 超出容差范围 0.0000000100 (相对差 8.4914E-08)
9:3.1415926536 <=> 3.1415929204 超出容差范围 0.0000000010 (相对差 8.4914E-08)
```

第63.18节：浮点字面量默认类型为double

初始化float类型变量为字面量值或与字面量值比较时必须小心，因为普通浮点字面量如0.1的类型是double。这可能导致意外情况：

```
#include <stdio.h>
int main() {
    float n;
    n = 0.1;
    if (n > 0.1) printf("奇怪");return 0;
}
// 当 n 是浮点数时打印"Wierd"
```

这里，n 被初始化并四舍五入为单精度，结果值为 0.1000000149011612。然后，n 被转换回双精度以与 0.1 字面量（等于 0.1000000000000001）进行比较，导致不匹配。

除了舍入误差外，将 float 变量与 double 字面量混合使用会导致在没有硬件支持双精度的平台上性能下降。

第 63.19 节：使用字符常量代替字符串字面量，反之亦然

在 C 语言中，字符常量和字符串字面量是不同的东西。

用单引号括起来的字符，如 'a'，是一个字符常量。字符常量是一个整数，其值是代表该字符的字符编码。如何解释包含多个字符的字符常量，如 'abc'，是由实现定义的。

用双引号括起来的零个或多个字符，如 "abc"，是一个字符串字面量。字符串字面量是一个不可修改的数组，其元素类型为 char。双引号中的字符串加上终止的空字符是内容，因此 "abc" 有 4 个元素 ('a', 'b', 'c', '\0')。

在此示例中，使用了字符常量而应使用字符串字面量。该字符常量将以实现定义的方式转换为指针，且转换后的指针很可能无效，因此此示例将引发未定义行为。

```
#include <stdio.h>

int main(void) {
    const char *hello = 'hello, world'; /* 错误 */
    puts(hello);
    return 0;
}
```

在此示例中，使用了字符串字面量而应使用字符常量。从字符串字面量转换的指针将以实现定义的方式转换为整数，并将转换为 char

```
5:3.1415926536 <=> 3.1415929204 within tolerance 0.0000100000 (rel diff 8.4914E-08)
6:3.1415926536 <=> 3.1415929204 within tolerance 0.0000010000 (rel diff 8.4914E-08)
7:3.1415926536 <=> 3.1415929204 within tolerance 0.0000001000 (rel diff 8.4914E-08)
8:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000100 (rel diff 8.4914E-08)
9:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000010 (rel diff 8.4914E-08)
```

Section 63.18: Floating point literals are of type double by default

Care must be taken when initializing variables of type `float` to literal values or comparing them with literal values, because regular floating point literals like `0.1` are of type `double`. This may lead to surprises:

```
#include <stdio.h>
int main() {
    float n;
    n = 0.1;
    if (n > 0.1) printf("Wierd\n");
    return 0;
}
// Prints "Wierd" when n is float
```

Here, n gets initialized and rounded to single precision, resulting in value 0.1000000149011612. Then, n is converted back to double precision to be compared with 0.1 literal (which equals to 0.1000000000000001), resulting in a mismatch.

Besides rounding errors, mixing `float` variables with `double` literals will result in poor performance on platforms which don't have hardware support for double precision.

Section 63.19: Using character constants instead of string literals, and vice versa

In C, character constants and string literals are different things.

A character surrounded by single quotes like '`a`' is a *character constant*. A character constant is an integer whose value is the character code that stands for the character. How to interpret character constants with multiple characters like '`abc`' is implementation-defined.

Zero or more characters surrounded by double quotes like "`abc`" is a *string literal*. A string literal is an unmodifiable array whose elements are type `char`. The string in the double quotes plus terminating null-character are the contents, so "`abc`" has 4 elements ('`a`', '`b`', '`c`', '\0')

In this example, a character constant is used where a string literal should be used. This character constant will be converted to a pointer in an implementation-defined manner and there is little chance for the converted pointer to be valid, so this example will invoke *undefined behavior*.

```
#include <stdio.h>

int main(void) {
    const char *hello = 'hello, world'; /* bad */
    puts(hello);
    return 0;
}
```

In this example, a string literal is used where a character constant should be used. The pointer converted from the string literal will be converted to an integer in an implementation-defined manner, and it will be converted to `char`

以实现定义的方式。（如何将一个整数转换为无法表示该值的有符号类型是实现定义的，是否char为有符号类型也是实现定义的。）输出将是一些无意义的内容。

```
#include <stdio.h>intmain(void) {char c = "a"; /* 错误 */printf("%c", c);return 0;}
```

在几乎所有情况下，编译器都会对这些混淆发出警告。如果没有，您需要使用更多的编译器警告选项，或者建议您使用更好的编译器。

第63.20节：递归函数——缺少基本条件

计算一个数的阶乘是递归函数的经典例子。

缺少基本条件：

```
#include <stdio.h>

int factorial(int n)
{
    return n * factorial(n - 1);
}

int main()
{
    printf("阶乘 %d = %d", 3, factorial(3));return 0;
}
```

典型输出：段错误: 11

这个函数的问题是它会无限循环，导致段错误—它需要一个基准条件来停止递归。

基本条件声明：

```
#include <stdio.h>

int factorial(int n)
{
    if (n == 1) // 基本条件，在设计递归函数时非常关键。
    {
        return 1;
    }
    else
    {
        return n * factorial(n - 1);
    }
}

int main()
```

in an implementation-defined manner. (How to convert an integer to a signed type which cannot represent the value to convert is implementation-defined, and whether `char` is signed is also implementation-defined.) The output will be some meaningless thing.

```
#include <stdio.h>

int main(void) {
    char c = "a"; /* bad */
    printf("%c\n", c);
    return 0;
}
```

In almost all cases, the compiler will complain about these mix-ups. If it doesn't, you need to use more compiler warning options, or it is recommended that you use a better compiler.

Section 63.20: Recursive function – missing out the base condition

Calculating the factorial of a number is a classic example of a recursive function.

Missing the Base Condition:

```
#include <stdio.h>

int factorial(int n)
{
    return n * factorial(n - 1);
}

int main()
{
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Typical output: Segmentation fault: 11

The problem with this function is it would loop infinitely, causing a segmentation fault — it needs a base condition to stop the recursion.

Base Condition Declared:

```
#include <stdio.h>

int factorial(int n)
{
    if (n == 1) // Base Condition, very crucial in designing the recursive functions.
    {
        return 1;
    }
    else
    {
        return n * factorial(n - 1);
    }
}

int main()
```

```
printf("阶乘 %d = %d", 3, factorial(3));return 0;  
}
```

示例输出

阶乘 3 = 6

该函数将在 n 等于 1 时终止（前提是 n 的初始值足够小——当 int 为 32 位时，上限为 12）。

需遵循的规则：

1. 初始化算法。递归程序通常需要一个种子值作为起点。这可以通过传递给函数的参数实现，或者通过提供一个非递归的入口函数来设置递归计算的种子值。
2. 检查当前处理的值是否匹配基本情况。如果是，则处理并返回该值。
3. 将答案重新定义为更小或更简单的子问题。
4. 在子问题上运行算法。
5. 将结果结合起来形成答案。
6. 返回结果。

来源：[递归函数](#)

第63.21节：越界访问数组

数组是从零开始的，也就是说索引总是从0开始，到数组长度减1结束。因此，以下代码不会输出数组的第一个元素，并且会输出最后打印值的垃圾数据。

```
#include <stdio.h>  
  
int main(void)  
{  
    int x = 0;  
    int myArray[5] = {1, 2, 3, 4, 5}; //声明5个元素  
  
    for(x = 1; x <= 5; x++) //从1循环到5。  
        printf("%d", myArray[x]);  
  
    printf("");return  
0;}
```

输出：2 3 4 5 垃圾值

下面演示了实现期望输出的正确方法：

```
#include <stdio.h>  
  
int main(void)  
{  
    int x = 0;  
    int myArray[5] = {1, 2, 3, 4, 5}; //声明5个元素  
  
    for(x = 0; x < 5; x++) //从0循环到4。  
        printf("%d", myArray[x]);
```

```
printf("Factorial %d = %d\n", 3, factorial(3));  
return 0;  
}
```

Sample output

Factorial 3 = 6

This function will terminate as soon as it hits the condition n is equal to 1 (provided the initial value of n is small enough — the upper bound is 12 when int is a 32-bit quantity).

Rules to be followed:

1. Initialize the algorithm. Recursive programs often need a seed value to start with. This is accomplished either by using a parameter passed to the function or by providing a gateway function that is non-recursive but that sets up the seed values for the recursive calculation.
2. Check to see whether the current value(s) being processed match the base case. If so, process and return the value.
3. Redefine the answer in terms of a smaller or simpler sub-problem or sub-problems.
4. Run the algorithm on the sub-problem.
5. Combine the results in the formulation of the answer.
6. Return the results.

Source: [Recursive Function](#)

Section 63.21: Overstepping array boundaries

Arrays are zero-based, that is the index always starts at 0 and ends with index array length minus 1. Thus the following code will not output the first element of the array and will output garbage for the final value that it prints.

```
#include <stdio.h>  
  
int main(void)  
{  
    int x = 0;  
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements  
  
    for(x = 1; x <= 5; x++) //Looping from 1 till 5.  
        printf("%d\t", myArray[x]);  
  
    printf("\n");  
    return 0;  
}
```

Output: 2 3 4 5 GarbageValue

The following demonstrates the correct way to achieve the desired output:

```
#include <stdio.h>  
  
int main(void)  
{  
    int x = 0;  
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements  
  
    for(x = 0; x < 5; x++) //Looping from 0 till 4.  
        printf("%d\t", myArray[x]);
```

```
printf("");return  
0;}
```

输出：1 2 3 4 5

在操作数组之前，了解数组的长度非常重要，否则可能会破坏缓冲区或通过访问越界的内存位置导致段错误。

第63.22节：向期望“真实”多维数组的函数传递非连续数组

使用malloc、calloc和realloc分配多维数组时，一个常见的模式是通过多次调用分配内层数组（即使调用只出现一次，也可能在循环中）：

```
/* 也可以是使用malloc分配外层数组的int **。 */  
int *array[4];  
int i;  
  
/* 分配4个包含16个int的数组。 */  
for (i = 0; i < 4; i++)  
    array[i] = malloc(16 * sizeof(*array[i]));
```

一个内层数组的最后一个元素与下一个内层数组的第一个元素之间的字节差可能不为0，这与“真实”多维数组（例如int array[4][16];）中的情况不同：

```
/* 0x40003c, 0x402000 */  
printf("%p, %p", (void *)(array[0] + 15), (void *)array[1]);
```

考虑到int的大小，你会得到8128字节的差异（8132-4），即2032个int大小的数组元素，这就是问题所在：“真正的”多维数组元素之间没有间隙。

如果你需要使用动态分配的数组，而函数期望的是“真正的”多维数组，你应该分配一个类型为int *的对象，并使用算术运算进行计算：

```
void func(int M, int N, int *array);  
  
/* 等同于声明 `int array[M][N] = {{0}};` 并赋值给 array4_16[i][j]。 */  
int *array;  
int M = 4, N = 16;  
array = calloc(M, N * sizeof(*array));  
array[i * N + j] = 1;  
func(M, N, array);
```

如果N是宏或整数常量而非变量，代码可以在分配指向数组的指针后，简单地使用更自然的二维数组表示法：

```
void func(int M, int N, int *array);  
#define N 16  
void func_N(int M, int (*array)[N]);  
  
int M = 4;  
int (*array)[N];  
array = calloc(M, sizeof(*array));  
array[i][j] = 1;
```

```
printf("\n");  
return 0;  
}
```

Output: 1 2 3 4 5

It is important to know the length of an array before working with it as otherwise you may corrupt the buffer or cause a segmentation fault by accessing memory locations that are out of bounds.

Section 63.22: Passing unadjacent arrays to functions expecting "real" multidimensional arrays

When allocating multidimensional arrays with malloc, calloc, and realloc, a common pattern is to allocate the inner arrays with multiple calls (even if the call only appears once, it may be in a loop):

```
/* Could also be `int **` with malloc used to allocate outer array. */  
int *array[4];  
int i;  
  
/* Allocate 4 arrays of 16 ints. */  
for (i = 0; i < 4; i++)  
    array[i] = malloc(16 * sizeof(*array[i]));
```

The difference in bytes between the last element of one of the inner arrays and the first element of the next inner array may not be 0 as they would be with a "real" multidimensional array (e.g. int array[4][16];):

```
/* 0x40003c, 0x402000 */  
printf("%p, %p\n", (void *)(array[0] + 15), (void *)array[1]);
```

Taking into account the size of int, you get a difference of 8128 bytes (8132-4), which is 2032 int-sized array elements, and that is the problem: a "real" multidimensional array has no gaps between elements.

If you need to use a dynamically allocated array with a function expecting a "real" multidimensional array, you should allocate an object of type int * and use arithmetic to perform calculations:

```
void func(int M, int N, int *array);  
  
/* Equivalent to declaring `int array[M][N] = {{0}};` and assigning to array4_16[i][j]. */  
int *array;  
int M = 4, N = 16;  
array = calloc(M, N * sizeof(*array));  
array[i * N + j] = 1;  
func(M, N, array);
```

If N is a macro or an integer literal rather than a variable, the code can simply use the more natural 2-D array notation after allocating a pointer to an array:

```
void func(int M, int N, int *array);  
#define N 16  
void func_N(int M, int (*array)[N]);  
  
int M = 4;  
int (*array)[N];  
array = calloc(M, sizeof(*array));  
array[i][j] = 1;
```

```
/* 转换为 `int *` 在这里有效，因为 `array` 是一个连续的  $M \times N$  个 int 的内存块，没有间隙，  
就像 `int array2[M * N];` 和 `int array3[M][N];` 一样。 */
```

```
func(M, N, (int *)array);  
func_N(M, array);
```

版本 \geq C99

如果 N 不是宏或整数常量，那么 array 将指向一个变长数组 (VLA)。这仍然可以通过将 array 转换为 int * 来与 func 一起使用，并且一个新的函数 func_vla 将替代 func_N：

```
void func(int M, int N, int *array);  
void func_vla(int M, int N, int array[M][N]);  
...  
  
int M = 4, N = 16;  
int (*array)[N];  
array = calloc(M, sizeof(*array));  
array[i][j] = 1;  
func(M, N, (int *)array);  
func_vla(M, N, array);
```

版本 \geq C11

注意：从C11开始，VLA是可选的。如果您的实现支持C11并定义了宏 __STDC_NO_VLA__ 为 1，则只能使用C99之前的方法。

```
/* Cast to `int *` works here because `array` is a single block of  $M \times N$  ints with no gaps,  
just like `int array2[M * N];` and `int array3[M][N];` would be. */
```

```
func(M, N, (int *)array);  
func_N(M, array);
```

Version \geq C99

If N is not a macro or an integer literal, then array will point to a variable-length array (VLA). This can still be used with func by casting to int * and a new function func_vla would replace func_N:

```
void func(int M, int N, int *array);  
void func_vla(int M, int N, int array[M][N]);  
...  
  
int M = 4, N = 16;  
int (*array)[N];  
array = calloc(M, sizeof(*array));  
array[i][j] = 1;  
func(M, N, (int *)array);  
func_vla(M, N, array);
```

Version \geq C11

Note: VLAs are optional as of C11. If your implementation supports C11 and defines the macro __STDC_NO_VLA__ to 1, you are stuck with the pre-C99 methods.

致谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

2501	第23、28、33、34和36章
3442	第4章
4386427	第11章、第19章和第46章
A_B	第19章
障碍	第61章和第63章
亚历杭德罗·卡罗	第12章
阿莱克西·托尔哈莫	第28章
亚历克斯	第55章
亚历克斯·加西亚	第30章
alk	第1、3、4、5、6、9、10、11、12、13、14、15、20、21、24、25、28、30、37、40、43、44、46、49和63章
阿尔特塞	第22章
阿马尼·基卢曼加	第4章
AnArrayOfFunctions	第10和44章
安德里亚·科贝利	第4章
安德烈·马尔凯耶夫	第6章和第46章
安库什	第1章、第2章、第46章和第61章
安蒂·哈帕拉	第22章、第28章和第46章
阿马利	第23章、第28章和第54章
阿图尔FH	第1章
阿谢利	第10章和第33章
巴赫蒂亚尔·哈桑	第4章
本·斯特凡	第28章
本G	第4章
贝文森	第6章、第11章、第16章、第20章、第30章、第37章、第38章和第63章
布莱克西尔弗	第37章和第44章
布拉戈维斯特·布尤克利耶夫	第3章、第14章、第33章、第41章和第43章
blatinox	第4章
鲍勃	第5章
布雷登·贝斯特	第5章
bta	第22、28和33章
BurnsBA	第28章
布泽尔	第42章
目录编号	第6章
cdrini	第10章
钱德拉哈斯·阿鲁里	第2、42和61章
chqrlie	第1章
克里斯·斯普拉格	第6章
克里斯托夫	第28章
时之狐 (Chrono Kitsune)	第6、13、43和63章
钦巴利	第35章
clearlight	第63章
科迪·格雷	第6章
cshu	第28章
cSmout	第10章
C_L_S	第59章
DaBler	第28章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

2501	Chapters 23, 28, 33, 34 and 36
3442	Chapter 4
4386427	Chapters 11, 19 and 46
A_B	Chapter 19
abacles	Chapters 61 and 63
Alejandro Caro	Chapter 12
Aleksi Torhamo	Chapter 28
Alex	Chapter 55
Alex Garcia	Chapter 30
alk	Chapters 1, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 15, 20, 21, 24, 25, 28, 30, 37, 40, 43, 44, 46, 49 and 63
Altece	Chapter 22
Amani Kilumanga	Chapter 4
AnArrayOfFunctions	Chapters 10 and 44
Andrea Corbelli	Chapter 4
Andrey Markeev	Chapters 6 and 46
Ankush	Chapters 1, 2, 46 and 61
Antti Haapala	Chapters 22, 28 and 46
Armali	Chapters 23, 28 and 54
ArturFH	Chapter 1
AShelly	Chapters 10 and 33
Bakhtiar Hasan	Chapter 4
Ben Steffan	Chapter 28
BenG	Chapter 4
bevenson	Chapters 6, 11, 16, 20, 30, 37, 38 and 63
Blacksilver	Chapters 37 and 44
Blagovest Buyukliev	Chapters 3, 14, 33, 41 and 43
blatinox	Chapter 4
Bob	Chapter 5
Braden Best	Chapter 5
bta	Chapters 22, 28 and 33
BurnsBA	Chapter 28
Buser	Chapter 42
catalogue_number	Chapter 6
cdrini	Chapter 10
Chandrahas Aroori	Chapters 2, 42 and 61
chqrlie	Chapter 1
Chris Sprague	Chapter 6
Christoph	Chapter 28
Chrono Kitsune	Chapters 6, 13, 43 and 63
Cimbali	Chapter 35
clearlight	Chapter 63
Cody Gray	Chapter 6
cshu	Chapter 28
cSmout	Chapter 10
C_L_S	Chapter 59
DaBler	Chapter 28

达克什·古普塔	第46章
达米恩	第4章和第6章
丹尼尔	第6章
丹尼尔·朱尔	第28章
丹尼尔·波尔特斯	第22章
达留什	第4章、第10章、第30章和第31章
暗尘	第28章
大卫·雷法利	第24章
deamentiaemundi	第26章
depperm	第6章
德万什·坦登	第61章
dhein	第46章
dkrmr	第46章
德米特里·格里戈里耶夫	第63章
别担心，孩子	第22章
唐老鸭	第1章
doppelheathen	第46章
梦想家	第63章
drov	第19章和第61章
DrPrItay	第30章
不知道	第63章
dvhh	第19章和第46章
dylanweber	第6章
e.jahandar	第22章
埃德·科特雷尔	第1章
埃尔扎尔	第10章和第13章
伊莱·萨多夫	第10章
elsloo	第22章和第46章
embedded_guy	第39章
EOF	第46章
erebos	第22章
EsmaeelE	第1章、第13章、第38章和第45章
eush77	第32章和第33章
EvilTeach	第9章
费萨尔·穆迪尔	第16章和第22章
了不起的狐狸先生	第4章、第9章和第22章
fastlearner	第30章
FedeWar	第6章、第22章和第63章
菲利普·奥尔贝格	第24章
菲拉斯·莫阿拉	第10章和第22章
fluter	第21章
foxtrot9	第18章和第22章
弗雷德·巴克莱	第63章
ganchito55	第29章
甘尼什·库马尔	第25章
加文·海厄姆	第22章和第63章
gdc	第46章
乔治·斯托克尔	第25章
乔治·莫尼瓦	第22章、第28章和第63章
gmug	第13章
善行	第15章和第42章
gsamaras	第4章和第28章
haccks	第6章、第8章、第22章和第23章

Daksh Gupta	Chapter 46
Damien	Chapters 4 and 6
Daniel	Chapter 6
Daniel Jour	Chapter 28
Daniel Porteous	Chapter 22
Dariusz	Chapters 4, 10, 30 and 31
DarkDust	Chapter 28
David Refaeli	Chapter 24
deamentiaemundi	Chapter 26
depperm	Chapter 6
Devansh Tandon	Chapter 61
dhein	Chapter 46
dkrmr	Chapter 46
Dmitry Grigoryev	Chapter 63
Don't You Worry Child	Chapter 22
Donald Duck	Chapter 1
doppelheathen	Chapter 46
Dreamer	Chapter 63
drov	Chapters 19 and 61
DrPrItay	Chapter 30
Dunno	Chapter 63
dvhh	Chapters 19 and 46
dylanweber	Chapter 6
e.jahandar	Chapter 22
Ed Cottrell	Chapter 1
Elazar	Chapters 10 and 13
Eli Sadoff	Chapter 10
elsloo	Chapters 22 and 46
embedded_guy	Chapter 39
EOF	Chapter 46
erebos	Chapter 22
EsmaeelE	Chapters 1, 13, 38 and 45
eush77	Chapters 32 and 33
EvilTeach	Chapter 9
Faisal Mudhir	Chapters 16 and 22
Fantastic Mr Fox	Chapters 4, 9 and 22
fastlearner	Chapter 30
FedeWar	Chapters 6, 22 and 63
Filip Allberg	Chapter 24
Firas Moalla	Chapters 10 and 22
fluter	Chapter 21
foxtrot9	Chapters 18 and 22
Fred Barclay	Chapter 63
ganchito55	Chapter 29
ganesh kumar	Chapter 25
Gavin Higham	Chapters 22 and 63
gdc	Chapter 46
George Stocker	Chapter 25
Giorgi Moniava	Chapters 22, 28 and 63
gmug	Chapter 13
GoodDeeds	Chapters 15 and 42
gsamaras	Chapters 4 and 28
haccks	Chapters 6, 8, 22 and 23

[haltode](#)
[哈里·约翰斯顿](#)
[赫曼特·库马尔](#)
[hexwab](#)
[hlovdal](#)
[hmijail](#)
[honk](#)
[hrs](#)
[immerhart](#)
[疯狂的](#)
[iRove](#)
[伊谢·佩莱德](#)
[伊斯卡尔·贾拉克](#)
[我不存在 我不存在](#)
[雅各布·H](#)
[贾斯敏·索兰基](#)
[jasoninnn](#)
[javac](#)
[让](#)
[让·维托](#)
[延斯·古斯特德](#)
[杰里米](#)
[杰里米·蒂恩](#)
[杰斯弗曼](#)
[约翰](#)
[约翰·博德](#)
[约翰·博林格](#)
[约翰·伯格](#)
[约翰·哈斯考尔](#)
[乔纳斯Cz](#)
[乔纳森·莱夫勒](#)
[乔纳森·莱因哈特](#)
[乔西](#)
[胡安·T](#)
[juleslasne](#)
[贾斯汀](#)
[jxh](#)
[卡米奇洛](#)
[kamoroso94](#)
[kdopen](#)
[肯·Y](#)
[Kerrek SB](#)
[克拉斯·林德巴克](#)
[库萨兰达](#)
[L.V.拉奥](#)
[LaneL](#)
[lardenn](#)
[Leandros](#)
[LiHRaM](#)
[李久·托马斯](#)
[法夸德勋爵](#)

第22章
第22章
第22章
第29章
第24章、第30章和第63章
第28章
第28章和第46章
第4章
第14章
第5章
第11章和第20章
第6章和第63章
第1章
第4章
第28章
第10章
第6章和第12章
第30章
第6、28、32、37、39和43章
第20章
第1、3、4、5、6、7、8、9、10、12、13、15、22、23、25、27、28、30、31、33、34、
35、36、37、39、40、41、43、44、46、47、48、49、55和57章
第63章
第40章
第41和45章
第28章
第23章
第6、10、28、35和47章
第24章
第63章
第1章
第1、2、4、6、7、9、11、13、14、17、19、20、21、22、28、30、32、33、39、42、44、4
5、46、49、50、53、59、61、62和63章
第44章
第11、21、33、37和56章
第1章
第1和46章
第30章
第15、33和37章
第28章
第13章
第4章
第3章
第8章和第18章
第7章
第1章
第4、10、15、16、22、43和52章
第22章
第21章
第1、3、4、10、24、25、28、29、30、31和33章
第1章
第10、17和45章
第63章

[haltode](#)
[Harry Johnston](#)
[Hemant Kumar](#)
[hexwab](#)
[hlovdal](#)
[hmijail](#)
[honk](#)
[hrs](#)
[immerhart](#)
[Insane](#)
[iRove](#)
[Ishay Peled](#)
[Iskar Jarak](#)
[Iwillnotexist Idonotexist](#)
[Jacob H](#)
[Jasmin Solanki](#)
[jasoninnn](#)
[javac](#)
[Jean](#)
[Jean Vitor](#)
[Jens Gustedt](#)
[Jeremy](#)
[Jeremy Thien](#)
[Jesferman](#)
[John](#)
[John Bode](#)
[John Bollinger](#)
[John Burger](#)
[John Hascall](#)
[JonasCz](#)
[Jonathan Leffler](#)
[Jonathon Reinhart](#)
[Jossi](#)
[Juan T](#)
[juleslasne](#)
[Justin](#)
[jxh](#)
[Kamiccolo](#)
[kamoroso94](#)
[kdopen](#)
[Ken Y](#)
[Kerrek SB](#)
[Klas Lindbäck](#)
[Kusalananda](#)
[L.V.Rao](#)
[LaneL](#)
[lardenn](#)
[Leandros](#)
[LiHRaM](#)
[Liju Thomas](#)
[Lord Farquaad](#)

Chapter 22
Chapter 22
Chapter 22
Chapter 29
Chapters 24, 30 and 63
Chapter 28
Chapters 28 and 46
Chapter 4
Chapter 14
Chapter 5
Chapters 11 and 20
Chapters 6 and 63
Chapter 1
Chapter 4
Chapter 28
Chapter 10
Chapters 6 and 12
Chapter 30
Chapters 6, 28, 32, 37, 39 and 43
Chapter 20
Chapters 1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15, 22, 23, 25, 27, 28, 30, 31, 33, 34,
35, 36, 37, 39, 40, 41, 43, 44, 46, 47, 48, 49, 55 and 57
Chapter 63
Chapter 40
Chapters 41 and 45
Chapter 28
Chapter 23
Chapters 6, 10, 28, 35 and 47
Chapter 24
Chapter 63
Chapter 1
Chapters 1, 2, 4, 6, 7, 9, 11, 13, 14, 17, 19, 20, 21, 22, 28, 30, 32, 33, 39, 42, 44,
45, 46, 49, 50, 53, 59, 61, 62 and 63
Chapter 44
Chapters 11, 21, 33, 37 and 56
Chapter 1
Chapters 1 and 46
Chapter 30
Chapters 15, 33 and 37
Chapter 28
Chapter 13
Chapter 4
Chapter 3
Chapters 8 and 18
Chapter 7
Chapter 1
Chapters 4, 10, 15, 16, 22, 43 and 52
Chapter 22
Chapter 21
Chapters 1, 3, 4, 10, 24, 25, 28, 29, 30, 31 and 33
Chapter 1
Chapters 10, 17 and 45
Chapter 63

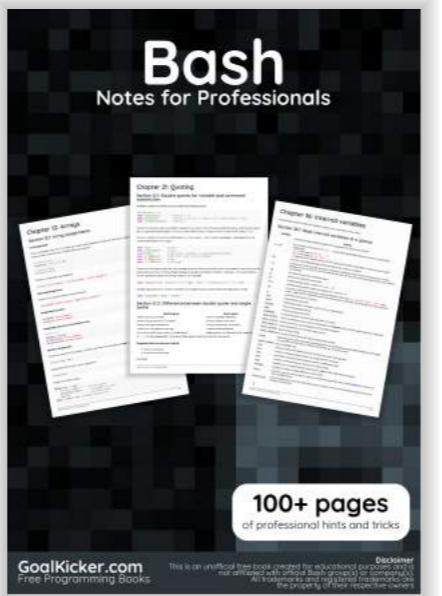
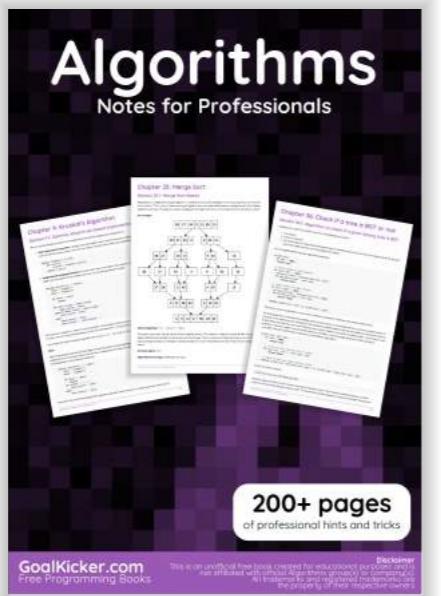
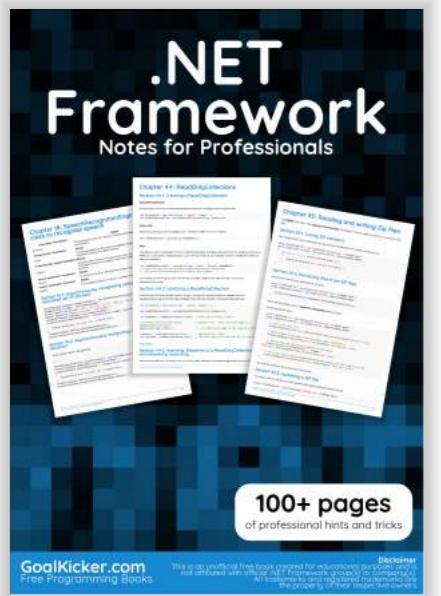
路易斯·贝蒂
伦丁
madD7
马杜苏丹·P
马尔科姆·麦克莱恩
马利克
马纳夫·M
曼塔尔
马克·伊斯里
马丁
马修
MC93
mhk
迈克尔·菲茨帕特里克
MikeCAT
mirabilos
mpromonet
内曼贾·博里奇
NeoR
Neui
尼廷库马尔·安贝卡尔
尼特耶什·阿加瓦尔
noamgot
OiciTrap
OznOg
P.P.
pandaman1234
帕尔哈姆·阿尔瓦尼
PassionInfinite
保罗·哈钦森
保罗·V
Paul92
彼得
光度立体
polarysekt
PSN
Purag
Qrchack
R. Joiny
Rakitić
拉面厨师
Ray
reshad
理查德·钱伯斯
里希凯什·拉杰
罗伯特·鲍尔迪加
罗兰·伊利格
rxantos
瑞安·海宁
瑞安·希尔伯特
Shog9
施里尼瓦斯·帕特加尔
舒布哈姆·阿格拉瓦尔

第46章
第40章
第9、41、43和46章
第30章
第6、15、19、20、22、24、26、29、33、46和49章
第1章
第14章
第6章
第1、28和46章
第28章
第46章
第1章和第37章
第10章
第22章
第4、6、16、22、25、28和63章
第31章和第32章
第35章
第28章
第16章
第43章和第46章
第22章、第30章和第43章
第62章
第4章和第16章
第1章
第10章、第12章和第16章
第1章、第4章、第6章、第16章、第17章、第22章、第28章、第43章、第46章和第57章
第10章、第16章、第49章和第60章
第58章
第53章
第9章
第46章
第4章和第7章
第4章和第28章
第13章和第42章
第21章
第1章
第6章
第37章
第37章
第1章
第56章
第10章
第15章和第20章
第25章和第30章
第9章
第30章
第51章和第63章
第63章
第10章和第33章
第1章
第19章
第26章
第62章

Luiz Berti
Lundin
madD7
Madhusoodan P
Malcolm McLean
Malick
manav m
mantal
Mark Yisri
Martin
Matthieu
MC93
mhk
Michael Fitzpatrick
MikeCAT
mirabilos
mpromonet
Nemanja Boric
NeoR
Neui
Nitinkumar Ambekar
Nityesh Agarwal
noamgot
OiciTrap
OznOg
P.P.
pandaman1234
Parham Alvani
PassionInfinite
Paul Hutchinson
Paul V
Paul92
Peter
PhotometricStereo
polarysekt
PSN
Purag
Qrchack
R. Joiny
Rakitić
RamenChef
Ray
reshad
Richard Chambers
Rishikesh Raje
Robert Baldyga
Roland Illig
rxantos
Ryan Haining
Ryan Hilbert
Shog9
Shrinivas Patgar
Shubham Agrawal
Chapter 46
Chapter 40
Chapters 9, 41, 43 and 46
Chapter 30
Chapters 6, 15, 19, 20, 22, 24, 26, 29, 33, 46 and 49
Chapter 1
Chapter 14
Chapter 6
Chapters 1, 28 and 46
Chapter 28
Chapter 46
Chapters 1 and 37
Chapter 10
Chapter 22
Chapters 4, 6, 16, 22, 25, 28 and 63
Chapters 31 and 32
Chapter 35
Chapter 28
Chapter 16
Chapters 43 and 46
Chapters 22, 30 and 43
Chapter 62
Chapters 4 and 16
Chapter 1
Chapters 10, 12 and 16
Chapters 1, 4, 6, 16, 17, 22, 28, 43, 46 and 57
Chapters 10, 16, 49 and 60
Chapter 58
Chapter 53
Chapter 9
Chapter 46
Chapters 4 and 7
Chapters 4 and 28
Chapters 13 and 42
Chapter 21
Chapter 1
Chapter 6
Chapter 37
Chapter 37
Chapter 1
Chapter 56
Chapter 10
Chapters 15 and 20
Chapters 25 and 30
Chapter 9
Chapter 30
Chapters 51 and 63
Chapter 63
Chapters 10 and 33
Chapter 1
Chapter 19
Chapter 26
Chapter 62

信号	第22章	signal	Chapter 22
西尔西里什·科达利	第52章	Sirsireesh Kodali	Chapter 52
skrtbhtngr	第1章	skrtbhtngr	Chapter 1
slugonamission	第22章	slugonamission	Chapter 22
斯奈普	第20章	Snaipe	Chapter 20
孙良	第1章	sohnryang	Chapter 1
someoneigna	第22章和第25章	someoneigna	Chapters 22 and 25
苏拉夫·高什	第49章	Sourav Ghosh	Chapter 49
蜘蛛侠	第22章	Spidey	Chapter 22
斯里卡尔	第46章	Srikar	Chapter 46
stackptr	第1、4、6、10、20、22、24、27、30、33、46和63章	stackptr	Chapters 1, 4, 6, 10, 20, 22, 24, 27, 30, 33, 46 and 63
星尘悟吉塔	第21章	StardustGogeta	Chapter 21
仍在学习	第6章	still_learning	Chapter 6
孙清尧	第34章	Sun Qingyao	Chapter 34
syb0rg	第1、6、19、20、22、37、46和49章	syb0rg	Chapters 1, 6, 19, 20, 22, 37, 46 and 49
塔马罗斯	第63章	Tamarous	Chapter 63
tbodt	第46章	tbodt	Chapter 46
techEmbedded	第63章	techEmbedded	Chapter 63
苏达卡尔	第46章	the sudhakar	Chapter 46
thndrwrks	第22章	thndrwrks	Chapter 22
tilzOR	第45章	tilzOR	Chapter 45
蒂姆·波斯特	第33章	Tim Post	Chapter 33
克林贡语	第1章	tlhIngan	Chapter 1
托比	第1、2、4、5、6、7、9、10、12、13、14、15、16、20、22、28、29、32、37、42、46、49、51、52、53、56、58、59、60和63章	Toby	Chapters 1, 2, 4, 5, 6, 7, 9, 10, 12, 13, 14, 15, 16, 20, 22, 28, 29, 32, 37, 42, 46, 49, 51, 52, 53, 56, 58, 59, 60 and 63
托弗罗	第16、37和46章	tofro	Chapters 16, 37 and 46
海龟	第37章	Turtle	Chapter 37
特弗斯蒂格	第20、28和41章	tversteeg	Chapters 20, 28 and 41
用户45891	第28章	user45891	Chapter 28
用户5389107	第4、5和10章	user5389107	Chapters 4, 5 and 10
v7d8dpo4	第30章	v7d8dpo4	Chapter 30
Vality	第13、22、54和56章	Vality	Chapters 13, 22, 54 and 56
vasili111	第1章	vasili111	Chapter 1
Vin	第1章	Vin	Chapter 1
Vivek S	第46章	Vivek S	Chapter 46
Vraj Pandya	第1章和第37章	Vraj Pandya	Chapters 1 and 37
vuko_zrno	第46和60章	vuko_zrno	Chapters 46 and 60
威廉·珀塞尔	第20章	William Pursell	Chapter 20
沃尔夫	第4章和第6章	Wolf	Chapters 4 and 6
伍德罗·巴洛	第19章	Woodrow Barlow	Chapter 19
巫师	第46章	Wyzard	Chapter 46
约塔姆·萨尔蒙	第19章	Yotam Salmon	Chapter 19
阿列克谢·涅乌达钦	第6章、第22章和第37章	Алексей Неудачин	Chapters 6, 22 and 37

你可能也喜欢



You may also like

