C 语言程序设计

卓能文

智能科技学院

2024-03-13

环境安装

1. Windows: msys2: https://www.msys2.org/ https://github.com/msys2/msys2-installer/releases/download/2024-01-13/msys2-x86_64-20240113.exe
2. visual studio code: https://code.visualstudio.com/ 下载并安装合适版本

编辑、编译、运行

```c
#include <stdio.h>

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

注释

在 C 语言中，注释语句用于对代码进行解释和说明，可以提高代码的可读性和可维护性，不会被编译器执行。C 语言有两种注释方式：

单行注释：使用//开头，后面的内容直到该行结束都被视为注释。

```
// 这是一个单行注释
int a = 10; // 定义一个整型变量 a 并赋值为 10
```

多行注释：使用/*开头，*/结尾，中间的内容都被视为注释。

```
/*
这是一个多行注释
可以跨越多行
```

注释

```
*/
int b = 20;
```

变量定义

在 C 语言中，变量定义是指为变量分配内存空间并指定其数据类型。变量定义的基本语法如下：

数据类型 变量名;

例如，定义一个整型变量 age，可以这样写：

```
int age;
```

变量赋值

在 C 语言中，变量赋值是指将一个值赋给变量。变量赋值的基本语法如下： 变量名 = 值;

例如，将变量 age 的值赋为 20，可以这样写：

```
age = 20;
```

变量输出

在 C 语言中，输出变量可以使用 printf()函数。例如：

```c
#include <stdio.h>

int main() {
    int num = 10;
    printf("num 的值为: %d", num);
    return 0;
}
```

其中，%d 表示输出整数类型的变量，如果需要输出其他类型的变量，可以使用相应的格式控制符。

数据类型

C 语言中的基本数据类型包括整型、浮点型、字符型等。具体如下：

1. 整型：用于存储整数，如 `int`。
2. 浮点型：用于存储带有小数点的数值，如 `float` 和 `double`。
3. 字符型：用于存储单个字符，如 `char`。
4. 枚举型：用户定义的数据类型，由一组命名的整数常量组成，如 `enum`。
5. 指针型：存储变量地址，如 `int *ptr`。
6. 数组型：存储一系列相同类型的元素，如 `int array[10]`。

数据类型

7. 结构体型：自定义的数据类型，可以包含不同类型的数据项，如 `struct`。
8. 联合体型：也一种自定义的数据类型，但与结构体不同的是，联合体中的所有成员共享同一块内存空间，如 `union`。
9. 在 C 语言编程时，选择合适的数据类型对于程序的效率和正确性至关重要。

整数类型

C 语言中的整数类型有以下几种：

1. `char`：字符类型，通常占用一个字节（8 位），用于表示 ASCII 码中的字符。
2. `short`：短整型，通常占用两个字节，可表示的范围为-32768 32767。
3. `int`：整型，通常占用四个字节，可表示的范围为-2147483648 2147483647。
4. `long`：长整型，占用四个字节或八个字节，可表示范围为-2147483648 2147483647

整数类型

或-9223372036854775808 9223372036854775807，具体取决于编译器。

5. `long long`：长长整型，通常占用八个字节，可表示范围为-9223372036854775808 9223372036854775807。

此外，这些整数类型还可以通过添加 `unsigned` 关键字变成无符号整数类型，表示范围是 0 到最大值。

字符类型

C 语言中的字符类型是指单个字符，使用 char 关键字进行声明。以下是关于字符类型的一些详细信息：

1. 存储大小：字符类型在计算机内部使用一个字节（8 位）存储。
2. 字符表示：字符常量必须放在单引号里面，例如'B'表示字符 B。
3. 数值对应：每个字符对应一个整数（由 ASCII 码确定），例如字符'B'对应整数 66。
4. 数据范围：char 类型是 8 位有符号整数类型，其默认范围取决于编译器和平台。

字符类型

5. 数学运算：两个字符类型可以做数学运算，因为它们本质上是整数。
6. 修饰符：可以使用 `signed` 和 `unsigned` 修饰符来指定字符类型是否有符号。
7. 转义字符：C 语言中有一些特殊的字符称为转义字符，例如换行符`'\n'`和制表符`'\t'`等。

此外，在使用 `char` 类型时，需要注意它与整数类型的区别以及在不同编译器和平台上可能存在的差异。

浮点类型

在 C 语言中，浮点类型主要分为 `float`、`double` 和 `long double`。具体介绍如下：

1. `float`（单精度浮点型）：这是基本的浮点数据类型，它在 C 语言中占用 4 个字节的内存空间。`float` 类型的数值有一个有限的精度，这意味着它们不能精确地表示所有小数。
2. `double`（双精度浮点型）：`double` 类型提供比 `float` 更高的精度，通常占用 8 个字节的内存空间。由于其更高的精度，double 类型常用于需要更高精度计算的情况。在 `printf` 函数中，即使参数是

浮点类型

`float` 类型，使用 `%f` 作为占位符也是正确的，因为 `float` 会自动提升为 `double` 类型。

3. `long double`（长双精度浮点型）：`long double` 类型提供比 `double` 更大的精度和范围，但具体的存储大小和精度取决于编译器。这种类型不常用，但在需要极高精度的数学计算中可能会用到。

此外，在使用浮点数时，需要注意浮点数的比较不应使用等号==，而应该使用一个足够小的正数来表示两个浮点数之间的最大允许差值。这是因为浮点数的表示可能不完全精确，直接比较可能会得到意外的结果。

浮点类型

总的来说，C 语言中的浮点类型主要用于处理实数，包括小数和很大的数。选择合适的浮点类型对于科学计算和工程应用尤为重要，以确保计算结果的准确性和可靠性。

条件语句

在 C 语言中，条件语句用于根据指定的条件执行不同的代码块。条件语句的基本语法如下：

```
if (表达式) {
    // 如果表达式的值为真，则执行这里的代码块
}
```

例如，定义一个整型变量 age，并将其赋值为 20，然后使用条件语句判断 age 是否大于 18：

```
int age = 20;

if (age > 18) {
```

条件语句

```
    printf("你已经成年了！\n");
}
```

在 C 语言中，条件语句通常使用 `if`、`else if` 和 `else` 关键字。例如：

```
#include <stdio.h>

int main() {
    int num = 10;
    if (num > 5) {
        printf("num 大于 5");
    } else if (num == 5) {
```

条件语句

```
        printf("num 等于 5");
    } else {
        printf("num 小于 5");
    }
    return 0;
}
```

在这个例子中，程序首先判断 num 是否大于 5，如果是则输出“num 大于 5”，否则继续判断 num 是否等于 5，如果是则输出“num 等于 5”，否则输出“num 小于 5”。

while 语句

在 C 语言中，循环语句用于重复执行一段代码。循环语句的基本语法如下：

```
while (表达式) {
    // 重复执行这里的代码块
}
```

例如，定义一个整型变量 i，并将其赋值为 0，然后使用循环语句重复执行代码块，直到 i 大于等于 10：

```
int i = 0;

while (i < 10) {
```

while 语句

```
        printf("i 的值为: %d\n", i);
        i++;
    }
```

for 语句

在 C 语言中，for 语句用于循环执行一段代码。它的基本语法如下：

```
for (初始化表达式;  条件表达式;  更新表达式) {
    // 循环体
}
```

其中，初始化表达式用于设置循环变量的初始值；条件表达式用于判断是否继续执行循环体；更新表达式用于更新循环变量的值。例如，下面的代码使用 for 语句输出 1 到 10 的数字：

```
#include <stdio.h>

int main() {
```

for 语句

```
    for (int i = 1; i <= 10; i++) {
        printf("%d ", i);
    }
    return 0;
}
```

在这个例子中，i 是循环变量，初始值为 1，每次循环后增加 1，直到 i 大于 10 时停止循环。

break 语句

在 C 语言中，break 语句用于跳出循环或者 switch 语句。当程序执行到 break 语句时，会立即退出当前所在的循环或 switch 语句，继续执行后面的代码。例如：

```c
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            break;
        }
        printf("%d ", i);
    }
```

break 语句

```
    return 0;
}
```

这个例子中，程序使用 for 语句输出 0 到 9 的数字。当 i 等于 5 时，程序执行到 break 语句，立即退出循环，不再输出数字。因此，程序只输出 0 到 4 的数字。

continue 语句

在 C 语言中，continue 语句用于跳过当前循环的剩余部分，直接进入下一次循环。当程序执行到 continue 语句时，会立即跳过当前循环体中 continue 后面的代码，并开始下一次循环。例如：

```c
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; i++) {
        if (i % 2 == 0) {
            continue;
        }
        printf("%d ", i);
    }
```

continue 语句

```
    return 0;
}
```

这个例子中，程序使用 for 语句输出 0 到 9 的数字。当 i 是偶数时，程序执行到 continue 语句，跳过本次循环的剩余部分，直接进入下一次循环。因此，程序只输出奇数 0 到 9 的数字。

switch 语句

在 C 语言中，switch 语句用于根据表达式的值选择执行不同的代码块。它的基本语法如下：

```
switch (表达式) {
    case 常量 1:
        // 当表达式的值等于常量 1 时执行的代码
        break;
    case 常量 2:
        // 当表达式的值等于常量 2 时执行的代码
        break;
    ...
    default:
```

switch 语句

```
        // 当表达式的值不等于任何常量时执行的代码
}
```

其中，case 关键字后面跟着一个常量，表示当表达式的值等于该常量时要执行的代码。break 关键字用于跳出 switch 语句，如果没有 break，程序会继续执行下一个 case 的代码。default 关键字表示当表达式的值不等于任何常量时要执行的代码。例如：

```c
#include <stdio.h>

int main() {
    int num = 2;
    switch (num) {
```

switch 语句

```
        case 1:
            printf("num 等于 1");
            break;
        case 2:
            printf("num 等于 2");
            break;
        case 3:
            printf("num 等于 3");
            break;
        default:
            printf("num 不等于 1、2、3");
    }
```

switch 语句

```
    return 0;
}
```

函数

在 C 语言中，函数是指一组语句的集合，用于完成特定的任务。函数的基本语法如下：

```
返回值类型 函数名(参数列表) {
    // 函数体
    return 返回值;
}
```

例如，定义一个函数 add，用于计算两个整数的和，可以这样写：

```
int add(int a, int b) {
    int sum = a + b;
```

```
    return sum;
}
```

在 C 语言中，函数调用是指执行一个函数的过程。以下是关于函数调用的一些关键概念：

1. 函数声明与定义：
   - 函数声明是告诉编译器函数的名称、返回类型和参数列表。
   - 函数定义是函数的实际实现，包括函数体。
2. 调用函数：
   - 使用函数名和传递的参数（如果有的话）来调用函数。例如，如果有一个名为 `add` 的函数，可以通过 `add(3, 4);` 来调用它。

函数

3. 参数传递：
   - C 语言默认使用值传递，即传递的是参数的副本，而不是原始数据。这意味着函数内部对参数的修改不会影响到外部变量。
4. 返回值：
   - 函数可以有返回值，也可以没有。返回值的类型应与函数声明中的返回类型匹配。
5. 函数原型：
   - 在调用函数之前，需要确保编译器知道函数的原型，即函数的返回类型、名称和参数列表。这可以通过包含头文件或在代码中直接声明函数原型来实现。

函数

6. 调用约定：
   - C 语言有特定的调用约定，用于确定如何将参数传递给函数以及如何从函数获取返回值。常见的调用约定包括 `cdecl`、`stdcall` 等。
7. 递归调用：
   - 函数可以调用自身，这称为递归调用。递归调用通常用于解决可以分解为更小相似问题的问题。
8. 函数指针：
   - 函数指针是指向函数的指针，可以用于间接调用函数。函数指针的类型应与被指向的函数的类型相匹配。

函数

通过以上介绍，可以看出函数调用在 C 语言编程中扮演着非常重要的角色，它允许将程序分解为可重用的代码块，提高了代码的模块化和可维护性。

```c
#include <stdio.h>

int main() {
    int a = 3, b = 4;
    int c = add(a, b);
    printf("结果为: %d", c);
}
```

指针

在 C 语言中，指针是指向内存地址的变量。指针的基本语法如下：

数据类型 *变量名;

例如，定义一个指向整型变量的指针 p，可以这样写：

```
int num = 10;
int *p = &num;
```

在 C 语言中，指针是一种特殊的变量，它存储着另一个变量的内存地址。通过指针，我们可以间接地访问和操作内存中的数据。以下是关于指针访问的一些基本信息：

指针

1. 声明指针：要声明一个指针，需要指定指针的类型（即它指向的变量的类型）和指针的名字。例如，`int *ptr;` 声明了一个指向整数的指针。
2. 初始化指针：可以在声明指针时对其进行初始化，即将一个变量的地址赋给指针。例如，`int num = 10; int *ptr = &num;`将变量 num 的地址赋给了指针 ptr。
3. 访问指针指向的值：使用 * 运算符可以访问指针指向的变量的值。例如，`int value = *ptr;`将指针 ptr 指向的值赋给了变量 value。

指针

4. 指针运算：指针可以进行加减运算，但这种运算是根据指针指向的数据类型的大小来进行的。例如，如果 `ptr` 是一个指向整数的指针，那么 `ptr++`将会使 `ptr` 指向下一个整数的位置。
5. 指针与数组：指针与数组紧密相关，数组名可以被当作指向数组第一个元素的指针。因此，可以使用指针来遍历数组。
6. 指针与函数：指针可以作为函数参数，这样函数就可以修改调用者的变量。此外，指针还可以用于动态内存分配，如 `malloc` 和 `calloc` 函数。

指针

总的来说，指针是 C 语言中的一个强大的特性，它提供了对内存的直接访问能力，允许程序员进行灵活的内存管理和数据操作。然而，指针的使用也需要谨慎，不当的使用可能导致程序错误或内存泄露。

数组

在 C 语言中，数组是指一组相同数据类型的元素的集合。数组的基本语法如下：

数据类型 数组名[元素个数];

例如，定义一个包含 5 个整型元素的数组 numbers，可以这样写：

```
int numbers[5];
```

在 C 语言中，数组是一种数据结构，用于存储相同类型的多个元素。以下是关于数组访问的一些关键概念：

1. 声明和定义：

数组

- 使用 `type arrayName[size];`的语法来声明一个数组，其中 `type` 是数组元素的类型，`arrayName` 是数组的名称，`size` 是数组的大小（即元素数量）。
- 例如，声明一个大小为 5 的整数数组可以使用 `int arr[5];`。

2. 初始化：
   - 可以在声明数组时进行初始化，也可以在声明后单独进行初始化。
   - 例如，可以在声明时初始化数组 `int arr[5] = {1, 2, 3, 4, 5};`，也可以在声明后初始化数组 `arr[0] = 1; arr[1] = 2;`等。
3. 访问数组元素：

数组

- 通过索引访问数组元素，索引从 0 开始。
- 例如，要访问数组的第一个元素，可以使用 `arr[0]`，要访问第二个元素，可以使用 `arr[1]`。

4. 遍历数组：
   - 可以使用循环结构（如 `for` 循环）来遍历数组的所有元素。
   - 例如，可以使用以下代码遍历数组并打印每个元素：

```
for (int i = 0; i < size; i++) {
    printf("%d ", arr[i]);
}
```

5. 多维数组：

数组

- C 语言支持多维数组，即数组的元素也是数组。
- 例如，可以声明一个二维整数数组 `int arr[3][4];`，表示一个 3 行 4 列的矩阵。

6. 指针与数组：
   - 数组名实际上是指向数组第一个元素的指针常量。
   - 可以通过指针间接访问数组元素。
   - 例如，可以使用以下代码通过指针访问数组元素：

```c
int *ptr = &arr[0]; // 指向数组第一个元素的指针
printf("%d ", *ptr); // 输出第一个元素
```

数组

综上所述，数组是 C 语言中一种非常有用的数据结构，它能够高效地存储和访问多个相同类型的元素。掌握数组的访问方法对于编写高效的 C 语言程序至关重要。

枚举类型

C 语言中的枚举类型是一种用户定义的数据类型，用于为一组整数值赋予有意义的名字。

枚举类型的主要特点包括：

1. 定义方式：使用 enum 关键字来定义枚举类型，后面跟一对花括号，其中列出了枚举的成员，每个成员都是一个整数值。例如，`enum Color {RED, GREEN, BLUE};`定义了一个名为 Color 的枚举类型，包含三个成员：RED、GREEN 和 BLUE。

枚举类型

2. 值的范围：枚举成员的值默认从 0 开始，依次递增。也可以手动为枚举成员赋值，如 `enum Color {RED=1, GREEN, BLUE};`中，RED 的值为 1，GREEN 为 2，BLUE 为 3。
3. 内存分配：枚举类型的大小取决于它所包含的成员的数量，通常情况下，一个枚举变量的大小与其表示的整数值的大小相同。
4. 用法：枚举类型主要用于提高代码的可读性和可维护性。在程序中使用枚举变量时，可以更加直观地表达意图，而不是使用抽象的数字。

枚举类型

5. 与宏的区别：虽然枚举和宏都可以用于创建常量集合，但枚举在编译阶段将名字替换成对应的值，而宏在预处理阶段进行名称替换。

综上所述，枚举类型是 C 语言中一种非常有用的数据类型，它不仅能够限定变量的取值范围，还能够提高代码的可读性和可维护性。

结构体

在 C 语言中，结构体是指一组不同数据类型的元素的集合。结构体的基本语法如下：

```
struct 结构体名 {
    成员 1 类型  成员 1 名;
    成员 2 类型  成员 2 名;
    ...
};
```

1. 定义结构体：首先，需要定义一个结构体来表示学生的信息。例如，一个学生可能包含姓名、学号和成绩等信息。可以这样定义：

结构体

```
struct Student {
    char name[50];
    int id;
    float grade;
};
```

2. 创建结构体变量：定义了结构体之后，就可以创建结构体变量来存储具体的学生信息。例如：

```
struct Student student1;
```

3. 访问结构体成员：可以通过.运算符来访问结构体的成员。例如，可以这样为 student1 赋值：

结构体

```
strcpy(student1.name, "张三");
student1.id = 12345;
student1.grade = 89.5;
```

4. 使用结构体数组：如果需要存储多个学生的信息，可以使用结构体数组。例如，可以创建一个包含 100 个学生的数组：

```
struct Student students[100];
```

5. 传递结构体给函数：可以将结构体作为参数传递给函数。例如，可以定义一个函数来打印学生的信息：

```
void printStudentInfo(struct Student student) {
    printf("Name: %s", student.name);
```

```
    printf("ID: %d", student.id);
    printf("Grade: %.2f", student.grade);
}
```

6. 动态分配结构体：可以使用 malloc 或 calloc 函数来动态分配结构体的内存。例如，可以这样创建一个动态的学生数组：

```
struct Student *students = (struct Student *)malloc(100 *
sizeof(struct Student));
```

通过以上介绍，可以看出结构体是 C 语言中一种非常有用的特性，它允许将相关的数据组合在一起，提高了代码的可读性和可维护性。

联合体

在 C 语言中，union（联合体）是一种数据类型，允许在同一块内存空间中存储不同类型的数据。以下是一个简单的示例：

```c
#include <stdio.h>

union Data {
    int i;
    float f;
    char str[20];
};

int main() {
    union Data data;
```

联合体

```c
// 使用联合体的整型成员
data.i = 10;
printf("data.i: %d", data.i);

// 使用联合体的浮点型成员
data.f = 3.14;
printf("data.f: %.2f", data.f);

// 使用联合体的字符数组成员
strcpy(data.str, "Hello, world!");
printf("data.str: %s", data.str);
```

```
    return 0;
}
```

在上面的示例中，我们定义了一个名为 Data 的联合体，它包含了一个整数、一个浮点数和一个字符数组。然后，我们在 main 函数中使用这个联合体来存储不同类型的数据。需要注意的是，由于联合体的所有成员共享同一块内存空间，所以只能同时访问其中的一个成员。

当我们将整数赋值给 data.i 时，浮点数和字符数组的值将被覆盖。同样地，当我们将浮点数赋值给 data.f 时，整数和字符数组的值将

联合体

被覆盖。因此，在使用联合体时需要小心，确保不会意外地覆盖其他成员的值。

指针和数组

在 C 语言中，指针和数组是两种不同的数据类型，但它们之间存在一些相似之处。指针和数组都可以用来存储多个相同数据类型的元素。指针和数组都可以通过下标来访问其中的元素。

例如，定义一个包含 5 个整型元素的数组 numbers，可以这样写：

```c
int numbers[5];
int *p = numbers;
printf("%d\n", *(p+2)); // 输出第 3 个元素的值
printf("%d\n", *(numbers+2)); // 输出第 3 个元素的值
```

函数和指针

在 C 语言中，函数和指针是两种不同的概念，但它们之间存在一些相似之处。函数可以作为参数传递给其他函数，也可以作为返回值返回。指针可以指向函数，也可以通过函数指针调用函数。

例如，定义一个函数 add，用于计算两个整数的和，可以这样写：

```
int add(int a, int b) {
    return a + b;
}
```

定义一个指向函数的指针 p，可以这样写：

函数和指针

```c
int (*p)(int, int);
p = add;
printf("%d\n", p(2, 3)); // 输出 5
```

函数和数组

在 C 语言中，函数和数组是两种不同的概念，但它们之间存在一些相似之处。函数可以作为参数传递给其他函数，也可以作为返回值返回。数组可以作为参数传递给其他函数，也可以作为返回值返回。

例如，定义一个函数 `printArray`，用于打印数组中的元素，可以这样写：

```c
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

函数和数组

定义一个函数 sumArray，用于计算数组中的元素之和，可以这样写：

```c
int sumArray(int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum;
}
```

字符串

在 C 语言中，字符串是指一组由空字符'\0'分隔的字符。字符串的基本语法如下：

```
char 数组名[字符个数 + 1];
```

例如，定义一个包含 5 个字符的字符串 str，可以这样写：

```
char str[6] = "Hello";
```

1. 声明和初始化字符串：可以使用字符数组来声明和初始化字符串。例如：

   ```
   char str1[20] = "Hello, world!";
   char str2[] = "Hello, world!";
   ```

字符串

2. 获取字符串长度：可以使用 strlen 函数来获取字符串的长度（不包括空字符）。例如：

```
char str[] = "Hello, world!";
int length = strlen(str);
printf("Length: %d", length);
```

3. 连接字符串：可以使用 strcat 函数来连接两个字符串。例如：

```
char str1[20] = "Hello";
char str2[] = ", world!";
strcat(str1, str2);
printf("Concatenated string: %s", str1);
```

字符串

4. 比较字符串：可以使用 strcmp 函数来比较两个字符串。例如：

```c
char str1[] = "apple";
char str2[] = "banana";
int result = strcmp(str1, str2);
if (result < 0) {
    printf("%s comes before %s", str1, str2);
} else if (result > 0) {
    printf("%s comes after %s", str1, str2);
} else {
    printf("%s and %s are equal", str1, str2);
}
```

5. 复制字符串：可以使用 strcpy 函数来复制一个字符串。例如：

字符串

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

int main() {
    char source[] = "Hello, world!";
    char destination[20];

    // 使用 strcpy 函数复制字符串
    strcpy(destination, source);

    printf("Source string: %s", source);
    printf("Destination string: %s", destination);
```

```
    return 0;
}
```

在上面的示例中，我们声明了两个字符数组 source 和 destination。然后，我们使用 strcpy 函数将 source 字符串复制到 destination 字符串中。最后，我们打印出源字符串和目标字符串的内容。

需要注意的是，strcpy 函数不会检查目标字符串的大小，因此在使用它时需要确保目标字符串有足够的空间来存储源字符串的内容。否则，可能会导致缓冲区溢出错误。为了避免这种情况，可以使用 strncpy 函数，它可以指定要复制的最大字符数。

字符串

6. 查找字符串：可以使用 strchr 函数来查找字符串中的特定字符或字符串。例如：

```
char str[] = "Hello, world!";
char target = 'o';

// 使用 strchr 函数查找目标字符
char *result = strchr(str, target);

if (result != NULL) {
    printf("Character '%c' found at position %ld", target,
result - str);
} else {
```

```
        printf("Character '%c' not found", target);
}
```

在上面的示例中，我们声明了一个字符数组 str 和一个目标字符 target。然后，我们使用 strchr 函数在 str 中查找 target 字符的第一次出现位置。如果找到了目标字符，strchr 函数将返回一个指向该字符的指针；否则，它将返回 NULL。最后，我们根据查找结果打印相应的信息。

需要注意的是，strchr 函数是区分大小写的，因此如果要进行不区分大小写的查找，可以使用 strcasestr 函数。

字符串

7. 查找子串：可以使用 strstr 函数来查找一个字符串是否包含另一个字符串。例如：

```
char str[] = "Hello, world!";
char substr[] = "world";
char *result = strstr(str, substr);
if (result != NULL) {
    printf("Substring found at position %ld", result -
str);
} else {
    printf("Substring not found");
}
```

字符串

8. 分割字符串：可以使用 strtok 函数来分割字符串。例如：

```
char str[] = "apple,banana,orange";
char delimiter[] = ",";
char *token = strtok(str, delimiter);
while (token != NULL) {
    printf("%s", token);
    token = strtok(NULL, delimiter);
}
```

9. 格式化字符串：可以使用 sprintf 函数来格式化字符串。例如：

```
char str[50];
int num = 42;
```

字符串

```c
float pi = 3.14159;

// 使用 sprintf 函数将数据写入字符串
sprintf(str, "The answer is %d and the value of pi is %.
2f", num, pi);

printf("Formatted string: %s", str);
```

在上面的示例中，我们声明了一个字符数组 str 和两个变量 num 和 pi。然后，我们使用 sprintf 函数将 num 和 pi 的值以指定的格式写入 str 字符串中。最后，我们打印出格式化后的字符串。

需要注意的是，`sprintf` 函数不会检查目标字符串的大小，因此在使用它时需要确保目标字符串有足够的空间来存储格式化后的数据。否则，可能会导致缓冲区溢出错误。为了避免这种情况，可以使用 `snprintf` 函数，它可以指定要写入的最大字符数。

通过以上示例，可以看出 C 语言提供了丰富的字符串处理功能，可以方便地对字符串进行各种操作。然而，需要注意的是，在使用这些函数时需要包含相应的头文件，如`<string.h>`。

文件操作

在 C 语言中，文件操作是指对文件进行读取、写入、删除等操作。文件操作的基本语法如下：

```
FILE *fopen(char *filename, char *mode);
```

例如，打开一个名为"example.txt"的文件，可以这样写：

```
FILE *file = fopen("example.txt", "r");
```

1. 逐字显示文件内容

```
#include <stdio.h>

int main() {
```

文件操作

```c
FILE *file;
char filename[] = "example.txt";
char ch;

// 打开文件
file = fopen(filename, "r");
if (file == NULL) {
    printf("无法打开文件 %s", filename);
    return 1;
}

// 逐字符读取文件内容并打印
while ((ch = fgetc(file)) != EOF) {
```

文件操作

```
            putchar(ch);
        }

        // 关闭文件
        fclose(file);

        return 0;
}
```

在上面的示例中，我们首先声明了一个 FILE 指针变量 file 和一个字符数组 filename，用于存储要读取的文件名。然后，我们使用 fopen 函数以只读模式打开文件，并将返回的文件指针赋值给

文件操作

file。如果文件打开失败，fopen 函数将返回 NULL，我们可以检查这种情况并打印错误信息。

接下来，我们使用 fgetc 函数逐字符读取文件内容，并将其打印到屏幕上。当读取到文件末尾时，fgetc 函数将返回 EOF（End of File），我们可以使用这个条件来结束循环。

最后，我们使用 fclose 函数关闭文件。这是一个良好的编程习惯，可以确保释放文件资源并避免潜在的错误。

文件操作

需要注意的是，在使用文件操作函数时，需要包含相应的头文件，如<stdio.h>。此外，还需要处理可能出现的错误情况，如文件不存在或无法打开等。

2. 逐行显示文件内容

```c
#include <stdio.h>

int main() {
    FILE *file;
    char filename[] = "example.txt";
    char line[100];
```

文件操作

```c
// 打开文件
file = fopen(filename, "r");
if (file == NULL) {
    printf("无法打开文件 %s", filename);
    return 1;
}

// 逐行读取文件内容并打印
while (fgets(line, sizeof(line), file) != NULL) {
    printf("%s", line);
}

// 关闭文件
```

文件操作

```
    fclose(file);

    return 0;
}
```

3. 写文件

```
#include <stdio.h>

int main(int argc, char const *argv[])
{
    FILE *file;
    file = fopen("score.txt", "w");
    if (file == NULL)
```

文件操作

```
        {
            printf("Error opening file!\n");
            return 1;
        }
        fputs("张三 80 85 90 88 75\n", file);
        fputs("李四 82 80 92 83 70\n", file);

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

socket 编程

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

int main() {
    int server_sock, client_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_size;
    char buffer[1024];
```

socket 编程

```c
// 创建套接字
server_sock = socket(PF_INET, SOCK_STREAM, 0);
if (server_sock == -1) {
    perror("socket");
    exit(1);
}

// 配置服务器地址
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
server_addr.sin_port = htons(12345);
```

socket 编程

```
    // 绑定套接字
    if (bind(server_sock, (struct sockaddr *)&server_addr,
sizeof(server_addr)) == -1) {
        perror("bind");
        exit(1);
    }

    // 监听连接
    if (listen(server_sock, 5) == -1) {
        perror("listen");
        exit(1);
    }
```

socket 编程

```
    // 接受客户端连接
    client_addr_size = sizeof(client_addr);
    client_sock = accept(server_sock, (struct sockaddr
*)&client_addr, &client_addr_size);
    if (client_sock == -1) {
        perror("accept");
        exit(1);
    }

    // 接收并发送数据
    while (1) {
        ssize_t recv_len = recv(client_sock, buffer,
sizeof(buffer) - 1, 0);
```

socket 编程

```
        if (recv_len <= 0) {
            break;
        }
        buffer[recv_len] = '\0';
        printf("Received: %s", buffer);
        send(client_sock, buffer, recv_len, 0);
    }

    // 关闭套接字
    close(client_sock);
    close(server_sock);
```

```
    return 0;
}
```

在上面的示例中，我们首先创建了一个套接字，并将其绑定到本地地址和指定端口上。然后，我们开始监听连接请求，并在接收到客户端连接后，使用 recv 函数接收数据，并使用 send 函数将数据发送回客户端。最后，我们关闭了套接字。

需要注意的是，在使用套接字进行网络编程时，需要包含相应的头文件，如<sys/socket.h>、<netinet/in.h>等。此外，还需要处理可能出现的错误情况，如套接字创建失败、绑定失败、监听失败等。