C Tutorial

https://www.sololearn.com/Play/C

整理：林庆泓

2018/11/12

linqinghong@email.szu.edu.cn

1.基本概念.....4
1.1.什么是 C.....4
1.2.Hello World！.....4
1.3.数据类型.....5
1.4.输入和输出.....8
1.5.注释.....13
1.6.运算符.....14
2.条件和循环.....18
2.1.条件.....18
2.2.嵌套 if 语句.....20
2.3.switch 语句.....21
2.4.逻辑运算符.....23
2.5.while 循环.....24
2.6.for 循环.....26
3.函数，数组和指针.....28
3.1.函数.....28
3.2 递归函数.....32
3.3.数组.....33
3.4.二维数组.....35
3.5.指针.....36
3.6.更多关于指针.....39
3.7.函数和数组.....41
4.字符串和函数指针.....43
4.1.字符串.....43
4.2.字符串函数.....45
4.3.函数指针.....48
4.4.void 指针.....51
5.结构体和联合体.....54
5.1.结构体.....54
5.2.使用结构体.....56
5.3.联合体.....59
5.4.使用联合体.....62
6.内存管理.....64
6.1.使用内存.....64
6.2.malloc 函数.....65
6.3.calloc 和 realloc 函数.....66
6.4.动态字符串和数组.....67
7.文件和异常处理.....68
7.1.使用文件.....68
7.2.二进制文件 I / O.....71
7.3.异常处理.....74
7.4.使用 Error 代码.....75
8.预处理器.....78
8.1.预处理器指令.....78

8.2.条件编译指令......80
8.3.预处理运算符......82

1.基本概念

1.1.什么是 C

✧ 介绍 C

• **C** 是一种通用编程语言，已有近 50 年的历史。
• **C** 已被用于编写从操作系统（包括 Windows 和许多其他系统）到复杂程序（如 Python 解释器，Git，Oracle 数据库等）的所有内容。
• C 的多功能性是设计的。它是一种低级语言，与机器的工作方式密切相关，同时仍然易于学习。
了解计算机内存的工作原理是 C 编程语言的一个重要方面。

1.2.Hello World！

✧ Hello World

• 在学习任何新语言时，总是从经典的"Hello World!"程序开始：

```
#include <stdio.h>

int main() {
   printf("Hello, World!\n");
   return 0;
}
```

• 让我们分解代码来理解每一行：
• **#include <stdio.h>**用于生成输出的函数在 stdio.h 中定义。为了使用 **printf** 函数，我们需要首先包含所需的文件，也称为**头文件**。

•**int main()**：**main()**函数是程序的入口点。花括号 {}表示函数的开头和结尾（也称为代码块）。括号内的语句确定执行函数的功能。

✧ Hello World

• **printf** 函数用于生成输出：

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

```
int main() {
   printf("Hello, World!\n");
   return 0;
}
```

• 在这里，我们传递“Hello World!”文本给它。
• \ n **转义字符**输出换行符。转义字符始终以反斜杠\开头。
• **分号**；表示声明的结束。每个语句必须以分号结尾。

• **return 0;** 此语句终止 **main()**函数并将值 0 返回给调用进程。数字 0 通常表示我们的程序已成功执行。任何其他数字表示程序已失败。

1.3.数据类型

✧ 数据类型

• C 支持以下基本数据类型：
int：整形，整数。
float：浮点数，带小数部分的数字。
double：双精度浮点值。
char：单个字符。

• 每种类型所需的存储量因平台而异。
• C 有一个内置的 **sizeof** 运算符，它为特定的数据类型提供内存需求。
例如：

```
#include <stdio.h>

int main() {
   printf("int: %d \n", sizeof(int));
   printf("float: %d \n", sizeof(float));
   printf("double: %d \n", sizeof(double));
   printf("char: %d \n", sizeof(char));

   return 0;
}
```

• 程序输出显示每种数据类型的相应大小（以字节为单位）。
• 该程序中的 **printf** 语句有两个**参数**。第一个是带有**格式说明符**(%d)的输出字符串，而下一个参数返回 **sizeof** 值。在最终输出中，**%d**（对于十进制）将替换为第二个参数中的值。

请注意，C 没有布尔类型。
printf 语句可以有多个格式说明符，并带有相应的参数来替换说明符。格式说明符也称为转换说明符。
我们将在即将到来的课程中详细了解格式说明符。

✧ 变量

• 变量是内存中区域的名称。
• 变量的名称（也称为**标识符**）必须以字母或下划线开头，并且可以由字母，数字和下划线字符组成。
• 变量命名约定不同，但是使用带有下划线的小写字母来分隔单词很常见（snake_case）。
• 在使用变量之前，还必须为变量声明数据类型。

• 声明变量的值使用**赋值语句**更改。
• 例如，以下语句声明一个整数变量 **my_var**，然后为其赋值 42：

```
int my_var;
my_var = 42;
```

• 您还可以在单个语句中声明和**初始化**（分配初始值）变量：

```
int my_var = 42;
```

• 让我们定义不同类型的变量，做一个简单的数学运算，然后输出结果：

```
#include <stdio.h>

int main() {
   int a, b;
   float salary = 56.23;
   char letter = 'Z';
   a = 8;
   b = 34;
   int c = a+b;

   printf("%d \n", c);
   printf("%f \n", salary);
   printf("%c \n", letter);

   return 0;
}
```

•如您所见，您可以通过用**逗号**分隔多个变量来在单行上声明它们。另外，请注意使用 float(%f)和 char(%c)输出的格式说明符。
C 编程语言**区分大小写**，因此 **my_Variable** 和 **my_variable** 是两个不同的标识符。

✧ 常量

• **常量**存储一个初始赋值后无法更改的值。
• 通过使用具有有意义名称的常量，代码更易于阅读和理解。
• 为了区分常量和变量，通常的做法是使用大写标识符。

• 定义常量的一种方法是在变量声明中使用 **const** 关键字：

```
#include <stdio.h>

int main() {
  const double PI = 3.14;
  printf("%f", PI);

  return 0;
}
```

• 在程序执行期间，不能更改 PI 的值。
• 例如，另一个赋值语句（如 PI = 3.141）将生成错误。
必须在声明时使用值初始化常量。

• 定义常量的另一种方法是使用**#define** 预处理器指令。
#define 指令使用宏来定义常量值。
例如：

```
#include <stdio.h>

#define PI 3.14

int main() {
  printf("%f", PI);
  return 0;
}
```

• 在编译之前，预处理器将代码中的每个宏标识符替换为指令中的相应值。在这种情况下，每次出现的 **PI** 都将替换为 **3.14**。
• 发送给编译器的最终代码已经具有常量值。

• **const** 和**#define** 之间的区别在于前者使用内存进行存储而后者不使用内存。
不要在**#define** 语句的末尾加上分号。这是一个常见的错误。
我们将在下一个模块中了解有关**预处理器指令**的更多信息。

1.4.输入和输出

✧ 输入

• C 支持多种用户输入方式。
• **getchar()**返回下一个单个字符输入的值。

例如：

```
#include <stdio.h>

int main() {
   char a = getchar();

   printf("You entered: %c", a);

   return 0;
}
```

• 输入存储在变量 **a** 中。

• **gets**()函数用于将输入读取为有序的字符序列，也称为**字符串**。
• 字符串存储在 char 数组中。

例如：

```
#include <stdio.h>

int main() {
   char a[100];

   gets(a);

   printf("You entered: %s", a);

   return 0;
}
```

• 这里我们将输入存储在一个包含 100 个字符的数组中。

• **scanf()**扫描与格式说明符匹配的输入。

例如：

```
#include <stdio.h>

int main() {
```

```
  int a;
  scanf("%d", &a);

  printf("You entered: %d", a);

  return 0;
}
```

• 变量名称前面的**&**符号是**地址运算符**。它给出变量的地址或内存位置。这是必需的，因为 **scanf** 将输入值放在变量地址处。

• 再举一个例子，让我们提示两个整数输入并输出它们的总和：

```
#include <stdio.h>

int main() {
  int a, b;
  printf("Enter two numbers:");
  scanf("%d %d", &a, &b);

  printf("\nSum: %d", a+b);

  return 0;
}
```

scanf()在遇到空格时立即停止读取，因此诸如“Hello World”之类的文本是 **scanf()**的两个独立输入。

✧ 输出

• 我们已经使用 **printf()**函数在前面的课程中生成输出。在本课程中，我们将介绍可用于输出的其他几个功能。

putchar()输出单个字符。

例如：

```
#include <stdio.h>

int main() {
  char a = getchar();

  printf("You entered: ");
  putchar(a);

  return 0;
```

```
}
```

• 输入存储在变量 **a** 中。

• **puts()**函数用于将输出显示为字符串。
• 字符串存储在 char 数组中。
例如：

```
#include <stdio.h>

int main() {
   char a[100];

   gets(a);

   printf("You entered: ");
   puts(a);

   return 0;
}
```

• 这里我们将输入存储在一个包含 100 个字符的数组中。

• **scanf()**扫描与格式说明符匹配的输入。
例如：

```
#include <stdio.h>

int main() {
   int a;
   scanf("%d", &a);

   printf("You entered: %d", a);

   return 0;
}
```

• 变量名称前面的**&**符号是**地址运算符**。它给出变量的地址或内存位置。这是必需的，因为 scanf 将输入值放在变量地址处

• 再举一个例子，让我们提示两个整数输入并输出它们的总和：

```
#include <stdio.h>

int main() {
   int a, b;
   printf("Enter two numbers:");
```

```
   scanf("%d %d", &a, &b);

   printf("\nSum: %d", a+b);

   return 0;
}
```

scanf()在遇到空格时立即停止读取，因此诸如“Hello World”之类的文本是 **scanf()**的两个独立输入。

✧ 格式化输入

•**scanf()**函数用于为变量分配输入。对此函数的调用将根据转换输入的格式说明符扫描输入。
• 如果无法转换输入，则不进行分配。
• **scanf()**语句等待输入，然后进行赋值：

```
int x;
float num;
char text[20];
scanf("%d %f %s", &x, &num, text);
```

• 输入 10 22.5 abcd 然后按 Enter 键将 10 分配给 x，将 22.5 分配给 num，将 abcd 分配给文本。
• 请注意，必须使用**&**来访问变量地址。字符串不需要**&**，因为字符串名称充当指针。

• 格式说明符以百分号**%**开头，用于在控制字符串后为相应的参数赋值。**空格、制表符和换行符将被忽略。**
• 格式说明符可以包含多个选项以及转换字符：
%[*] [max_field]conversion character
• 可选*将跳过输入字段。
• 可选的 max_width 给出了要分配给输入字段的最大字符数。
• 如有必要，转换字符会将参数转换为指示的类型：
d 十进制
c 字符
s 字符串
f 浮点型
x 十六进制

例如：

```
int x, y;
char text[20];

scanf("%2d %d %*f %5s", &x, &y, text);
```

```
/* input: 1234    5.7    elephant */
printf("%d    %d    %s", x, y, text);
/* output: 12    34    eleph */
```

✧ 格式化输出

• **printf** 函数是在您的第一个 Hello World 程序中引入的。对此函数的调用需要一个**格式字符串**，该字符串可以包含用于输出特殊字符的转义字符和由值替换的格式说明符。
例如：

```
printf("The tree has %d apples.\n", 22);
/* The tree has 22 apples. */

printf("\"Hello World!\"\n");
/* "Hello World!" */
```

• 转义字符以**反斜杠**开头：
\n 新行
\t 水平标签
**** 反斜杠
\b 退格
\' 单引号
\" 双引号

• 格式说明符以**百分号%**开头，并在格式字符串后用相应的参数替换。格式说明符可以包含多个选项以及转换字符：

```
%[-][width].[precision]conversion character
```

• 可选 **-** 指定字符串中数据的左对齐。
• 可选 **width** 为数据提供最小字符数。
• **period .**将**宽度**与**精度**分开。
• 可选 **precision** 给出了数值数据的小数位数。如果 **s** 用作转换字符，则 precision 将确定要打印的字符数。
• 如有必要，转换字符会将参数转换为指示的类型：
d 十进制
c 字符
s 字符串
f 浮点型
e 科学记数法
x 十六进制

例如：

```
printf("Color: %s, Number: %d, float: %5.2f \n", "red", 42, 3.14159);
/* Color: red, Number: 42, float:    3.14    */

printf("Pi = %3.2f", 3.14159);
/* Pi = 3.14 */

printf("Pi = %8.5f", 3.14159);
/* Pi =      3.14159 */

printf("Pi = %-8.5f", 3.14159);
/* Pi = 3.14159 */

printf("There are %d %s in the tree.", 22, "apples");
/* There are 22 apples in the tree. */
```

• 要打印%符号，请在格式字符串中使用%%。

1.5.注释

✧ 注释

• 注释是解释性信息，您可以将其包含在程序中以使代码的读者受益。编译器忽略注释，因此它们对程序没有影响。
• 注释以斜杠星号**/***开头，以星号斜杠***/**结尾，可以在代码中的任何位置。
• 注释可以与语句位于同一行，也可以跨越多行。
例如：

```
#include <stdio.h>

/* A simple C program
 *    Version 1.0
 */
int main() {
   /* Output a string */
   printf("Hello World!");
   return 0;
}
```

如您所见，评论向读者阐明了程序的意义。使用注释来阐明代码段背后的目的和逻辑。

✧ 单行注释

• C ++引入了一个双斜杠注释//作为注释单行的方法。一些 C 编译器也支持这种评论风格。
例如：

```
#include <stdio.h>

int main() {
   int x = 42; //int for a whole number

   //%d is replaced by x
   printf("%d", x);

   return 0;
}
```

在代码中添加注释是很好的编程习惯。它有助于您和其他阅读它的人清楚地理解代码。

1.6.运算符

✧ 算术运算符

• C 支持算术**运算符+**（加法）、-（减法）、*（乘法）、/（除法）和%（模数除法）。
• 运算符通常用于形成**数值表达式**，例如 10 + 5，在这种情况下，它包含两个**操作数**和加法**运算符**。
• 数值表达式通常用于赋值语句。
例如：

```
#include <stdio.h>

int main() {
   int length = 10;
   int width = 5;
   int area;

   area = length * width;
   printf("%d \n", area);    /* 50 */

   return 0;
}
```

除法

• C 有两个除法运算符：/和%。

• 除法/运算符根据操作数的数据类型执行不同的操作。当两个操作数都是 int 数据类型时，**整数除法**（也称为**截断除法**）会删除任何余数以产生整数。当一个或两个操作数是实数（浮点数或双精度数）时，结果是实数。

• %运算符仅返回整数除法的余数。它对许多算法都很有用，包括从数字中检索数字。**模数除法**不能在浮点数或双精度数上执行。

• 以下示例演示了除法：

```
#include <stdio.h>

int main() {
   int i1 = 10;
   int i2 = 3;
   int quotient, remainder;
   float f1 = 4.2;
   float f2 = 2.5;
   float result;

   quotient = i1 / i2; // 3
   remainder = i1 % i2; // 1
   result = f1 / f2; // 1.68

   return 0;
}
```

✧ 运算符优先级

• C 根据**运算符优先级**计算数值表达式。

• +和-的优先级相同，*、/和%也是如此。

• *、/和%首先按从左到右依次执行，然后按+和-依次从左到右依次执行。

• 您可以使用括号()更改操作顺序，以指示首先执行哪些操作。

• 例如，5 + 3 * 2 的结果是 11，其中(5 + 3)*2 的结果是 16。

例如：

```
#include <stdio.h>

int main() {
   int a = 6;
   int b = 4;
   int c = 2;
   int result;
   result = a - b + c; // 4
```

```
    result = a + b / c; // 8
    result = (a + b) / c; // 5

    return 0;
}
```

当关联属性允许任何顺序时，C 可能不会根据需要计算数值表达式。例如，(x * y)* z 可以被运算为 x *(y * z)。如果顺序很重要，请将表达式分解为单独的语句。

✧ 类型转换

• 当数值表达式包含不同数据类型的操作数时，它们会在称为**类型转换**的过程中根据需要自动转换。
• 例如，在涉及浮点数和整数的操作中，编译器会将 int 值转换为浮点值。
在以下程序中，**increase** 变量自动转换为 **float**：

```
#include <stdio.h>

int main() {
    float price = 6.50;
    int increase = 2;
    float new_price;

    new_price = price + increase;
    printf("New price is %4.2f", new_price);
    /* Output: New price is 8.50 */

    return 0;
}
```

• 请注意，格式说明符包括 4.2，表示浮点数将打印在一个至少 4 个字符宽且 2 位小数的空间中。
• 如果要将表达式的结果强制为其他类型，可以通过类型转换执行显式类型转换，如以下语句中所示：

```
float average;
int total = 23;
int count = 4;

average = (float) total / count;
/* average = 5.75 */
```

• 如果没有类型转换，则将分配平均值 5。
• 显式类型转换，即使编译器可能进行自动类型转换，也被认为是良好的编程风格。

✧ 赋值运算符

• 赋值语句首先计算等号右侧的表达式，然后将该值赋给=左侧的变量。这使得可以在赋值语句的两侧使用相同的变量，这通常在编程中完成。

例如：

```
int x = 3;
x = x + 1;    /* x is now 4 */
```

• 为了缩短这种类型的赋值语句，C 提供了**+=赋值运算符**。上面的陈述可以写成

```
x += 1;    /* x = x + 1 */
```

• 许多 C 运算符都有相应的赋值运算符。下面的程序演示了算术赋值运算符：

```
int x = 2;

x += 1;    // 3
x -= 1;    // 2
x *= 3;    // 6
x /= 2;    // 3
x %= 2;    // 1
x += 3 * 2;    // 7
```

• 仔细查看最后一个赋值语句。运算右侧的整个表达式，然后在分配给 x 之前将其添加到 x。您可以将语句视为 x = x +(3 * 2)。

✧ 自增和自减

• 可以使用**自增运算符++**将变量 1 添加到变量中。类似地，**自减运算符--** 用于从变量中减去 1。

例如：

```
z--;    /* decrement z by 1 */
y++; /* increment y by 1 */
```

• 递增和递减运算符可以使用**前缀**（在变量名之前）或**后缀**（在变量名之后）。在赋值语句中使用运算符的方式很重要，如下例所示。

```
z = 3;
x = z--;    /* assign 3 to x, then decrement z to 2 */
y = 3;
x = ++y;    /* increment y to 4, then assign 4 to x */
```

• **前缀**形式先自增/自减变量，然后在赋值语句中使用它。
• **后缀**形式在自增/自减之前首先使用变量的值。

2.条件和循环

2.1.条件

✧ 条件

• 条件用于执行不同的计算或操作，具体取决于条件是评估为真还是假。

if 语句

• **if** 语句称为**条件控制结构**，因为它在表达式为 true 时执行语句。因此，**if** 也称为**判断结构**。它采取以下形式：

```
if (expression)
   statements
```

• 表达式的计算结果为 **true** 或 **false**，语句可以是单个语句或由花括号{ }括起的代码块。

例如：

```
#include <stdio.h>

int main() {
   int score = 89;

   if (score > 75)
      printf("You passed.\n");

   return 0;
}
```

• 在上面的代码中，我们检查 **score** 变量是否大于 75，如果条件为真，则打印一条消息。

✧ 关系运算符

• 有六个**关系运算符**可用于形成布尔表达式，返回 **true** 或 **false**：

< 小于
<= 小于或等于
> 大于
>= 大于或等于
== 等于
!= 不等于

例如：

```
int num = 41;
num += 1;
if (num == 42) {
    printf("You won!");
}
```

• 计算结果为非零值的表达式被视为 true。

例如：

```
int in_stock = 20;
if (in_stock)
   printf("Order received.\n");
```

✧ if-else 语句

• **if** 语句可以包含一个可选的 **else** 子句，该子句在表达式为 **false** 时执行语句。
• 例如，以下程序计算表达式，然后执行 **else** 子句语句：

```
#include <stdio.h>

int main() {
   int score = 89;

   if (score >= 90)
      printf("Top 10%%.\n");
   else
      printf("Less than 90.\n");

   return 0;
}
```

✧ 条件表达式

形成 if-else 语句的另一种方法是在**条件表达式**中使用?:运算符。?:运算符只能有一个与 **if** 和 **else** 关联的语句。

例如：

```
#include <stdio.h>

int main() {
   int y;
```

```
   int x = 3;

   y = (x >= 5) ?   5 : x;

/* This is equivalent to:
   if (x >= 5)
      y = 5;
   else
      y = x;
*/

   return 0;
}
```

2.2.嵌套 if 语句

✧ 嵌套 if 语句

• **if** 语句可以包含另一个 **if** 语句来形成嵌套语句。嵌套 **if** 允许基于进一步的条件做出决定。请思考以下语句：

```
if (profit > 1000)
   if (clients > 15)
      bonus = 100;
   else
bonus = 25;
```

• 适当缩进嵌套语句将有助于向读者阐明其含义。但是，请务必了解 **else** 子句与最近的 **if** 语句关联，除非使用花括号{}来更改关联。

例如：

```
if (profit > 1000) {
   if (clients > 15)
      bonus = 100;
}
else
   bonus = 25;
```

✧ if-else if 语句

• 当需要在三个或更多个动作中做出决定时，可以使用 **if-else if** 语句。
可以有多个 **else if** 子句，最后一个 **else** 子句是可选的。
例如：

```
int score = 89;

if (score >= 90)
   printf("%s", "Top 10%\n");
else if (score >= 80)
   printf("%s", "Top 20%\n");
else if (score > 75)
   printf("%s", "You passed.\n");
else
   printf("%s", "You did not pass.\n");
```

• 在采用 **if-else if** 语句时要仔细考虑所涉及的逻辑。当程序流进行到与第一个表达式关联的语句，并不会测试任何剩余的表达式。
• 虽然缩进不会影响编译代码，但是当 **else** 子句对齐时，读者更容易理解 **if-else if** 的逻辑。
在可能的情况下，为了清楚起见，**if-else if** 语句优先于嵌套 if 语句。

2.3.switch 语句

✧ switch 语句

• **switch** 语句通过将表达式的结果与常量 **case** 值匹配来分支控制程序。

• **switch** 语句通常为 **if-else if** 和**嵌套 if** 语句提供更优雅的解决方案。
该开关采用以下形式：

```
switch (expression) {
   case val1:
      statements
   break;
   case val2:
      statements
   break;
   default:
      statements
}
```

• 例如，以下程序输出“Three”：

```
int num = 3;

   switch (num) {
   case 1:
      printf("One\n");
      break;
   case 2:
      printf("Two\n");
      break;
   case 3:
      printf("Three\n");
      break;
   default:
      printf("Not 1, 2, or 3.\n");
}
```

✧ switch 语句

• 可以有多个唯一的 **cases** 值。
• 当没有其他匹配时，执行可选的 **default** 情况。
• 在每种情况下都需要 **break** 语句来分支到 **switch** 语句的末尾。
• 如果没有 **break** 语句，程序执行将进入下一个 **case** 语句。当几种情况需要相同的语句时，这可能很有用。请考虑以下 **switch** 语句：

```
switch (num) {
   case 1:
   case 2:
   case 3:
      printf("One, Two, or Three.\n");
      break;
   case 4:
   case 5:
   case 6:
      printf("Four, Five, or Six.\n");
      break;
   default:
      printf("Greater than Six.\n");
}
```

• 以这种方式构造 **switch** 时必须小心。稍后修改可能会导致意外结果。

2.4.逻辑运算符

✧ &&运算符

• 逻辑运算符**&&**和**||**用于形成复合布尔表达式从而测试多条件。第三个逻辑运算符是**!**用于反转布尔表达式的状态。

&&运算符

• 仅当两个表达式都为真时，逻辑 AND 运算符**&&**才返回 true 结果。

例如：

```
if (n > 0 && n <= 100)
   printf("Range (1 - 100).\n");
```

• 上面的语句只加入两个表达式，但逻辑运算符可用于连接多个表达式。

• **从左到右**计算复合布尔表达式。当不需要进一步测试来确定结果时，评估停止，因此当一个结果影响后面结果的结果时，请务必考虑操作数的排列。

✧ ||运算符

• 逻辑 OR 运算符**||**当任何一个表达式或两个表达式都为真时返回 true 结果。

例如：

```
if (n == 'x' || n == 'X')
   printf("Roman numeral value 10.\n");
```

可以通过&&和||连接任意数量的表达式。

例如：

```
if (n == 999 || (n > 0 && n <= 100))
   printf("Input valid.\n");
```

括号用于清晰逻辑，因此&&的优先级高于||并将优先计算。

✧ ||运算符

• 逻辑 NOT 运算符**!**返回其值的反转。

• NOT true 返回 false，NOT false 返回 true。

例如：

```
if (!(n == 'x' || n == 'X'))
   printf("Roman numeral is not 10.\n");
```

在 C 中，任何非零值都被认为是**真**，0 是**假**。因此，逻辑 NOT 运算符将 true 值转换为 0，将 false 值转换为 1。

2.5.while 循环

✧ while 循环

• **while** 语句称为**循环结构**，因为它在表达式为 true 时重复执行语句，一遍又一遍地循环。它采取以下形式：

```
while (expression) {
   statements
}
```

• 表达式的计算结果为 **true** 或 **false**，语句可以是单个语句，或者更常见的是由大括号{}括起来的代码块。

例如：

```
#include <stdio.h>

int main() {
   int count = 1;

   while (count < 8) {
      printf("Count = %d\n", count);
      count++;
   }

   return 0;
}
```

• 上面的代码将输出 **count** 变量 7 次。

• while 循环在输入循环之前计算条件，可能使得 while 语句永远不会执行。

无限循环是一个无限循环的循环，因为循环条件永远不会计算为 false。这可能会导致运行时错误。

✧ do-while 循环

• **do-while** 循环在计算表达式之前执行循环语句，以确定是否应该重复循环。
它采取以下形式：

```
do {
   statements
} while (expression);
```

• 表达式的计算结果为 true 或 false，语句可以是单个语句或由大括号{}括起的代码块。
例如：

```
#include <stdio.h>

int main() {
   int count = 1;

   do {
      printf("Count = %d\n", count);
      count++;
   } while (count < 8);

   return 0;
}
```

• 请注意 while 语句后面的分号。
• 即使表达式判断为 false，**do-while** 循环也至少执行一次。

✧ break-continue

• 在 **switch** 语句中引入 **break** 语句并使用它。它对于立即退出循环也很有用。
• 例如，以下程序使用 **break** 来退出 **while** 循环：

```
int num = 5;

while (num > 0) {
   if (num == 3)
      break;
   printf("%d\n", num);
   num--;
}
```

该程序显示：

```
5
```

```
4
```
然后退出循环。

• 如果要保持循环，但跳到下一次迭代，则使用 **continue** 语句。
例如：

```
int num = 5;

while (num > 0) {
   num--;
   if (num == 3)
      continue;

   printf("%d\n", num);
}
```

程序输出显示：

```
4
2
1
0
```

如您所见，3 被跳过。

• 在上面的代码中，如果 **num** 在 **continue** 语句之后递增，则会创建一个无限循环。
虽然 **break** 和 **continue** 语句很方便，但它们不应该代替更好的算法。

2.6.for 循环

✧ break-continue

• **for** 语句是一个循环结构，它执行语句固定次数。
它采取以下形式：

```
for (initvalue; condition; increment) {
   statements;
}
```

• **initvalue** 是一个设置为初始值的计数器。**for** 循环的这一部分只执行一次。
• **condition** 是一个布尔表达式，它在每次迭代后将计数器和给定的值进行比较，并在返回 false 时停止循环。
• **increment** 将计数器增加（或减少）一个设定值。

例如，下面的程序显示 0 到 9：

```
int i;
int max = 10;

for (i = 0; i < max; i++) {
   printf("%d\n", i);
}
```

✧ break-continue

• **for** 循环可以包含多个用逗号分隔的表达式。
例如：

```
for (x = 0, y = num; x < y; i++, y--) {
   statements;
}
```

• 此外，您可以跳过**初始值**，**条件**和/或**增量**。
例如：

```
int i=0;
int max = 10;
for (; i < max; i++) {
   printf("%d\n", i);
}
```

• 循环也可以**嵌套**。
• 以这种方式编写程序时，有一个外循环和一个内循环。对于外循环的每次迭代，内循环重复其整个循环。
在以下示例中，嵌套 for 循环用于输出乘法表：

```
int i, j;
int table = 10;
int max = 12;

for (i = 1; i <= table; i++) {
   for (j = 0; j <= max; j++) {
      printf("%d x %d = %d\n", i, j, i*j);
   }
   printf("\n"); /* blank line between tables */
}
```

内循环中的 **break** 退出该层循环，并继续执行外循环。
continue 语句在嵌套循环中的工作方式类似。

3.函数，数组和指针

3.1.函数

✧ C 中的函数

• **函数**是 C 编程的核心，用于将程序解决方案作为一系列子任务来完成。
• 到目前为止，您知道每个 C 程序都包含 **main()**函数。而且你熟悉 **printf()**函数。

您还可以创建自己的函数。
函数：
•是执行特定任务的代码块
•可重复使用
•使程序更容易测试
•可以在不更改调用程序的情况下进行修改

• 当 **main()**被分解为用函数实现的子任务时，即使是简单的程序也更容易理解。
例如，很明显这个程序的目标是计算数字的平方：

```
int main() {
   int x, result;

   x = 5;
   result = square(x);
   printf("%d squared is %d\n", x, result);

   return 0;
}
```

• 为了使用 **square** 函数，我们需要声明它。
• 声明通常出现在 **main()**函数上方并采用以下形式：

```
return_type function_name(parameters);
```

• **return_type** 是函数发送回调用语句的值的类型。**function_name** 后跟括号。带有类型声明的可选**参数**名称放在括号内。
即使函数不需要返回值，返回类型仍必须在声明中。在这种情况下，使用关键字 void。
例如，**display_message** 函数声明指示函数不返回值：**void display_message()**;

✧ C 中的函数

• 当参数类型和名称包含在声明中时，声明称为**函数原型**。

• 例如，**square** 函数原型出现在 main()上方：

```
#include <stdio.h>

/* declaration */
int square (int num);

int main() {
   int x, result;

   x = 5;
   result = square(x);
   printf("%d squared is %d\n", x, result);

   return 0;
}
```

• 我们的 **square** 函数返回一个整数，并取一个 int 类型的参数。

• 最后一步实际上是**定义**函数。函数定义通常出现在 **main()**函数之后。
• 下面的完整程序显示了 **square** 函数声明和定义：

```
#include <stdio.h>

/* declaration */
int square (int num);

int main() {
   int x, result;

   x = 5;
   result = square(x);
   printf("%d squared is %d\n", x, result);

   return 0;
}

/* definition */
int square (int num) {
   int y;
```

```
   y = num * num;

   return(y);
}
```

• 如您所见，**square** 函数计算并返回其参数的平方。
• 函数可以使用多个参数 - 在这种情况下，它们必须用逗号分隔。
• **return** 语句用于将值发送回调用语句。

✧ 函数参数

• 函数的**参数**用于接收函数所需的值。值通过函数调用作为**参数**传递给这些参数。
• 默认情况下，参数按值传递，这意味着数据的副本将被赋予被调用函数的参数。实际变量未传递给函数，因此不会更改。
• 传递给函数的参数按位置与参数匹配。因此，第一个参数传递给第一个参数，第二个参数传递给第二个参数，依此类推。
以下程序演示了按值传递的参数：

```
#include <stdio.h>

int sum_up (int x, int y);

int main() {
   int x, y, result;

   x = 3;
   y = 12;
   result = sum_up(x, y);
   printf("%d + %d\ = %d\n", x, y, result);

   return 0;
}

int sum_up (int x, int y) {
   x += y;
   return(x);
}
```

• 程序输出为：3 + 12 = 15
• **x** 和 **y** 的值传递给 **sum_up**。请注意，即使参数 **x** 的值在 **sum_up** 中更改，main()中参数 x 的值也未更改，因为只有其值传递给参数 **x**。
函数声明中的参数是**形式参数**。传递给这些参数的值是 arguments，有时称为**实际参数**。

✧ 作用域

• **作用域**是指程序中变量的可见性。
• 函数中声明的变量是该代码块的**本地**变量，不能在函数外部引用。
• 在所有函数之外声明的变量对于整个程序是**全局**的。
例如，在程序顶部使用**#define** 声明的常量对整个程序可见。
以下程序使用**本地**和**全局**变量：

```
#include <stdio.h>

int global1 = 0;

int main() {
   int local1, local2;

   local1 = 5;
   local2 = 10;
   global1 = local1 + local2;
   printf("%d \n", global1);    /* 15 */

   return 0;
}
```

• 当参数传递给函数参数时，参数充当局部变量。退出函数时，函数中的参数和任何局部变量都将被销毁。
谨慎使用全局变量。应在使用前初始化它们以避免意外结果。由于它们可以在程序中的任何位置进行更改，因此全局变量可能导致难以检测到错误。

✧ 静态变量

• **静态**变量具有局部范围，但在退出函数时不会被销毁。因此，静态变量保留其程序生命周期的值，并且可以在每次重新输入函数时访问。
• 声明时初始化静态变量，并且需要前缀 **static**。
以下程序使用静态变量：

```
#include <stdio.h>

void say_hello();

int main() {
```

```
    int i;

    for (i = 0; i < 5; i++) {
        say_hello();
    }

    return 0;
}

void say_hello() {
    static int num_calls = 1;

    printf("Hello number %d\n", num_calls);
    num_calls++;

}
```

The program output is:

```
Hello number 1
Hello number 2
Hello number 3
Hello number 4
Hello number 5
```

3.2 递归函数

✧ 递归函数

• 可以使用称为**递归**的过程来最好地实现用于解决问题的算法。考虑一个数字的阶乘，通常写成 5!= 5 * 4 * 3 * 2 * 1。
• 该计算也可以被认为是重复计算 num *(num -1)直到 num 为 1。

• **递归函数**是一个自我调用函数，它包含一个基本情况或退出条件，用于结束递归调用。在计算阶乘的情况下，基本情况是 **num** 等于 1。
例如：

```
#include <stdio.h>

int factorial(int num);

int main() {
    int x = 5;
```

```
    printf("The factorial of %d is %d\n", x, factorial(x));

    return 0;
}

int factorial(int num) {

    if (num == 1)    /* base case */
        return (1);
    else
        return (num * factorial(num - 1));
}
```

程序输出为：The factorial of 5 is 120

• 递归通过“堆栈”调用来工作，直到执行基本案例。此时，呼叫从最新到最旧完成。阶乘调用栈可以被认为是：
2*factorial(1)
3*factorial(2)
4*factorial(3)
5*factorial(4)

•当达到基本情况时，返回值 1 触发堆栈调用的完成。从最新到最旧的返回值创建以下计算，最终计算(5 * 24)返回到调用函数 main()：
2 * 1
3 * 2
4 * 6
5 * 24

递归方法需要一个基本情况来防止无限循环。

3.3.数组

✧ C 中的数组

• **数组**是一种数据结构，用于存储所有相同类型的相关值的集合。
• 数组很有用，因为它们可以用一个描述性名称表示相关数据，而不是使用每个必须唯一命名的单独变量。
• 例如，数组 **test_scores [25]**可以容纳 25 个测试分数。
• 数组声明包括它存储的值的类型，标识符和带有表示数组大小的数字的方括号[]。
例如：

```
int test_scores[25]; /* An array size 25 */
```

• 您还可以在声明数组时初始化数组，如以下语句中所示：

```
float prices[5] = {3.2, 6.55, 10.49, 1.25, 0.99};
```

• 请注意，初始值由逗号分隔并放在花括号{}内。
• 可以部分初始化数组，如下所示：

```
float prices[5] = {3.2, 6.55};
```

• 缺失值设置为 0。
数组存储在连续的内存位置，并且在声明后不能更改大小。

✧ 访问数组元素

• 数组的内容称为**元素**，每个元素都可以通过索引号访问。
• 在 C 中，索引号从 **0** 开始。
• 具有 5 个元素的数组将具有索引号 0,1,2,3 和 4.考虑一个数组 x：

```
int x[5] = {20, 45, 16, 18, 22};
```

它可以被认为是：
0 => [20]
1 => [45]
2 => [16]
3 => [18]
4 => [22]

要访问数组元素，根据其索引号。
例如：

```
int x[5] = {20, 45, 16, 18, 22};
printf("The second element is %d\n", x[1]); /* 45 */
```

• 可以通过赋值语句更改数组元素的值，该语句还需要使用数组名称和索引：

```
int x[5] = {20, 45, 16, 18, 22};
x[1] = 260;
printf("The second element is %d\n", x[1]); /* 260 */
```

数组的索引也称为**下标**。

✧ 数组使用循环

• 许多算法需要访问数组的每个元素以检查数据，存储信息和其他任务。这可以在一个称为

遍历数组的过程中完成，该过程通常用 for 循环实现，因为循环控制变量自然对应于数组索引。
考虑以下程序：

```
float purchases[3] = {10.99, 14.25, 90.50};
float total = 0;
int k;

/* total the purchases */
for (k = 0; k < 3; k++) {
   total += purchases[k];
}

printf("Purchases total is %6.2f\n", total);
/* Output:    Purchases total is 115.74 */
```

- 循环控制变量从 0 到一小于待匹配索引值对应元素数的数。
- 循环对分配也很有用。

例如：

```
int a[10];
int k;

for (k = 0; k < 10; k++) {
   a[k] = k * 10;
}
```

3.4.二维数组

✧ 二维数组

- **二维数组**是一个数组数组，可以看作是一个表。您还可以将二维数组视为用于表示国际象棋棋盘，城市街区等网格。

- 二维数组声明表示数字行数和列数。

例如：

```
int a[2][3]; /* A 2 x 3 array */
```

- 嵌套花括号用于逐行初始化元素，如下面的语句所示：

```
int a[2][3] = {
   {3, 2, 6},
   {4, 5, 20}
};
```

• 同样的陈述也可以采取以下形式：

```
int a[2][3] = { {3, 2, 6}, {4, 5, 20} };
```

• 前一种语句提供了更清晰的可视化数组结构。

一个数组可以有两个以上的维度。例如，[5] [3] [4]是一个包含 5 个元素的数组，每个元素存储 3 个元素，每个元素存储 4 个元素。

✧ 访问二维数组

• 要访问二维数组的元素，需要行索引和列索引。

• 例如，以下语句显示元素的值，然后分配新值：

```
int a[2][3] = {
  {3, 2, 6},
  {4, 5, 20}
};
printf("Element 3 in row 2 is %d\n", a[1][2]); /* 20 */
a[1][2] = 25;
printf("Element 3 in row 2 is %d\n", a[1][2]); /* 25 */
```

• 正如 **for** 循环用于迭代一维数组一样，嵌套 **for** 循环用于遍历二维数组：

```
int a[2][3] = {
  {3, 2, 6},
  {4, 5, 20}
};
int k, j;
/* display array contents */
for (k = 0; k < 2; k++) {
  for (j = 0; j < 3; j++) {
    printf(" %d", a[k][j]);
  }
  printf("\n");
}
```

3.5.指针

✧ 使用内存

• C 被设计为低级语言，可以轻松访问内存位置并执行与内存相关的操作。

• 例如，**scanf()**函数将用户输入的值放在变量的位置或**地址**处。这是通过使用&符号来完成的。

例如：

```
int num;
printf("Enter a number: ");

scanf("%d", &num);

printf("%d", num);
```

&num 是变量 **num** 的地址。

• 存储器地址以**十六进制**数给出。**Hexadecimal** 或 **hex** 是一个基于 16 的数字系统，它使用数字 0 到 9 和字母 A 到 F（16 个字符）来表示一组四个二进制数字，其值可以是 0 到 15。
•读取 32 位内存长度为 8 个字符的十六进制数比尝试以二进制方式解密 32 个 1 和 0 更容易。以下程序显示变量 **i** 和 **k** 的内存地址：

```
void test(int k);

int main() {
   int i = 0;

   printf("The address of i is %x\n", &i);
   test(i);
   printf("The address of i is %x\n", &i);
   test(i);

   return 0;
}

void test(int k) {
   printf("The address of k is %x\n", &k);
}
```

• 在 printf 语句中，**%x** 是十六进制格式说明符。
• 程序输出因运行而异，但看起来类似于：

```
The address of i is 846dd754
The address of k is 846dd758
The address of i is 846dd754
The address of k is 846dd758
```

• 变量的地址从声明它的域到结束时保持不变。

✧ 什么是指针？

• **指针**在 C 编程中非常重要，因为它们允许您轻松处理内存位置。
• 它们是数组，字符串和其他数据结构和算法的基础。
• **指针**是包含另一个变量的**地址**的变量。换句话说，它“指向”分配给变量的位置，并且可以间接访问变量。
• 指针使用*符号声明并采用以下形式：

```
pointer_type *identifier
```

• **pointer_type** 是指针指向的数据类型。实际指针数据类型是**十六进制数**，但在声明指针时，必须指出它将指向哪种类型的数据。
• 星号 *声明一个指针，应该出现在用于指针变量的标识符旁边。
以下程序演示变量，指针和地址：

```
int j = 63;
int *p = NULL;
p = &j;

printf("The address of j is %x\n", &j);
printf("p contains address %x\n", p);
printf("The value of j is %d\n", j);
printf("p is pointing to the value %d\n", *p);
```

关于这个程序有几点需要注意：
•指针应初始化为 **NULL**，直到为其指定有效位置。
•可以使用**&**符号为指针指定变量的地址。
•要查看指针指向的内容，请再次使用*，如***p** 中所示。在这种情况下，*被称为间接或**解引用**运算符。该过程称为**解引用**。

程序输出类似于：

```
The address of j is ff3652cc
p contains address ff3652cc
The value of j is 63
p is pointing to the value 63
```

一些算法使用**指向指针的指针**。这种类型的变量声明使用**，并且可以为其分配另一个指针的地址，如下所示：

```
int x = 12;
int * p = NULL
int ** ptr = NULL;
p =&x;
ptr =&p;
```

✧ 表达式中的指针

• 指针可以像任何变量一样在**表达式**中使用。算术运算符可以应用于指针指向的任何内容。
例如：

```
int x = 5;
int y;
int *p = NULL;
p = &x;

y = *p + 2; /* y is assigned 7 */
y += *p;         /* y is assigned 12 */
*p = y;            /* x is assigned 12 */
(*p)++;          /* x is incremented to 13 */

printf("p is pointing to the value %d\n", *p);
```

• 请注意，++运算符需要使用括号来增加指向的值。使用--运算符时也是如此。

3.6.更多关于指针

✧ 指针和数组

• 指针对数组特别有用。数组声明为其元素保留一块连续的内存地址。使用指针，我们可以指向第一个元素，然后使用**地址运算**遍历数组：
+ 内存位置向前移动
- 内存位置向后移动

思考以下程序：

```
int a[5] = {22, 33, 44, 55, 66};
int *ptr = NULL;
int i;

ptr = a;
for (i = 0; i < 5; i++) {
   printf("%d ", *(ptr + i));
}
```

程序输出为：22 33 44 55 66

• 数组的一个重要概念是**数组名称**充当指向数组第一个元素的**指针**。因此，语句 **ptr = a** 可以被认为是 **ptr=&a [0]**。

• 考虑以下语句，它打印数组的第一个元素：printf("%d ",*a);

✧ 更多地址运算

• 地址运算也可以被认为是指针算术，因为操作涉及指针。
• 除了使用+和-来引用下一个和前一个内存位置，您还可以使用赋值运算符来更改指针包含的地址。
例如：

```
int a[5] = {22, 33, 44, 55, 66};
int *ptr = NULL;

ptr = a;    /* point to the first array element */
printf("%d    %x\n", *ptr, ptr);    /* 22 */
ptr++;
printf("%d    %x\n", *ptr, ptr);    /* 33 */
ptr += 3;
printf("%d    %x\n", *ptr, ptr);    /* 66 */
ptr--;
printf("%d    %x\n", *ptr, ptr);    /* 55 */
ptr -= 2;
printf("%d    %x\n", *ptr, ptr);    /* 33 */
```

程序输出类似于：

```
22 febd4760
33 febd4764
66 febd4770
55 febd476c
33 febd4764
```

• 当指针递增时，指针指向增加字节数后的存储地址。在上面的程序中，当使用递增运算符(ptr ++)时，指针增加 4，因为指针指向 int。
您还可以使用==，< 和> 运算符来比较指针地址。

✧ 指针和函数

• 指针极大地扩展了函数的可能性。我们不再仅限于返回一个值。使用指针参数，您的函数可以更改实际数据而不是数据副本。
• 要更改变量的实际值，调用语句将地址传递给函数中的指针参数。
• 例如，以下程序交换两个值：

```
void swap (int *num1, int *num2);
```

```
int main() {
   int x = 25;
   int y = 100;

   printf("x is %d, y is %d\n", x, y);
   swap(&x, &y);
   printf("x is %d, y is %d\n", x, y);

   return 0;
}

void swap (int *num1, int *num2) {
   int temp;

   temp = *num1;
   *num1 = *num2;
   *num2 = temp;
}
```

• 程序交换变量的实际值，因为函数使用指针通过地址访问它们。

3.7.函数和数组

✧ 具有数组参数的函数

• 数组不能通过值传递给函数。但是，数组名称是一个指针，因此只需将数组名称传递给函数即可将指针传递给该数组。

思考以下程序：

```
int add_up (int *a, int num_elements);

int main() {
   int orders[5] = {100, 220, 37, 16, 98};

   printf("Total orders is %d\n", add_up(orders, 5));

   return 0;
}

int add_up (int *a, int num_elements) {
   int total = 0;
   int k;
```

```
   for (k = 0; k < num_elements; k++) {
      total += a[k];
   }

   return (total);
}
```

程序输出为：“Total orders is 471”

✧ 返回数组的函数

•就像指向数组的指针可以传递给函数一样，可以返回指向数组的指针，如下面的程序所示：

```
int * get_evens();

int main() {
   int *a;
   int k;

   a = get_evens(); /* get first 5 even numbers */
   for (k = 0; k < 5; k++)
      printf("%d\n", a[k]);

   return 0;
}

int * get_evens() {
   static int nums[5];
   int k;
   int even = 0;

   for (k = 0; k < 5; k++) {
      nums[k] = even += 2;
   }

   return (nums);
}
```

• 请注意，声明指针而不是数组来存储函数返回的值。另请注意，当局部变量从函数中传出时，您需要在函数中将其声明为**静态**。

• 请记住 **a[k]**与***(a + k)**相同。

4.字符串和函数指针

4.1.字符串

✧ 字符串

• C 中的**字符串**是以 **NULL 字符'\ 0'**结尾的字符数组。
字符串声明可以通过多种方式进行，每种方式都有自己的注意事项。
例如：

```
char str_name[str_len] = "string";
```

• 这将创建一个名为 str_name 的 str_len 字符的字符串，并将其初始化为值“string”。
• 当您提供**字符串文字**来初始化字符串时，编译器会自动向 char 数组添加 NULL 字符'\ 0'。
• 因此，必须声明数组大小至少比预期的字符串长度长一个字符。
• 下面的语句创建包含 NULL 字符的字符串。如果声明不包含 char 数组大小，那么它将根据初始化中字符串的长度加上'\ 0'的长度计算：

```
char str1[6] = "hello";
char str2[ ] = "world";    /* size 6 */
```

• 字符串也可以声明为一组字符：

```
char str3[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
char str4[ ] = {'h', 'e', 'l', 'l', 'o', '\0'}; /* size 6 */
```

• 使用此方法，必须显式添加 NULL 字符。请注意，**字符**用单引号括起来。
• 与任何数组一样，字符串的名称充当指针。
• 字符串文字是用双引号括起来的文本。
• 字符（如'b'）由单引号表示，不能视为字符串。

• 字符串指针声明，例如 char ***str = "stuff";** 被认为是常数，不能从其初始值改变。

• 要使用字符串安全方便地操作，可以使用下面显示的*标准库*字符串函数。不要忘记包含<string.h>。
strlen() - 获取字符串的长度
strcat() - 合并两个字符串
strcpy() - 将一个字符串复制到另一个字符串
strlwr() - 将字符串转换为小写
strupr() - 将字符串转换为大写字母
strrev() - 反向字符串
strcmp() - 比较两个字符串

✧ 字符串输入

• 程序通常是交互式的，要求用户输入。
• 要从用户检索一行文本或其他字符串，C 提供 **scanf()**，**gets()**和 **fgets()**函数。
• 您可以使用 **scanf()**根据格式说明符读取输入。
例如：

```
char first_name[25];
int age;
printf("Enter your first name and age: \n");
scanf("%s %d", first_name, &age);
```

• 当 **scanf()**用于读取字符串时，不需要**&**来访问变量地址，因为数组名称充当指针。
• **scanf()**在到达空格时停止读取输入。要读取带空格的字符串，请使用 **gets()**函数。它会读取输入，直到到达终止换行符（按下 Enter 键）。
例如：

```
char full_name[50];
printf("Enter your full name: ");
gets(full_name);
```

• **gets()**的一个更安全的替代方法是 **fgets()**，它读取指定数量的字符。此方法有助于防止**缓冲区溢出**，当字符串数组对于键入的文本不够大时会发生缓冲区溢出。
例如：

```
char full_name[50];
printf("Enter your full name: ");
fgets(full_name, 50, stdin);
```

• **fgets()**参数是字符串名称，要读取的字符数，以及指向要从中读取字符串的位置的指针。**stdin** 意味着从**标准输入**读取，即键盘。
• **gets** 和 **fgets** 之间的另一个区别是换行符由 **fgets** 存储。
fgets()只从 **stdin** 读取 **n-1** 个字符，因为**'\ 0'**必须有空间。

✧ 字符串输出

• 字符串输出由 **fputs()**，**putf()**和 **printf()**函数处理。
• **fputs()**需要字符串的名称和指向要打印字符串的位置的指针。要打印到屏幕，请使用参考**标准输出**的 **stdout**。
例如：

```
#include <stdio.h>
int main()
{
   char city[40];
   printf("Enter your favorite city: ");
```

```
   gets(city);
   // Note: for safety, use
   // fgets(city, 40, stdin);

   fputs(city, stdout);
   printf(" is a fun city.");

   return 0;
}
```

• **puts()**函数只接受一个字符串参数，也可以用来显示输出。但是，它为输出添加了换行符。

例如：

```
#include <stdio.h>
int main()
{
   char city[40];
   printf("Enter your favorite city: ");
   gets(city);
   // Note: for safety, use
   // fgets(city, 40, stdin);

   printf("%s is a fun city.", city);

   return 0;
}
```

4.2.字符串函数

✧ sprintf 和 sscanf 函数

• 可以使用 **sprintf()**函数创建格式化的字符串。这对于从其他数据类型构建字符串很有用。

例如：

```
#include <stdio.h>
int main()
{
   char info[100];
   char dept[ ] = "HR";
   int emp = 75;
   sprintf(info, "The %s dept has %d employees.", dept, emp);
   printf("%s\n", info);

   return 0;
```

```
}
```

• 另一个有用的函数是 **sscanf()**，用于扫描字符串的值。该函数从字符串中读取值并将它们存储在相应的变量地址中。

例如：

```
#include <stdio.h>
int main()
{
   char info[ ] = "Snoqualmie WA 13190";
   char city[50];
   char state[50];
   int population;
   sscanf(info, "%s %s %d", city, state, &population);
   printf("%d people live in %s, %s.", population, city, state);

   return 0;
}
```

✧ string.h 库

• **string.h** 库包含许多字符串函数。

• 程序顶部的语句**#include <string.h>**使您可以访问以下内容：

strlen(str)返回存储在 **str** 中的字符串的长度，不包括 NULL 字符。

strcat(str1,str2)将（连接）**str2** 追加到 **str1** 的末尾并返回指向 **str1** 的指针。

strcpy(str1,str2)将 **str2** 复制到 **str1**。此函数可用于为字符串分配新值。

下面的程序演示了 string.h 函数：

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

int main()
{
   char s1[ ] = "The grey fox";
   char s2[ ] = " jumped.";

   strcat(s1, s2);
   printf("%s\n", s1);
   printf("Length of s1 is %d\n", strlen(s1));
   strcpy(s1, s2);
   printf("s1 is now %s \n", s1);

   return 0;
}
```

• 其他的 string.h 函数包括：
strncat(str1,str2,n)将 **str2** 的前 **n** 个字符（连接）追加到 **str1** 的末尾并返回指向 **str1** 的指针。
strncpy(str1,str2,n)将 **str2** 的前 **n** 个字符复制到 **str1**。
strcmp(str1,str2)当 **str1** 等于 **str2** 时返回 **0**，当 **str1 <str2** 时小于 0，当 **str1> str2** 时大于 0。
strncmp(str1,str2,n)当 **str1** 的前 **n** 个字符等于 **str2** 的前 **n** 个字符时，返回 0，当 **str1 <str2** 时小于 0，当 **str1> str2** 时大于 0。
strchr(str1,c)返回指向 **str1** 中第一次出现的字符 **c** 的指针，如果找不到字符，则返回 NULL。
strrchr(str1,c)反向搜索 **str1** 并返回指向 **str1** 中字符 **c** 位置的指针，如果找不到字符，则返回 NULL。
strstr(str1,str2)返回指向 **str1** 中第一次出现 **str2** 的指针，如果未找到 **str2**，则返回 NULL。

✧ 字符串转化为数字

• 将数字字符串转换为数值是 C 中的常见任务，通常用于防止运行时错误。
• 读取字符串比预期数字值更不容易出错，只是让用户不小心输入“o”而不是“0”（零）。

stdio.h 库包含以下用于将字符串转换为数字的函数：
int atoi(str)代表 ASCII 到整数。将 **str** 转换为等效的 int 值。如果第一个字符不是数字或没有遇到数字，则返回 0。
double atof(str)代表 ASCII 浮点。将 **str** 转换为等效的 double 值。如果第一个字符不是数字或没有遇到数字，则返回 0.0。
long int atol(str)代表 ASCII 到 long int。将 **str** 转换为等效的长整数值。如果第一个字符不是数字或没有遇到数字，则返回 0。

以下程序演示了 **atoi**。

```
#include <stdio.h>
int main()
{
   char input[10];
   int num;

   printf("Enter a number: ");
   gets(input);
   num = atoi(input);

   return 0;
}
```

注意，**atoi()**缺少错误处理，如果要确保正确的错误处理完成，建议使用 **strtol()**。

✧ 字符串数组

• 二维数组可用于存储相关的字符串。
• 请考虑以下语句，该语句声明一个包含 3 个元素的数组，每个元素包含 15 个字符：

```
char trip[3][15] = {
   "suitcase",
   "passport",
   "ticket"
};
```

• 尽管字符串长度不同，但有必要声明一个足够大的大小来容纳最长的字符串。另外，访问元素可能非常麻烦。
• 用 trip[0]访问”suitcase”是容易出错的。相反，你必须将**[0] [0]**处的元素视为's'，将**[2] [3]**处的元素视为'**k**'，依此类推。

• 处理相关字符串集合的更简单，更直观的方法是使用指针数组，如以下程序：

```
char *trip[ ] = {
   "suitcase",
   "passport",
   "ticket"
};

printf("Please bring the following:\n");
for (int i = 0; i < 3; i++) {
   printf("%s\n", trip[ i ]);
}
```

• 因为每个元素的长度可以变化，所以字符串指针的数组具有更通用的结构，而不是二维网格结构。
• 使用这种方法，字符串长度没有限制。更重要的是，项目可以通过指向每个字符串的第一个字符的指针来引用。

• 请记住像 **char * items [3]**这样的声明；只保留三个指针的空间；这些指针引用了实际的字符串。

4.3.函数指针

✧ 函数指针

• 由于指针可以指向任何内存位置中的地址，因此它们也可以指向可执行代码的开头。
• 指向函数或**函数指针**的指针指向内存中函数的可执行代码。函数指针可以存储在数组中，

也可以作为参数传递给其他函数。

• 函数指针**声明**使用*就像使用任何指针一样：

```
return_type (*func_name)(parameters)
```

• 括号(* func_name)很重要。没有它们，编译器会认为函数正在返回一个指针。

•声明函数指针后，必须将其指定给函数。下面的短程序声明一个函数，声明一个函数指针，将函数指针赋给该函数，然后通过指针调用该函数：

```
#include <stdio.h>
void say_hello(int num_times); /* function */

int main() {
   void (*funptr)(int);    /* function pointer */
   funptr = say_hello;    /* pointer assignment */
   funptr(3);    /* function call */

   return 0;
}

void say_hello(int num_times) {
   int k;
   for (k = 0; k < num_times; k++)
      printf("Hello\n");
}
```

• 函数名称指向可执行代码的开头，就像数组名称指向其第一个元素一样。因此，虽然诸如**funptr=&say_hello** 和**(* funptr)(3)**之类的语句是正确的，但是在函数赋值和函数调用中不必包括地址运算符&和间接运算符*。

✧ 函数指针数组

• 函数指针数组可以替换 **switch** 或 **if** 语句来选择操作，如下面的程序所示：

```
#include <stdio.h>

int add(int num1, int num2);
int subtract(int num1, int num2);
int multiply(int num1, int num2);
int divide(int num1, int num2);

int main()
{
```

```
    int x, y, choice, result;
    int (*op[4])(int, int);

    op[0] = add;
    op[1] = subtract;
    op[2] = multiply;
    op[3] = divide;
    printf("Enter two integers: ");
    scanf("%d%d", &x, &y);
    printf("Enter 0 to add, 1 to subtract, 2 to multiply, or 3 to divide: ");
    scanf("%d", &choice);
    result = op[choice](x, y);
    printf("%d", result);

    return 0;
}

int add(int x, int y) {
    return(x + y);
}

int subtract(int x, int y) {
    return(x - y);
}

int multiply(int x, int y) {
    return(x * y);
}

int divide(int x, int y) {
    if (y != 0)
        return (x / y);
    else
        return 0;
}
```

• 语句 **int(*op[4])(int,int);** 声明函数指针数组。每个数组元素必须具有相同的参数和返回类型。在这种情况下，分配给数组的函数有两个 **int** 参数并返回一个 **int**。语句 **result= op [choice](x,y);** 根据用户的选择执行适当的功能。先前输入的整数是传递给函数的参数。

4.4.void 指针

✧ void 指针

• **void** 指针用于引用内存中的任何地址类型，并具有如下声明：
void * ptr;

以下程序对三种不同的数据类型使用相同的指针：

```
int x = 33;
float y = 12.4;
char c = 'a';
void *ptr;

ptr = &x;
printf("void ptr points to %d\n", *((int *)ptr));
ptr = &y;
printf("void ptr points to %f\n", *((float *)ptr));
ptr = &c;
printf("void ptr points to %c", *((char *)ptr));
```

• 取消引用 void 指针时，必须首先在使用*解除引用之前键入将指针**强制转换**为适当的数据类型。
您无法使用 void 指针执行指针运算。

✧ 函数使用 void 指针

• Void 指针通常用于函数声明。
例如：

```
void * square (const void *);
```

• 使用 **void** *返回类型允许任何返回类型。类似地，**void** *的参数接受任何参数类型。如果要使用参数传入的数据而不更改它，则将其声明为 **const**。

•您可以省略参数名称以进一步隔离声明与其实现。以这种方式声明函数允许根据需要定义定义，而无需更改声明。

考虑以下程序：

```
#include <stdio.h>

void* square (const void* num);
```

```
int main() {
   int x, sq_int;
   x = 6;
   sq_int = square(&x);
   printf("%d squared is %d\n", x, sq_int);

   return 0;
}

void* square (const void *num) {
   int result;
   result = (*(int *)num) * (*(int *)num);
   return result;
}
```

• 这个**平方**函数已被写为**整数**乘法，这就是将 **num** void 指针强制转换为 **int** 的原因。如果要将实现更改为允许 **square**()乘以**浮点数**，则只需更改定义而无需更改声明。

✧ 函数指针作为参数

• 使用函数指针的另一种方法是将其作为参数传递给另一个函数。
• 用作参数的函数指针有时被称为**回调函数**，因为接收函数“将其回调”。
• **stdlib.h** 头文件中的 **qsort()**函数使用此技术。

• Quicksort 是一种广泛使用的排序数组的算法。要在程序中实现排序，只需要包含 **stdlib.h** 文件，然后编写一个与 **qsort** 中使用的声明匹配的 compare 函数：

```
void qsort(void *base, size_t num, size_t width, int (*compare)(const void *, const void *))
```

• 要分解 **qsort** 声明：
void *base 指向数组的 void 指针。
size_t num 数组中的元素数。
size_t width 元素的大小。
int(*compare(const void *,const void *)一个函数指针，它有两个参数，当参数具有相同的值时返回 0，当 arg1 在 arg2 之前时返回<0，当 arg1 在 arg2 之后时返回> 0。

• 比较功能的实际实现取决于您。它甚至不需要名称“比较”。您可以指定从高到低或从低到高的排序，或者如果数组包含结构元素，则可以比较成员值。

以下程序使用 **qsort** 从低到高对一组 **int** 进行排序：

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

int compare (const void *, const void *);
```

```
int main() {
   int arr[5] = {52, 23, 56, 19, 4};
   int num, width, i;

   num = sizeof(arr)/sizeof(arr[0]);
   width = sizeof(arr[0]);
   qsort((void *)arr, num, width, compare);
   for (i = 0; i < 5; i++)
      printf("%d ", arr[ i ]);

   return 0;
}

int compare (const void *elem1, const void *elem2) {
   if ((*(int *)elem1) == (*(int *)elem2))
      return 0;
   else if ((*(int *)elem1) < (*(int *)elem2))
      return -1;
   else
      return 1;
}
```

- 我们在 **qsort** 调用中使用了函数名，因为函数名作为指针。

5.结构体和联合体

5.1.结构体

✧ 结构体

• **结构体**是**用户定义的数据类型**，它对不同数据类型的相关变量进行分组。

• 结构**声明**包括关键字 **struct**，用于引用结构的**结构标记**，以及名为 **members** 的变量声明列表的花括号{}。

例如：

```
struct course {
   int id;
   char title[40];
   float hours;
};
```

• 此 struct 语句定义一个名为 **course** 的新数据类型，该类型具有三个成员。
• 结构成员可以是任何数据类型，包括基本类型、字符串、数组、指针，甚至其他结构，您将在后面的课程中学习。
不要忘记在结构声明后加上分号。
结构也称为**复合**或**聚合**数据类型。有些语言将结构称为**记录**。

✧ 使用结构的声明

• 要**声明**结构数据类型的**变量**，请使用关键字 **struct**，后跟 struct 标记，然后使用变量名称。
• 例如，下面的语句声明了一个结构数据类型，然后使用 **student** 结构声明变量 **s1** 和 **s2**：

```
struct student {
   int age;
   int grade;
   char name[40];
};

/* declare two variables */
struct student s1;
struct student s2;
```

结构变量存储在连续的内存块中。必须使用 **sizeof** 运算符来获取结构所需的字节数，就像使用基本数据类型一样。

✧ 使用结构的声明

• 通过在花括号内按顺序列出初始值，也可以在声明中初始化 struct 变量：

```
struct student s1 = {19, 9, "John"};
struct student s2 = {22, 10, "Batman"};
```

• 如果要在声明后使用花括号初始化结构，则还需要在语句中**强制转换**：

```
struct student s1;
s1 = (struct student) {19, 9, "John"};
```

• 初始化结构以初始化相应的成员时，可以使用命名成员初始化：

```
struct student s1
= { .grade = 9, .age = 19, .name = "John"};
```

• 在上面的示例中，**.grade** 是指结构的成员成员。同样，**.age** 和**.name** 指的是年龄和名称成员。

✧ 访问结构成员

• 您可以在变量名和成员名之间使用**.（点运算符）**访问 struct 变量的成员。
例如，要为 **s1** 结构体变量的 **age** 成员**赋值**，请使用如下语句：

```
s1.age = 19;
```

• 您还可以将一个结构分配给同一类型的另一个结构：

```
struct student s1 = {19, 9, "Jason"};
struct student s2;
//....
s2 = s1;
```

• 以下代码演示了如何使用结构：

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

struct course {
   int id;
   char title[40];
   float hours;
};

int main() {
   struct course cs1 = {341279, "Intro to C++", 12.5};
   struct course cs2;
```

```
    /* initialize cs2 */
    cs2.id = 341281;
    strcpy(cs2.title, "Advanced C++");
    cs2.hours = 14.25;

    /* display course info */
    printf("%d\t%s\t%4.2f\n", cs1.id, cs1.title, cs1.hours);
    printf("%d\t%s\t%4.2f\n", cs2.id, cs2.title, cs2.hours);

    return 0;
}
```

- 字符串赋值需要 string.h 库中的 **strcpy()**。
- 另请注意，格式说明符**%4.2f** 包括宽度和精度选项。

✧ 使用 typedef

- **typedef** 关键字创建一个类型定义，简化代码并使程序更易于阅读。
- **typedef** 通常与结构一起使用，因为它在声明变量时不需要使用关键字 **struct**。

例如：

```
typedef struct {
    int id;
    char title[40];
    float hours;
} course;

course cs1;
course cs2;
```

- 请注意，不再使用结构标记，而是在 struct 声明之前出现 **typedef** 名称。
- 现在变量声明中不再需要 struct 这个词，使代码更清晰，更易于阅读。

5.2.使用结构体

✧ 具有结构体的结构体

- 结构的成员也可以是结构。

例如，请考虑以下语句：

```
typedef struct {
```

```
    int x;
    int y;
} point;

typedef struct {
    float radius;
    point center;
} circle;
```

• 嵌套花括号用于初始化结构的成员。点运算符用于访问成员的成员，如在语句中：

```
circle c = {4.5, {1, 3}};
printf("%3.1f %d,%d", c.radius, c.center.x, c.center.y);
/* 4.5    1,3 */
```

必须先出现结构体的定义，然后才能在另一个结构体中使用它。

✧ 指向结构体的指针

• 就像指向变量的指针一样，也可以定义指向结构体的指针。

struct myStruct * struct_ptr;
定义指向 myStruct 结构体的指针。

struct_ptr =＆struct_var;
将结构体变量 struct_var 的地址存储在指针 struct_ptr 中。

struct_ptr - > struct_mem;
访问结构体成员 struct_mem 的值。

例如：

```
struct student{
    char name[50];
    int number;
    int age;
};

// Struct pointer as a function parameter
void showStudentData(struct student *st) {
    printf("\nStudent:\n");
    printf("Name: %s\n", st->name);
    printf("Number: %d\n", st->number);
    printf("Age: %d\n", st->age);
}
```

```
struct student st1 = {"Krishna", 5, 21};
showStudentData(&st1);
```

• **->**运算符允许通过指针访问结构体的成员。
(* st).age 与 **st-> age** 相同。
此外，当使用 **typedef** 命名结构时，则仅使用 typedef 名称以及*和指针名称来声明指针。

✧ 结构体作为函数参数

• 函数可以具有结构参数，当需要所有结构变量的副本时，该参数**按值**接受参数。
• 对于更改 struct 变量中实际值的函数，需要**指针参数**。

例如：

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

typedef struct {
   int id;
   char title[40];
   float hours;
} course;

void update_course(course *class);
void display_course(course class);

int main() {
   course cs2;
   update_course(&cs2);
   display_course(cs2);
   return 0;
}

void update_course(course *class) {
   strcpy(class->title, "C++ Fundamentals");
   class->id = 111;
   class->hours = 12.30;
}

void display_course(course class) {
   printf("%d\t%s\t%3.2f\n", class.id, class.title, class.hours);
}
```

• 如您所见，**update_course()**将指针作为参数，而 **display_course()**则按值获取结构。

✧ 结构数组

• 数组可以存储任何数据类型的元素，包括结构。
• 声明结构数组后，可以使用索引号访问元素。
• 然后使用点运算符来访问元素的成员，如在程序中：

```
#include <stdio.h>

typedef struct {
   int h;
   int w;
   int l;
} box;

int main() {
   box boxes[3] = {{2, 6, 8}, {4, 6, 6}, {2, 6, 9}};
   int k, volume;

   for (k = 0; k < 3; k++) {
      volume = boxes[k].h*boxes[k].w*boxes[k].l;
      printf("box %d volume %d\n", k, volume);
   }
   return 0;
}
```

• 结构数组用于数据结构，例如链表，二叉树等。

5.3.联合体

✧ 联合体

• **联合体**允许在同一内存位置存储不同的数据类型。
• 它就像一个结构，因为它有成员。但是，union 变量对其所有成员使用相同的内存位置，并且一次只有一个成员可以占用内存位置。

• union **声明**使用关键字 **union**，**union 标记**和带有**成员**列表的花括号{}。

• 联合体成员可以是任何数据类型，包括基本类型、字符串、数组、指针和结构。

例如：

```
union val {
   int int_num;
   float fl_num;
   char str[20];
};
```

•声明联合体后，可以声明联合体变量。您甚至可以将一个联合体分配给同一类型的另一个：

```
union val u1;
union val u2;
u2 = u1;
```

•联合体用于**内存管理**。最大的成员数据类型用于确定要共享的内存大小，然后所有成员使用此位置。此过程还有助于限制内存碎片。内存管理将在后面的课程中讨论。

✧ 访问联合体成员

• 您可以使用通过变量名和成员名之间的.**点运算符**来访问 union 变量的成员。
• 执行赋值时，联合体内存位置将用于该成员，直到执行另一个成员分配。

• 尝试访问未占用内存位置的成员会产生**意外结果**。

以下程序演示了如何访问 union 成员：

```
union val {
   int int_num;
   float fl_num;
   char str[20];
};

union val test;

test.int_num = 123;
test.fl_num = 98.76;
strcpy(test.str, "hello");

printf("%d\n", test.int_num);
printf("%f\n", test.fl_num);
printf("%s\n", test.str);
```

最后一个赋值会覆盖先前的赋值，这就是 **str** 存储值并访问 int_num 和 fl_num 无意义的原因。

✧ 带联合体的结构体

• 联合体通常在结构体中使用，因为结构体可以有一个成员来跟踪哪个联合体成员存储值。例如，在以下程序中，车辆结构使用车辆识别号(VIN)或指定的 ID，但不是两者：

```
typedef struct {
   char make[20];
   int model_year;
   int id_type; /* 0 for id_num, 1 for VIN */
   union {
      int id_num;
      char VIN[20];
   } id;
} vehicle;

vehicle car1;
strcpy(car1.make, "Ford");
car1.model_year = 2017;
car1.id_type = 0;
car1.id.id_num = 123098;
```

• 请注意，联合体已在结构体内声明。执行此操作时，声明末尾需要**联合体名称**。
• 具有联合体标记的联合体可以在结构外部声明，但是通过这种特定用途，结构体中的联合提供了更容易理解的代码。

• 另请注意，访问 struct 成员的 union 成员使用**点运算符**两次。

• id_type 跟踪哪个联合体成员存储值。以下语句显示 car1 数据，使用 id_type 确定要读取的联合体成员：

```
/* display vehicle data */
printf("Make: %s\n", car1.make);
printf("Model Year: %d\n", car1.model_year);
if (car1.id_type == 0)
   printf("ID: %d\n", car1.id.id_num);
else
   printf("ID: %s\n", car1.id.VIN);
```

联合体也可以包含结构体。

5.4.使用联合体

✧ 指向联合体的指针

• 指向 union 的指针指向分配给 union 的内存位置。
• 通过使用关键字 **union** 和 union 标记以及*和指针名称声明**联合体指针**。
例如，请考虑以下语句：

```
union val {
   int int_num;
   float fl_num;
   char str[20];
};

union val info;
union val *ptr = NULL;
ptr = &info;
ptr->int_num = 10;
printf("info.int_num is %d", info.int_num);
```

如果要通过指针访问联合成员，则需要**->运算符**。
(*ptr).int_num 与 ptr->int_num 相同。

✧ 联合体作为函数参数

• 函数可以具有联合体参数，当联合体变量的副本完全所需时，它们可以**通过值**接受参数。

• 对于更改联合内存位置中的实际值的函数，需要指针参数。
例如：

```
union id {
   int id_num;
   char name[20];
};

void set_id(union id *item) {
   item->id_num = 42;
}

void show_id(union id item) {
   printf("ID is %d", item.id_num);
}
```

✧ 联合体数组

• 数组可以存储任何数据类型的元素，包括联合体。
• 对于联合体，重要的是要记住只有一个联合体成员可以存储每个数组元素的数据。

• 声明一个联合数组后，可以使用索引号访问一个元素。然后使用点运算符来访问联合体成员，如在程序中：

```
union val {
   int int_num;
   float fl_num;
   char str[20];
};

union val nums[10];
int k;

for (k = 0; k < 10; k++) {
   nums[k].int_num = k;
}

for (k = 0; k < 10; k++) {
   printf("%d    ", nums[k].int_num);
}
```

•数组是一种数据结构，用于存储所有**类型相同**的集合值。联合数组允许存储**不同类型**的值。
例如：

```
union type {
   int i_val;
   float f_val;
   char ch_val;
};
union type arr[3];
arr[0].i_val = 42;
arr[1].f_val = 3.14;
arr[2].ch_val = 'x';
```

6.内存管理

6.1.使用内存

✧ 内存管理

• 了解内存是 C 编程的一个重要方面。使用基本数据类型声明变量时，C 会自动为称为**堆栈**的内存区域中的变量分配空间。

例如，**int** 变量通常在声明时分配 4 个字节。 我们通过使用 **sizeof** 运算符来了解这一点：

```
int x;
printf("%d", sizeof(x)); /* output: 4 */
```

作为另一个示例，具有指定大小的数组被**分配连续**的内存块，每个块具有一个元素的大小：

```
int arr[10];
printf("%d", sizeof(arr)); /* output: 40 */
```

• 只要程序显式声明基本数据类型或数组大小，就会自动管理内存。但是，您可能已经希望实现一个程序，其中数组大小在运行时尚未确定。

• **动态内存分配**是根据需要分配和释放内存的过程。现在，您可以在运行时提示数组元素的数量，然后创建一个包含许多元素的数组。动态内存由指针管理，指针指向称为**堆**的区域中新分配的内存块。

除了使用堆栈的自动内存管理和使用堆的动态内存分配之外，**主内存**中还有**静态管理的数据**，用于在程序的生命周期内持续存在的变量。

✧ 内存管理函数

• **stdlib.h** 库包含内存管理功能。
• 程序顶部的语句**#include <stdlib.h>**使您可以访问以下内容：

malloc(*bytes*)返回指向大小为 bytes 的连续内存块的指针。

calloc(*num_items,item_size*)返回指向具有 num_items 项的连续内存块的指针，每个项的大小为 item_size 字节。通常用于数组，结构和其他派生数据类型。分配的内存初始化为 0。

realloc(*ptr,bytes*)将 ptr 指向的内存大小调整为 size 字节。新分配的内存未初始化。

free(*ptr*)释放 ptr 指向的内存块。

当您不再需要已分配内存块时，请使用函数 free()使该块可用于再次分配。

6.2.malloc 函数

✧ malloc 函数

• **malloc()**函数在内存中分配指定数量的**连续字节**。

例如：

```
#include <stdlib.h>

int *ptr;
/* a block of 10 ints */
ptr = malloc(10 * sizeof(*ptr));

if (ptr != NULL) {
   *(ptr + 2) = 50;    /* assign 50 to third int */
}
```

• **malloc** 返回指向已分配内存的指针。

• 请注意，sizeof 应用于* ptr 而不是 int，如果稍后将* ptr 声明更改为其他数据类型，则会使代码更加包容。

✧ malloc 函数

• 分配的内存是**连续**的，可以视为一个**数组**。而不是使用方括号[]来引用元素，而是使用指针算法来遍历数组。建议您使用+来引用数组元素。使用++或+ =更改指针存储的地址。

• 如果分配不成功，则返回 **NULL**。因此，您应该包含检查 NULL 指针的代码。

一个简单的二维数组需要**(rows*columns)*sizeof(datatype)**字节的内存。

✧ free 函数

• **free()**函数是一个内存管理函数，调用它来**释放内存**。通过释放内存，您可以在以后的程序中使用更多内存。

例如：

```
int* ptr = malloc(10 * sizeof(*ptr));
if (ptr != NULL)
```

```
   *(ptr + 2) = 50;    /* assign 50 to third int */
printf("%d\n", *(ptr + 2));

free(ptr);
```

6.3.calloc 和 realloc 函数

✧ calloc 函数

• **calloc()**函数根据特定项的大小（例如结构）分配内存。
• 下面的程序使用 **calloc** 为结构体分配内存，**malloc** 为结构体中的字符串分配内存：

```
typedef struct {
   int num;
   char *info;
} record;

record *recs;
int num_recs = 2;
int k;
char str[ ] = "This is information";

recs = calloc(num_recs, sizeof(record));
if (recs != NULL) {
   for (k = 0; k < num_recs; k++) {
      (recs+k)->num = k;
      (recs+k)->info = malloc(sizeof(str));
      strcpy((recs+k)->info, str);
   }
}
```

• **calloc** 在**连续的内存块**中为结构元素数组分配内存块。您可以使用指针算法从一个结构体导航到下一个结构体。

• 为结构体分配空间后，必须为结构中的字符串分配内存。使用 info 成员的指针允许存储任何长度的字符串。

• 动态分配的结构是**链表**和**二叉树**以及其他数据结构的基础。

✧ realloc 函数

• **realloc()**函数扩展当前块以包含额外的内存。

例如：

```
int *ptr;
ptr = malloc(10 * sizeof(*ptr));
if (ptr != NULL) {
   *(ptr + 2) = 50;    /* assign 50 to third int */
}
ptr = realloc(ptr, 100 * sizeof(*ptr));
*(ptr + 30) = 75;
```

• **realloc** 将原始内容保留在内存中并扩展块以允许更多存储。

6.4.动态字符串和数组

✧ 为字符串分配内存

•为字符串指针分配内存时，您可能希望使用字符串长度而不是 sizeof 运算符来计算字节数。考虑以下程序：

```
char str20[20];
char *str = NULL;

strcpy(str20, "12345");
str = malloc(strlen(str20) + 1);
strcpy(str, str20);
printf("%s", str);
```

•这种方法更有益于内存管理，因为您没有分配比指针所需的更多空间。使用 **strlen** 确定字符串所需的字节数时，请确保为 NULL 字符'\ 0'预留一个额外字节。

• **char** 总是一个字节，因此不需要将内存需求乘以 sizeof(char)。

✧ 动态数组

• 许多算法实现**动态数组**，因为这允许元素的数量根据需要增长。

•由于元素不是一次性分配的，因此动态数组通常使用结构体来跟踪当前数组大小、当前容量和指向元素的指针，如下面的程序所示。

```
typedef struct {
   int *elements;
```

```
  int size;
  int cap;
} dyn_array;

dyn_array arr;

/* initialize array */
arr.size = 0;
arr.elements = calloc(1, sizeof(arr.elements) );
arr.cap = 1;   /* room for 1 element */
```

要扩展更多元素：

```
arr.elements = realloc(arr.elements, (5 + arr.cap) * sizeof(arr.elements));
if (arr.elements != NULL)
  arr.cap += 5; /* increase capacity */
```

向数组添加元素会增加其大小：

```
if (arr.size < arr.cap) {
  arr.elements[arr.size] = 50;
  arr.size++;
} else {
  printf("Need to expand the array.");
}
```

• 整个程序用 main()编写，用于演示。要正确实现动态数组，应将子任务分解为 init_array()，increase_array()，add_element()和 display_array()等函数。此外以使示例保持简短还跳过了错误检查。

7.文件和异常处理

7.1.使用文件

✧ 访问文件

•可以在 C 程序中打开，读取和写入外部文件。对于这些操作，C 包含用于定义文件流的 **FILE** 类型。**文件流**记录上次读取和写入的位置。

• **stdio.h** 库包含文件处理函数：
• **FILE** Typedef 用于定义文件指针。

• **fopen(filename,mode)**返回文件 *filename* 的 FILE 指针，该文件使用 *mode* 打开。如果无法打开文件，则返回 NULL。
• 模式选项包括：
- **r** 读取（文件必须存在）
- **w** 写入（文件不需要存在）
- **a** 追加（文件不需要存在）
- **r+** 从头读写
- **w+** 读写，覆盖文件
- **a+** 读写，追加到文件尾部

•**fclose(fp)**关闭用 FILE fp 打开的文件，如果关闭成功则返回 0。如果关闭时出错，则返回 **EOF**（文件结尾）。

• 以下程序打开一个文件进行写入，然后将其关闭：

```
#include <stdio.h>

int main() {
   FILE *fptr;

   fptr = fopen("myfile.txt", "w");
   if (fptr == NULL) {
      printf("Error opening file.");
      return -1;
   }
   fclose(fptr);
   return 0;
}
```

• 当字符串文字用于指定文件名时，转义序列\\表示单个反斜杠。在此程序中，如果打开文件时出错，则会向系统返回**-1** 异常代码。异常处理将在以后的课程中解释。
完成使用后关闭文件是一种很好的编程习惯。

✧ 从文件中读取

• **stdio.h** 库还包括从打开的文件中读取的函数。文件可以一次读取一个字符，也可以将整个字符串读入字符**缓冲区**，该缓冲区通常是用于临时存储的字符数组。

• **fgetc(fp)**返回 fp 指向的文件中的下一个字符。如果已到达文件末尾，则返回 **EOF**。

• **fgets(buff,n,fp)**从 *fp* 指向的文件中读取 n-1 个字符，并将字符串存储在 buff 中。NULL 字符'\ 0'被附加为 *buff* 中的最后一个字符。如果 fgets 在达到 n-1 个字符之前遇到换行符或文件结尾，则只有到该位置的字符才会存储在 buff 中。

• **fscanf(fp,conversion_specifiers,vars)** 从 *fp* 指向的文件中读取字符，并使用 *conversion_specifiers* 将输入分配给变量指针变量 *vars*。与 scanf 一样，当遇到空格或换行符时，fscanf 会停止读取字符串。

• 以下程序演示了从文件中读取：

```
#include <stdio.h>

int main() {
   FILE *fptr;
   int c, stock;
   char buffer[200], item[10];
   float price;

   /* myfile.txt: Inventory\n100 Widget 0.29\nEnd of List */

   fptr = fopen("myfile.txt", "r");

   fgets(buffer, 20, fptr);       /* read a line */
   printf("%s\n", buffer);

   fscanf(fptr, "%d%s%f", &stock, item, &price); /* read data */
   printf("%d    %s    %4.2f\n", stock, item, price);

   while ((c = getc(fptr)) != EOF) /* read the rest of the file */
      printf("%c", c);

   fclose(fptr);
   return 0;
}
```

• **gets()**函数读取直到换行符。**fscanf()**根据转换说明符读取数据。然后 while 循环一次读取一个字符，直到文件结束。在打开文件（NULL 指针）时检查问题是为了缩短示例。

✧ 写入文件

• **stdio.h** 库还包括写入文件的函数。写入文件时，必须明确添加换行符'\ n'。

• **fputc(char,fp)**将字符 *char* 写入 *fp* 指向的文件。

• **fputs(str,fp)**将字符串 *str* 写入 *fp* 指向的文件。

• **fprintf(fp,str,vars)**将字符串 *str* 打印到 *fp* 指向的文件。*str* 可以选择包括格式说明符和变量列表变量。

• 以下程序演示了如何写入文件：

```
FILE *fptr;
char filename[50];
printf("Enter the filename of the file to create: ");
gets(filename);
fptr = fopen(filename, "w");

/* write to file */
fprintf(fptr, "Inventory\n");
fprintf(fptr, "%d %s %f\n", 100, "Widget", 0.29);
fputs("End of List", fptr);
```

“**w**”参数定义 **fopen** 函数的“写入模式”。

7.2.二进制文件 I / O

✧ 二进制文件 I/O

• 当您拥有数组或结构时，只将字符和字符串写入文件可能会变得乏味。要将整个内存块写入文件，有以下二进制函数：

• fopen()函数的二进制文件模式选项是：
- **rb** 读取（文件必须存在）
- **wb** 写入（文件不必存在）
- **ab** 追加（文件不必存在）
- **rb+** 从头读写
- **wb+** 读写，覆盖文件
- **ab+** 读写，追加到文件尾部

fwrite(ptr,item_size,num_items,fp)将指针 *ptr* 中的 *item_size* 大小的 *num_items* 项写入文件指针 *fp* 指向的文件。

fread(ptr,item_size,num_items,fp)将文件指针 *fp* 指向的文件中的 *item_size* 大小的 *num_items* 项读入 *ptr* 指向的内存。

fclose(fp)关闭用文件 *fp* 打开的文件，如果关闭成功则返回 0。如果关闭时出错，则返回 **EOF**。
feof(fp)当到达文件流的末尾时返回 0。

✧ 二进制文件 I/O

• 以下程序演示了对二进制文件的写入和读取：

```
FILE *fptr;
int arr[10];
int x[10];
int k;

/* generate array of numbers */
for (k = 0; k < 10; k++)
   arr[k] = k;

/* write array to file */
fptr = fopen("datafile.bin", "wb");
fwrite(arr, sizeof(arr[0]), sizeof(arr)/sizeof(arr[0]), fptr);
fclose(fptr);

/* read array from file */
fptr = fopen("datafile.bin", "rb");
fread(x, sizeof(arr[0]), sizeof(arr)/sizeof(arr[0]), fptr);
fclose(fptr);

/* print array */
for (k = 0; k < 10; k++)
   printf("%d", x[k]);
```

• 这个程序为文件写了一个 int 数组，但是一个结构数组可以很容易地写入文件。请注意，项目大小和项目数是通过使用元素的大小和整个变量的大小来确定的。

• 仅文件扩展名不能确定文件中数据的格式，但它们对于指示期望的数据类型很有用。例如，**.txt** 扩展名表示文本文件，**.bin** 表示二进制数据，**.csv** 表示逗号分隔值，**.dat** 表示数据文件。

✧ 控制文件指针

• stdio.h 中有一些函数用于控制文件指针在二进制文件中的位置：
ftell(fp)返回与文件开头的字节数相对应的 *fp* 文件指针位置的 long int 值。

fseek(fp,num_bytes,from_pos)将 *fp* 文件指针位置相对于位置 *from_pos* 移动 *num_bytes* 个字节，该位置可以是以下常量之一：
- **SEEK_SET** 文件开头

- **SEEK_CUR** 当前位置
- **SEEK_END** 文件结尾

• 以下程序从结构文件中读取记录：

```
typedef struct {
   int id;
   char name[20];
} item;

int main() {
   FILE *fptr;
   item first, second, secondf;

   /* create records */
   first.id = 10276;
   strcpy(first.name, "Widget");
   second.id = 11786;
   strcpy(second.name, "Gadget");

   /* write records to a file */
   fptr = fopen("info.dat", "wb");
   fwrite(&first, 1, sizeof(first), fptr);
   fwrite(&second, 1, sizeof(second), fptr);
   fclose(fptr);

   /* file contains 2 records of type item */
   fptr = fopen("info.dat", "rb");

   /* seek second record */
   fseek(fptr, 1*sizeof(item), SEEK_SET);
   fread(&secondf, 1, sizeof(item), fptr);
   printf("%d    %s\n", secondf.id, secondf.name);
   fclose(fptr);
   return 0;
}
```

• 该程序将两个项目记录写入文件。为了只读取第二条记录，**fseek()**将文件指针从文件的开头移动到 1*sizeof(item)字节。例如，如果要将指针移动到第四个记录，则从文件的开头(SEEK_SET)寻找 **3 * sizeof(item)**。

7.3.异常处理

✧ 异常处理

• 良好的编程实践的核心是使用错误处理技术。如果您忘记包含异常处理，即使是再熟练的编码水平可能也无法阻止程序崩溃。

• 任何导致程序停止正常执行的情况都是**异常**。**异常处理**（也称为**错误处理**）是一种处理运行时错误的方法。

• C 没有明确支持异常处理，但有一些方法可以管理错误：
- 编写代码以防止错误。您无法控制用户输入，但可以检查以确保用户输入了有效输入。执行除法时，请采取额外步骤以确保不会出现**除以 0**。
- 使用 **exit** 语句优雅地结束程序执行。您可能无法控制文件是否可读，但您不需要让问题导致程序崩溃。
使用 **errno**，**perror()**和 **strerror()**通过错误代码识别错误。

✧ 退出命令

• **exit** 命令立即停止执行程序并将退出代码发送回调用进程。例如，如果程序被另一个程序调用，则调用程序可能需要知道退出状态。

• 使用 **exit** 来避免程序崩溃是一种很好的做法，因为它会关闭所有打开的文件连接和进程。

• 您可以通过 exit 语句返回任何值，但是 **0** 表示成功，**-1** 表示失败。预定义的 stdlib.h 宏 **EXIT_SUCCESS** 和 **EXIT_FAILURE** 也是常用的。
例如：

```
int x = 10;
int y = 0;

if (y != 0)
   printf("x / y = %d", x/y);
else {
   printf("Divisor is 0. Program exiting.");
   exit(EXIT_FAILURE);
}
```

7.4.使用 Error 代码

✧ 使用 errno

• 某些库函数（如 fopen()）在未按预期执行时会设置错误代码。错误代码在名为 **errno** 的全局变量中设置，该变量在 **errno.h** 头文件中定义。使用 errno 时，应在调用库函数之前将其设置为 0。

• 要输出存储在 errno 中的错误代码，可以使用 fprintf 打印到 **strerr** 文件流，将标准错误输出到屏幕。使用 strerr 是一个常规问题和良好的编程习惯。

• 您可以通过其他方式输出 errno，但如果仅使用 strerr 来显示错误消息，则更容易跟踪异常处理。

•要使用 errno，需要使用语句 **extern int errno** 声明它；在程序的顶部（或者您可以包含 **errno.h** 头文件）。
例如：

```
#include <stdio.h>
#include <stdlib.h>
// #include <errno.h>

extern int errno;

int main() {
   FILE *fptr;
   int c;

   errno = 0;
   fptr = fopen("c:\\nonexistantfile.txt", "r");
   if (fptr == NULL) {
      fprintf(stderr, "Error opening file. Error code: %d\n", errno);
      exit(EXIT_FAILURE);
   }

   fclose(fptr);
   return 0;
}
```

✧ perror 和 strerror 函数

• 当库函数设置 errno 时，会分配一个神秘的错误编号。有关错误的更具描述性的消息，您可以使用 **perror()**。您还可以使用 **string.h** 头文件中的 **strerror()**获取消息，该文件返回指向消息文本的指针。

•**perror()**必须包含一个在实际错误消息之前的字符串。通常，对于同一错误，不需要 perror() 和 strerror()，但是为了演示目的，两者都在下面的程序中使用：

```
FILE *fptr;
errno = 0;

fptr = fopen("c:\\nonexistantfile.txt", "r");
if (fptr == NULL) {
   perror("Error");
   fprintf(stderr, "%s\n", strerror(errno));
   exit(EXIT_FAILURE);
}
```

• 有超过一百个**错误代码**。使用这些语句列出它们：

```
for (int x = 0; x < 135; x++)
   fprintf(stderr, "%d: %s\n", x, strerror(x));
```

✧ EDOM 和 ERANGE 错误代码

• 当域超出范围时，**math.h** 库中的某些数学函数将 **errno** 设置为定义的宏值 **EDOM**。
• 同样，当出现范围错误时，将使用 **ERANGE** 宏值。
例如：

```
float k = -5;
float num = 1000;
float result;

errno = 0;
result = sqrt(k);
if (errno == 0)
   printf("%f ", result);
else if (errno == EDOM)
   fprintf(stderr, "%s\n", strerror(errno));

errno = 0;
```

```
result = exp(num);
if (errno == 0)
   printf("%f ", result);
else if (errno == ERANGE)
   fprintf(stderr, "%s\n", strerror(errno));
```

✧ feof 和 ferror 函数

• 除了检查 NULL 文件指针和使用 errno 之外，**feof()**和 **ferror()**函数还可用于确定文件 I / O 错误：
• **feof(fp)**如果已到达流的末尾，则返回非零值，否则返回 0。feof 也设置为 EOF。
• **ferror(fp)**如果有错误则返回非零值，0 表示无错误。

• 以下程序包含几种异常处理技术：

```
FILE *fptr;
int c;

errno = 0;

fptr = fopen("myfile.txt", "r");
if (fptr == NULL) {
   fprintf(stderr, "Error opening file. %s\n", strerror(errno));
   exit(EXIT_FAILURE);
}

while ((c = getc(fptr)) != EOF) /* read the rest of the file */
   printf("%c", c);

if (ferror(fptr)) {
   printf("I/O error reading file.");
   exit(EXIT_FAILURE);
}
else if (feof(fptr)) {
   printf("End of file reached.");
}
```

• 程序输出会有所不同。但是，如果文件正确打开并且程序完成读取整个文件，则会显示以下消息：“文件结束”。

8.预处理器

8.1.预处理器指令

✧ 预处理器指令

• C **预处理器**使用**#指令**在编译之前在程序源代码中进行替换。
例如，在编译程序之前，行**#include <stdio.h>**将替换为 stdio.h 头文件的内容。

• 预处理程序指令及其用途：
#include 包括头文件。
#define，**#undef** 定义和取消定义宏。
#ifdef，**#ifnf**，**#if**，**#else**，**#elif**，**#endif** 条件编译。
#pragma 特定于实现和编译器。
#error,#warning 输出错误或警告消息错误将停止编译。

不要在#指令的末尾加上分号。

✧ #include 指令

• #include 指令用于在程序中包含**头文件**。头文件声明了**库**的函数和宏的集合，这个术语来自可以重用代码集合的方式。

• 一些有用的 C 库是：
stdio 输入/输出功能，包括 printf 和文件操作。
stdlib 内存管理和其他实用程序
string 用于处理字符串的函数
errno errno 全局变量和错误代码宏
math 常用的数学函数
time 时间/日期工具

• 库的相应头文件按惯例以**.h** 结尾。如果应在编译器包含路径中搜索文件，则#include 指令需要在头文件名周围使用括号<>。

• **用户定义的头文件**也被赋予.h 扩展名，但引用了引号，如“myutil.h”中所示。 使用引号时，将在源代码目录中搜索该文件。
例如：

```
#include <stdio.h>
#include “myutil.h”
```

一些开发人员对头文件使用.hpp 扩展名。

✧ #define 指令

• **#define** 指令用于基于值或表达式为常量创建类似对象的宏。
• #define 也可用于创建类似函数的宏，其参数将被预处理器替换。

• 对类似函数的定义要谨慎。请记住，预处理器可以直接替换而无需任何计算，这可能会导致意外结果，如以下程序所示：

```
#include <stdio.h>
#define PI 3.14
#define AREA(r) (PI*r*r)

int main() {
   float radius = 2;
   printf("%3.2f\n", PI);
   printf("Area is %5.2f\n", AREA(radius));
   printf("Area with radius + 1: %5.2f\n", AREA(radius+1));
   return 0;
}
```

• 在编译之前，预处理器会扩展每个宏标识符。在这种情况下，每次出现的 **PI** 都被 3.14 替换，**AREA(arg)**被替换为表达式 PI*arg*arg。发送给编译器的最终代码已经具有常量值。

• 但是，如果您认为#define 严格地通过替换文本来工作，那将不是我们所期望的！您将看到 AREA(radius+1)变为 PI *radius+ 1*radius+ 1，即 3.14*2+1*2+1。

解决方案是将每个参数括在括号中以获得正确的操作顺序。
例如：

```
#define AREA(r) (PI*(r)*(r))
```

代码生成输出：Area with radius+1:28.26

✧ 格式化预处理指令

• 使用预处理程序指令时，#必须是一行中的第一个字符。但是#之前和#和指令之间可以有任何数量的空格。

• 如果#指令很长，则可以使用**\ 延续字符**将定义扩展到多行。

例如：

```
#define VERY_LONG_CONSTANT \
23.678901

#define MAX 100
#define MIN 0
#       define SQUARE(x) \
x*x
```

#之前和#和指令之间可以有任意数量的空格。

✧ 预定义的宏指令

•除了定义自己的宏之外，还有几个标准的**预定义宏**在 C 程序中始终可用，而不需要#define 指令：

__DATE__当前日期为字符串，格式为 Mmm dd yyyy
__TIME__当前时间为字符串，格式为 hh:mm:ss
__FILE__当前文件名为字符串
__LINE__当前行号作为 int 值
__STDC__ 1

例如：

```
char curr_time[10];
char curr_date[12];
int std_c;

strcpy(curr_time, __TIME__);
strcpy(curr_date, __DATE__);
printf("%s %s\n", curr_time, curr_date);
printf("This is line %d\n", __LINE__);
std_c = __STDC__;
printf("STDC is %d", std_c);
```

8.2.条件编译指令

✧ The #ifdef, #ifndef,和#undef 指令

• **#ifdef**，**#ifndef** 和**#undef** 指令对使用**#define** 创建的宏进行操作。
例如，如果定义了两次相同的宏，则会出现编译问题，因此可以使用**#ifdef** 指令进行检查。或者，如果您可能想要重新定义宏，则首先使用**#undef**。

下面的程序演示了这些指令：

```
#include <stdio.h>

#define RATE 0.08
#ifndef TERM
   #define TERM 24
#endif

int main() {
   #ifdef RATE    /* this branch will be compiled */
      #undef RATE
      printf("Redefining RATE\n");
      #define RATE 0.068
   #else    /* this branch will not be compiled */
      #define RATE 0.068
   #endif

   printf("%f    %d\n", RATE, TERM);

   return 0;
}
```

• 因为 RATE 是在顶部定义的，所以只编译#ifdef 子句。在预处理期间#ifdef RATE 为 false 时，可选的#else 分支被编译。
• 需要**#endif** 来关闭代码块。
#elif 指令就像一个 else，可用于在#else 之后提供额外的替代方案。

✧ 条件编译指令

• 代码段的条件编译由一组指令控制：**#if,#else,#elif** 和**#endif**。
例如：

```
#define LEVEL 4

int main() {
   #if LEVEL > 6
      /* do something */
   #elif LEVEL > 5
      /* else if branch */
   #elif LEVEL > 4
      /* another else if */
   #else
      /* last option here */
```

```
  #endif

  return 0;
}
```

• 有些情况下这种条件编译可能很有用，但应谨慎使用这种类型的代码。
• **defined()**预处理程序运算符可以与#if 一起使用，如下所示：

```
#if !defined(LEVEL)
  /* statements */
#endif
```

• **#if** 和 if 语句不可互换。#if 使用预处理器可用的数据进行评估，然后预处理器只发送真正的分支进行编译。
if 语句使用在运行时提供的数据，可以分支到任何 else 子句。

8.3.预处理运算符

✧ 预处理运算符

• C 预处理器提供以下运算符。
#运算符
#macro 运算符称为**字符串化**运算符，它告诉预处理器将参数转换为字符串常量。
参数两侧的空格被忽略，并且可以识别转义序列。

例如：

```
#define TO_STR(x) #x

printf("%s\n", TO_STR( 123\\12 ));
```

✧ ##运算符

• **##运算符**也称为**标记粘贴**运算符，因为它将标记附加或“粘贴”在一起。
例如：

```
#define VAR(name, num) name##num

int x1 = 125;
int x2 = 250;
int x3 = 500;

printf("%d\n", VAR(x, 3));
```

CERTIFICATE

SOLOLEARN
DISCOVER YOUR POTENTIAL

Issued 12 November, 2018

This is to certify that

林庆泓

has successfully completed the

C Tutorial course

Yeva Hyusyan
Chief Executive Officer

Certificate #1089-6684149