C 语言教程

（C11 版）

零基础入门系列教程

作者：魏明择

2025 年版

https://weimingze.com

目录

发布记录

版本	发布日期	备注
V1.0	2025年12月15日	V1.0 发布

序

前言

接触 C 语言已经二十多年的时间了。当年在大学开始接触计算机，我的第一门学习的编程语言是 `Fortran`，当年学习的是 `Fortran77` 的版本。`Fortran77` 的语法格式非常严格（可能是为了严谨），导致写代码的时候经常出错。后来我又自学了 `Basic`、`Pascal` 等编程语言。当我第一次接触 `C` 语言时，它的语法风格就深深的吸引了我。C 语言的复合语句使用一对美妙的大括号 `{}` 代替 `Pascal` 语言中的 `begin` 和 `end`，相对来讲 C 语言的语法简单且语义明确。

之前看过很多 C 语言的书，包括大学里普遍使用的教材。但说实话，我感觉很多书都是太结构化，没有按着逐步深入的思想来讲解，有时候会莫名其妙的引用一些名词又不做解释。我感觉这些书都是给学会了 C 语言的人看的，而不是给要学习的人看的。因此我决定在我自己的个人网站编写这篇 **《C 语言教程》** 献给喜欢我和热爱软件编程的朋友。我相信这是对初学者最友好的 **《C 语言教程》**。 当然，这个 **教程** 也适合工作一段时间的 C 语言开发者对 C 的语法进行查缺补漏。

从2025年7月底至今（2025年12月15日），经过不断地努力，我终于完成了整篇内容，现在上线发布。希望我的勤奋能为努力改变明天的朋友铺路和助力。

C 语言简介

先说一下编程语言的发展历史。

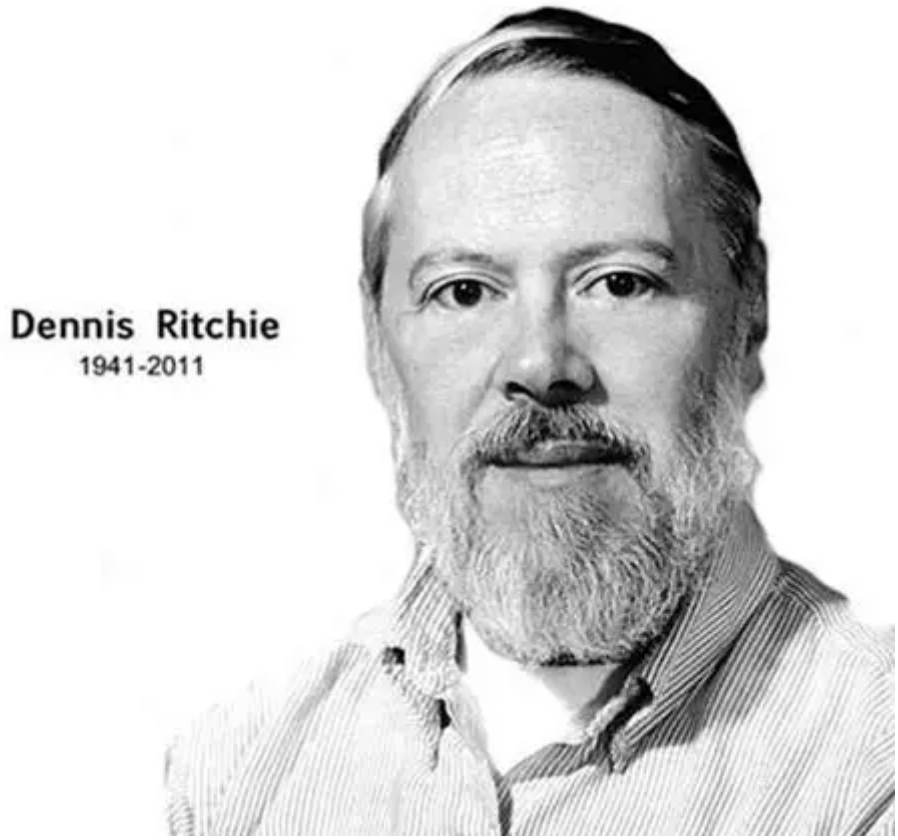
计算机是一个数字电路的硬件设备。要想让这个硬件设备按着我们预想的逻辑来运行，我们就需要给计算机设置一些硬件指令，计算机会严格按着这些硬件指令来执行。这些硬件指令可能是由一个字节、二个字节等不同长度的二进制数字组成。早期为了编写这些指令可以直接写二进制的代码，比如在有标记的纸带上打孔或不打孔来表示 `1` 或者 `0`，这样就可以驱动计算机硬件按着我们编写的逻辑运行并得到运行结果。我们把编写计算机逻辑指令的过程称为**编程**。 后来出现了汇编语言，用符号来代替 `0/1` 的代码，效率明显提高很多。再后来出现了 C 语言，C 语言能够将编码直接转化为汇编语言，并转化为 `0/1` 代码效率更进一步。用 C 语言编写的代码在编译的时候会自动计算各个变量的存放地址，并且可以根据硬件平台的不同转换成不同的汇编语言进行编译，进而降低了编程难度和代码的可移植性。再后来出现了解释执行类型的编程语言，让编程更加轻松，当然也会有效率上的损失，比如：Python、java 等。

C 语言是一种通用的、高效的编程语言。C语言是一门编译型的语言，所谓编译型语言是指你编写的 C 语言程序通过编译器最终编译成 **处理器**（也称为 CPU）能够直接执行的计算机指令。这些指令能在 **处理器** 中直接运行，而不在依赖任何的程序来运行。编译型的语言的特点是执行速度快，消耗内存少，甚至不需要有操作系统。

要学习 C 语言。我们先来介绍一下 C 语言的创始人。

C 语言是 丹尼斯·里奇（Dennis Ritchie） 在 1972 年于贝尔实验室开发。它最初用于重新实现 Unix 操作系统，后来因其高效性和灵活性而广泛应用于系统编程、嵌入式开发、游戏开发等领域。丹尼斯·里奇（Dennis Ritchie）也被称为 **C 语言之父**。

作者：丹尼斯·里奇（Dennis Ritchie）



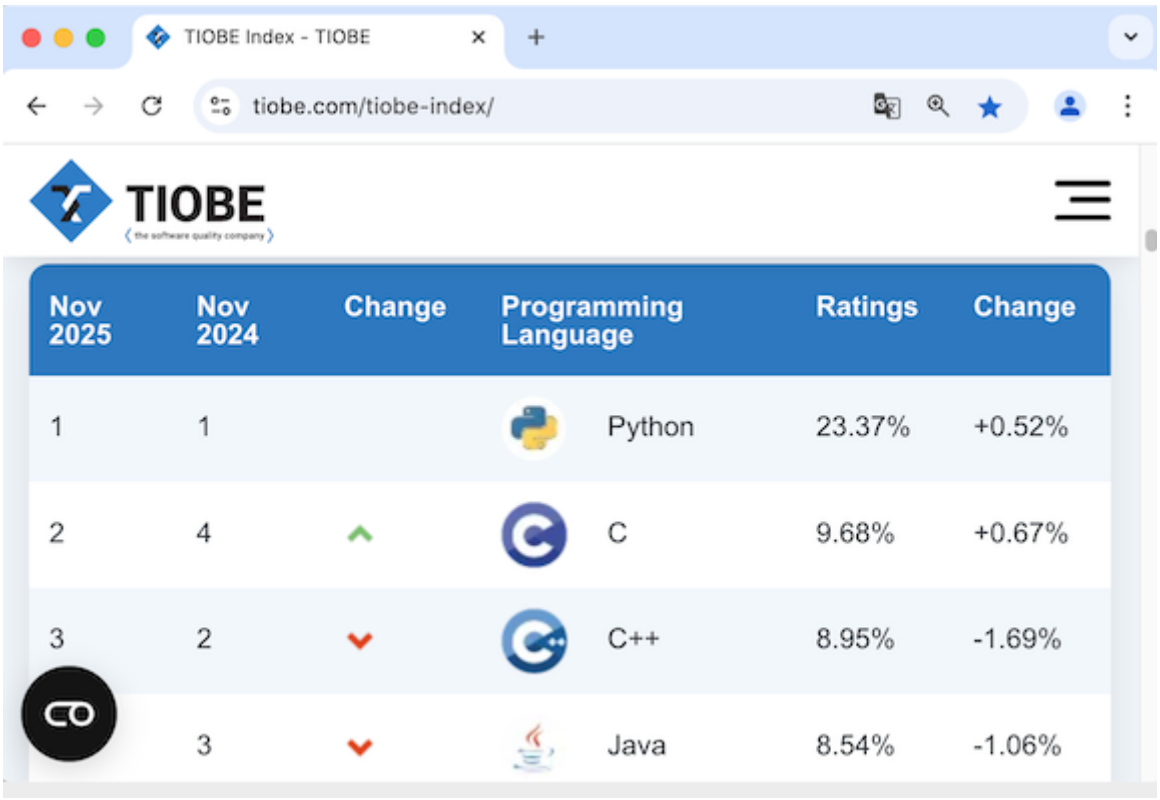
为什么学 C语言

一直以来 C 、 C++ 和嵌入式开发等相关行业一直是成熟的、稳步发展的行业。相关的开发岗位人才缺乏、就业前景比较乐观。而 C 语言是这些领域的基础学科。

学习本课程将为您在嵌入式、智能硬件、物联网、人工智能底层算法实现、Linux 内核驱动等领域打下良好基础。

C 语言目前的状况

C 语言从诞生到现在有五十多年的历史。并且经久不衰，目前经 tiobe-index (最权威的编程语言排行网站) 的统计，截止 2025年11月，C 语言的排名是第二位，并且一直保持领先的位置。如下图所示：



下面是自 2002 年至今的 C 语言排名趋势图，可见近 20 年来 C 语言排名一直处于前列。

C 语言的应用场景

- 系统编程：操作系统（Linux、Windows 内核）、编译器。
- 嵌入式开发：单片机、物联网设备。
- 高性能计算：科学计算、硬件控制。
- 底层开发：驱动程序、网络协议栈。
- 服务器开发：网络后端服务引擎、游戏后端引擎等。

C 语言的主要特点

- 高效性：
  - 接近底层硬件，可以直接操作内存（指针），适合开发操作系统、驱动程序等。
  - 编译型语言，运行速度快。
- 简洁灵活：
  - 语法简洁，核心关键字少（如：if、for、while、int 等）。
  - 支持结构化编程（函数、迭代、选择判断）。
- 可移植性：
  - 标准 C 代码可以在不同平台（Windows、Linux、嵌入式系统等）上编译运行。
- 丰富的运算符：
  - 支持位运算、指针运算等底层操作。
- 面向过程：
  - 以函数为基本单位，不同于 C++/Java 的面向对象。

C 语言的优缺点

- 优点：高效、灵活、跨平台、影响深远（C++/Java/Python 等语言都参照C语言设计）。
- 缺点：缺乏现代语言特性（如：面向对象、垃圾回收），手动内存管理易出错。

C 语言的标准

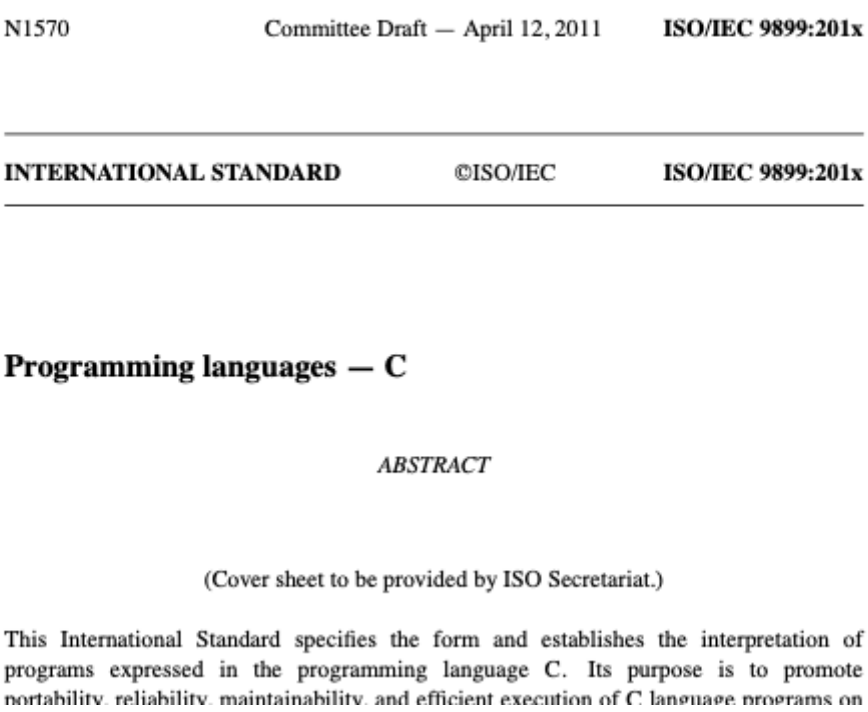
C语言有自己的编码标准，C语言的标准经历了多个版本的演进，每个版本引入了新的特性和改进。以下列出几种主要的C语言标准及其关键特性：

1. `ANSI C` / `C89` / `C90`（首个官方标准）
   - ISO/IEC 9899:1990（1990年，国际标准化组织采纳，称为C90）。

- 基本确定了现代 C 语言的基本特征和关键字，如 void、enum、const、volatile和预处理指令#elif等。

2. C99（ISO/IEC 9899:1999）
   - 新增了单行注释 `//`、`restrict` 指针、 `long long` 类型、变长数组等。
3. C11（ISO/IEC 9899:2011）
   - 引入了多线程支持 `<threads.h>`(但要依赖编译器)、匿名结构体/联合体等。
4. 后续版本 C17 / C18 只是修正了前面版本的缺陷，无新功能
5. C23（ISO/IEC 9899:2023）
   - 引入了 `nullptr` 代替 `NULL`（兼容C++）和 #elifdef 等预编译宏以及常量表格式 `constexpr`等。目前有部分编译器支持。

本课程参照 C11 标准（ISO/IEC9899:201x）官方文档编写。请各位自行搜索下载。参照如下：



N1570 Committee Draft — April 12, 2011 **ISO/IEC 9899:201x**

INTERNATIONAL STANDARD ©ISO/IEC **ISO/IEC 9899:201x**

Programming languages — C

ABSTRACT

(Cover sheet to be provided by ISO Secretariat.)

This International Standard specifies the form and establishes the interpretation of programs expressed in the programming language C. Its purpose is to promote portability, reliability, maintainability, and efficient execution of C language programs on

本课程的教学环境

本课程使用 Linux 作为教学环境。以 GNU 的 C 语言编译器 `gcc` 作为编译器进行讲解。如果你是 Windows 操作系统或 MacOS 操作系统，请按第一章的内容来搭建相应的 C 语言开发环境。

> Linux 是大厂公司最常用的操作系统，使用 Linux 环境已经是大势所趋。

本教程的案例均在 Ubuntu 24.04 LTS 上验证，学习本教程前请先自行安装 Ubuntu 24.04 LTS 操作系统或下载由本站制作的 VMware 虚拟机镜像。

Ubuntu24.04 VMware镜像:

- 百度网盘下载地址:

- https://pan.baidu.com/s/1jYAdvByctakhRVrDIteoLA 提取码: m8ee

- 用法:
  1. 解压缩 `Ubuntu24.04LTS_IntelX86CPU.zip`
  2. 使用`VMware workstation pro v16.0` 或 `VMware fusion v12.1` 及以上版本打开 `Ubuntu24.04LTS_IntelX86CPU/Ubuntu24.04LTS.vmx` 文件并运行。

- 登录信息: 此虚拟机的登录用户名是：`weimingze`，密码：`weimingze`，root 用户的密码也是：`weimingze`。

Ubuntu下载和安装方法详见: Ubuntu Linux 简介

前置课程

在学习此内容之前你需要先学习 《Linux教程》。关于Ubuntu 操作系统的安装和虚拟机的下载和安装都在 《Linux教程》 中讲过，这里不在赘述。

版权声明

第一章、开发环境的搭建

C 语言是编译型语言，我们编写的代码需要使用编译器编译成计算机能够执行的指令和数据才能执行，因此要编写和运行 C 语言的程序需要编辑器（如：vim、Visual Studio Code、记事本等）和编译器（如：gcc、Clang、MSVC等）。当然我们也可以使用有些厂家提供的将编辑器、编译器、调试器等集成在一起的开发环境（如：XCode、Microsoft Visual Studio等）。

1. Linux 安装 C 开发环境

Linux 下比较成熟的 C 语言编译器是 **gcc**。

gcc 是 GNU 项目组开发的，专门为 Linux 内核编译和应用程序编译而设计的编译器。并且开源免费。

1.1 gcc 安装

在 Ubuntu Linux 下使用 apt 命令。在 CentOS 下使用 yum 命令就可以很方便的安装 gcc 编译器。

有些 版本的 Linux 甚至已经将 gcc 预先安装好了，直接使用即可。

打开 Linux 的 **终端**，使用如下命令安装 gcc。

Ubuntu 下安装 gcc

```
sudo apt install gcc
```

CentOS 下安装 gcc

```
sudo yum install gcc
```

验证 gcc 是否成功安装

在终端下，输入 gcc 命令，如果提示没有这个命令。则说明没有安装 gcc。安装成功的效果如下：

```
weimingze@mzstudio:~$ gcc
gcc: fatal error: no input files
compilation terminated.
```

> 以上提示是没有给出 C 语言的文件而导致的。

使用 gcc 的 `--version` 选项可以查看 gcc 的版本信息。如下所示：

```
weimingze@mzstudio:~$ gcc --version
gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

> 当前版本是 `13.3.0`。

至此，Linux 下的 C 语言编译器已经安装成功，你可以使用 gcc 将 C 语言的代码编译成可执行的应用程序了。

1.2 Vim 编辑器

C 语言程序都是文字信息，因此需要使用纯文本编辑器编写 C 语言的代码。

在 Linux 下可以使用 **Vim** 作为 C 语言的编辑器，也可以使用 Visual Studio Code 作为编辑器。以下介绍 Vim 编辑器的安装、配置和使用。

Ubuntu 下安装 Vim

```
sudo apt install vim
```

CentOS 下安装 Vim

```
sudo yum install vim
```

Vim 编辑器的用法详见我的 《Linux 教程》 的第五章的内容。

在使用 Vim 编写 C 语言代码时。可以修改 Vim 的配置文件 `.vimrc` 来更改默认配置，以便实现代码高亮，自动缩进等常用功能。

> 修改 `~/.vimrc` 只对本用户的配置生效。如果想对所有用户的vim配置，请修改 `/etc/vim/vimrc` 文件。

修改 .vimrc 方法:

终端内使用 Vim 打开用户主文件夹下的 `.vimrc` 文件。命令如下：

```
vim ~/.vimrc
```

以下是魏明择本人电脑的 Vim 配置，供参考。内容如下：

```
" 这是 .vimrc 配置文件的注释，注释以双引号(")开头直至行尾。
set number   "  设置显示行号
set tabstop=4      " 设置制表符(Tab)转换为4个空格
set shiftwidth=4   " 自动缩进时使用 4 个空格
set expandtab      " 将 Tab 转换为空格

set mouse=a   " 设置鼠标支持
set cursorline   " 高亮当前行
set showmatch   " 高亮显示匹配的括号

set cindent   " 设置 C 语言编写时自动缩进

syntax on   " 设置语法高亮
```

> 注： `.vimrc` 配置文件的注释是以 双引号(")开始直至行尾的内容。

至此，我们可以在 Linux 下编写 C 语言的代码了。

第二章、初步认识 C 语言

本章我们将从最简的C语言程序开始了解 C 语言的语法规则、编写、编译及运行。

1. 从 Hello World 开始

C 语言的程序需要使用文本编辑器进行编写。

C 语言的文件通常是 `.c` 结尾，即后缀名是 `.c`。

编写第一个 C 语言程序

使用编辑器编写一个文件 `hello.c`

内容如下：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello World!\n");
    return 0;
}
```

> 在 Linux 下可以使用 **Vim** 或 **文本编辑器** 编写。

使用 gcc 编译这个文本文件成为可执行文件。

打开一个 UNIX 或 Linux 终端，在终端中找到这个 `hello.c` 文件的位置。然后执行 `gcc hello.c` 来编译这个 `hello.c` 文件。如下所示：

```
weimingze@mzstudio:~$ gcc hello.c
weimingze@mzstudio:~$ ls
a.out  hello.c
```

可见此时多了一个可执行文件 `a.out`。

> 如果你是使用 MacOS 系统或 Windows 系统下的其它 C 语言开发环境。请找到相关的方法进行编译和运行。

在 Linux 运行编译后的可执行程序

```
weimingze@mzstudio:~$ ./a.out
Hello World!
```

此时你可以看见在终端中打印了一行 `Hello World!`。至此你已经生成了第一个 C 语言编写的可执行程序（a.out）。

练习

打印如下如正方形。

```
########
########
########
########
```

2. C 语言程序的最小结构

本小结我们来介绍 C 语言的最小程序结构。

C 语言的程序是从一个名为 `main` 的函数开始运行的。这个函数也通常被称为应用程序的入口函数。

C 语言的入口函数是操作系统启动这个程序的起点。这个函数在经过编译器编译时会放在二进制代码的特殊位置，方便操作系统找到并执行这个程序。

C11 的标准规定了两种 `main` 函数的写法：

写法1

```c
int main(void) {

    return 0;
}
```

写法2

```c
int main(int argc, char *argv[]) {

    return 0;
}
```

上述两种 C 语言的写法的不同之处：

- 写法1 中 `int main(void)` 表示不接收任何命令行传入的参数和选项（通常用于 Windows 系统中）。
- 写法2 中 `int main(int argc, char *argv[])` 表示接收命令行传入的参数和选项（通常用于 UNIX/Linux 系统中）。

详细说明：

1. `main` 是主函数的名字，也是程序的入口（相当于家里的入户门，要进来必须走这里），并且全局只能有一个`main`函数。
2. `main` 函数左侧的 `int` 代表返回的类型是整数(`int`)，`main` 函数要求必须有一个整数(`int`)的返回值，在 MacOS 和 Linux 下，这个值必须是 0~255 范围内的数，如果超出这个范围，则会留取低 8位的数值。在Windows 下基本不会用到这个值。这个值代表当前程序是否执行成功，通常 `0` 代表成功，非零值代表不同类型的错误。 如：`return 0;` 就是程序执行完返回 `0`。这个返回值会被 UNIX/Linux Shell中的 `$?` 特殊变量绑定。
3. main 后面的 `(void)` 或 `(int argc, char *argv[])`时调用参数列表，这个参数用来接收命令行传入的参数和选项，并使用 `argc` 变量来记录选项和参数的个数，使用 `argv[]` 指针数组来记录选项和参数的内容（这些内容会在后续进行详细讲解，此时不明白也没关系）。
4. 一对大括号（`{}`）是 main 函数的函数体部分。起初执行的语句要放在其中。

上述两种写法是 C 语言的最小程序结构，任选其一即可。如果要完成更过的功能我们需要向这个框架中加入更多的代码。

上节课我们编写了 `hello.c` 这个程序内容如下：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello World!\n");
    return 0;
}
```

其实我们也可以改写如下：

```c
#include <stdio.h>

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

运行结果没有变化。

hello.c 详解：

上述程序中 `printf()` 是函数调用，括号中的 `"Hello World!\n"` 是字符串（用来表达人类文字的内容）。字符串相当于一段人类的文字，字符都是以英文的双引号 (") 开始和结束。字符串中的 `\n` 表示打印一个换行符。

`printf()` 函数后面的英文分号（`;`）是语句的结束符号。

此时你看到的 `Hello World!` 就是由 `printf()` 函数打印的标准输出。

在使用 `printf` 这个函数在终端打印文字前，必须先告诉 C 语言的编译器 printf 是一个怎样的函数，这个叫做函数声明。而这个声明在文件 `stdio.h` 中。我们需要在 `hello.c` 这个文件在使用 `printf()` 函数前面包含这个头文件。

第一行的 `#include <stdio.h>` 是预处理一条预处理指令，意思是将 `stdio.h` 文件中的内容插入到当前 `hello.c` 文件中。

stdio 这个文件名的含义是标准输入输出（即：Standard Input/Output），因为对于程序来讲，打印属于输出。

stdio.h 这个文件有很多基本输入输出函数的声明，如：`printf`、`scanf`、`getchar`、`putchar`等。如果要使用这些函数，则必须包含此头文件。

练习

使用两种 C 语言的 main 函数写法改写 `Hello World` 程序，并尝试打印多个 `Hello World`。

3. C 语言的编译过程

当一个编写好的 `.c` 文件经过编译变成二进制的可执行文件一共经过四个阶段。

这四个阶段分别是：

1. 预处理（Preprocessing）。
2. 编译（Compilation）。
3. 汇编（Assembly）。
4. 连接（Linking）。

下面我们使用 Linux 下的 gcc 编译器来了解一下这四个阶段。这也是C语言开发者必须知道的编译过程。

GCC 编译器

GCC（GNU Compiler Collection）是GNU 开源组织开发的编译器集合。 它是开源领域最重要、应用最广泛的编译器之一，支持多种编程语言和硬件平台。

gcc 命令格式如下：

```
gcc [选项] 文件路径
```

常用选项

选项	说明
`-o <文件路径>`	指定输出路径，代替`a.out`。
`-g`	加入调试信息（用于GDB调试）。
`-I<文件夹路径>`	指定头文件搜索路径。
`-l<库名>`	指定库文件名称。
`-L<路径>`	指定库文件搜索路径。
`-static`	静态链接。
`-shared`	生成动态库（.so）。
`-fPIC`	生成位置无关代码（用于动态库）。
`-D<宏>`	定义宏。
`-E`	仅预处理，不编译、汇编和连接。
`-S`	仅编译，不汇编和连接。
`-c`	编译和汇编，但不连接。
`-std=<standard>`	指定编译标准，如:`c89`、`c99`、`c11` 等。
`--version`	显示版本信息。
`-O<n>`	设置优化级别，n为 0~3。
`-Wall`	报告所有警告。

1. 预处理

预处理是将 C 语言中以井号(#) 开头的预处理指令（如:`#include <xxx.h>`、`#ifdef XXX`等）进行处理后生成可以编译的 **C 程序**。

在 GCC 中使用 `-E` 选项可以进行预处理但不执行后续步骤。如下所示：

```
weimingze@mzstudio:~$ gcc -E -o hello.i hello.c
weimingze@mzstudio:~$ ls
hello.c  hello.i
```

上述运行后生成的 `hello.i` 的部分内容如下：

```
# 0 "hello.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4

...  # 此处省略多行

extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)
) __attribute__ ((__nonnull__ (1)));
# 959 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 983 "/usr/include/stdio.h" 3 4

# 2 "hello.c" 2


# 3 "hello.c"
int main(int argc, char *argv[]) {
    printf("Hello World!\n");
    return 0;
}
```

2. 编译。

编译是将预处理后的 C 语言信息进行编译，将其生成汇编代码。

在 GCC 中使用 `-S` 选项可以进行编译，但不执行后续步骤。以下对 `hello.i` 进行编译，生成文件 `hello.s`。

```
weimingze@mzstudio:~$ gcc -S hello.i -o hello.s
weimingze@mzstudio:~$ ls
hello.c  hello.i  hello.s
```

`hello.s` 的内容如下：

```
    .file   "hello.c"
    .text
```

```
    .section    .rodata
.LC0:
    .string "Hello World!"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movq    %rsi, -16(%rbp)
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    call    puts@PLT
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long   1f - 0f
    .long   4f - 1f
    .long   5
0:
    .string "GNU"
1:
    .align 8
    .long   0xc0000002
    .long   3f - 2f
2:
    .long   0x3
3:
    .align 8
4:
```

3. 汇编。

编译是将上述汇编程序的指令等进行替换，替换成二进制的机器指令，通常生成的 `.o` 文件，我们称之为**目标文件**。`.o` 文件是二进制文件，用文本编辑器无法阅读。

以下将 `hello.s` 汇编后生成 `hello.o`


```
weimingze@mzstudio:~$ gcc -c -o hello.o hello.s
weimingze@mzstudio:~$ ls
hello.c  hello.i  hello.o  hello.s
```

> 文件 `hello.o` 是汇编后的二进制文件。

4. 链接。

链接是将一个或多个 `.o` 文件连接成为一个可执行程序程序。

以下将 `hello.o` 链接成为可执行程序 `hello`。

```
weimingze@mzstudio:~$ gcc -o hello hello.o
weimingze@mzstudio:~$ ls
hello  hello.c  hello.i  hello.o  hello.s
weimingze@mzstudio:~$ ./hello
Hello World!
```

经历上述四步，我们将 `hello.c` 最终生成了可执行程序 `hello`。

练习：

写一个程序，打印两行 `Hello World!`，使用上述四个步骤最终生成可执行文件并运行。

4. C 语法基础

什么是语法

语法（Syntax） 是指编写 C 语言程序时必须遵循的规则和结构，它定义了如何正确地组合关键字、标识符、运算符和其它元素来形成有效的 C 语言程序。如果违反语法规则，编译器会报错（Error）。

白话文解释

语法就是语言的模版或是套路，你只用按这个套路写，C 编译器才能认得你想表达的东西。

本站语法标注规则

1. 语法都会写在一个代码框内。
2. 代码框内的中文需要用相应的内容替代。
3. 代码框内的英文单词是关键字，不能改动；
4. 代码框内的符号通常是分隔符，如果下方无备注说明也不能改动。

> 本规则适用于本人文档、视频等全部内容。

语法相关术语

- 字面值
  - 字面值是内置类型常量值的表示法。
- 标识符
  - 标识符（也称为**名称**），是用来表示一段代码后一段数据的名称，这个名字有起名字的规则，不能随便取名，这个规则，后面会讲。
- 关键字
  - 关键字也被称为保留字，是特殊的标识符，不可用于普通标识符。就像 **中国** 这个名字一样，你去给孩子或公司取名子都不能用这个名。
- 运算符
  - 表示运算规则的符号。
- 缩进
  - 通常位于行首的1个或多个空格，从视觉上显示包含关系。C语言建议使用4个空格表示一个缩进。
- 表达式
  - 用于计算并一定能返回一个值的字面值、变量值或计算式，它一定能返回一个结果(数字、字符串、指针等)。
- 语句
  - 一个完整的执行单位，就像你说的一句话一样，你完整的说出来一句话，别人也能按这句话去做事情，你说不完整或有错，别人也无法按你说的做。

练习

尝试使用一个 printf 函数打印如下的一个长方形。

```
**********
*        *
*        *
**********
```

5. 注释

什么是注释

注释通常是用于解释代码的文本。注释的内容是让编译器忽略其中的内容，不让其参与编译和执行。

注释的作用

注释的主要目的是提高代码的可读性，方便开发者或其他人理解代码的逻辑、功能或设计意图。

C 语言的注释方法有两种:

1. 多行注释：以 `/*` 开头，以 `*/` 结尾，中间的所有内容都是注释，可以跨越多行（C89/C90标准）。
2. 单行注释：以 `//` 开头，直到行末的内容都是注释（C99标准启用）。

> 注意： 多行注释不可嵌套，如 `/* aaa /* bbb */ ccc */` 其中 `ccc */` 的内容不是注释。

示例

```c
#include <stdio.h>

/* 这是一个多行注释。
   通常用于函数说明或较长的解释。
*/
int main(int argc, char *argv[]) {
    // 这是一个单行注释，以下用来打印一行 Hello World!
    printf("Hello World!\n");
    return 0;  // 这也是一个单行注释
}
```

在 C 语言中，除了注释和字符串的内部可以写入中文以外，其它任何地方不允许出现中文。否则将会报告各种的错误（**Error**）。

练习

打印如下的一个三角形。

```
   *
  ***
 *****
*******
```

第三章、基础数据类型

本章我们将学习 C 语言的基础数据类型和它们的内存结构。要学好 C 语言则必须了解每个数据的内存结构。

C 语言不同于 Python 编程语言。Python 语言则完全封装了内存结构，使用者不需要了解内存结构就可以编写出符合逻辑的应用程序。

C 语言是计算机机器指令的表达方式，它的每一条语句都会翻译成相关的 CPU 指令或者数据在 CPU 上执行。C语言的执行速度基本是没有损耗的。但这也给 C 语言的编写者带来很多学习上的难度。

本章将从最简单的 **进制** 讲起，让我们深入了解计算机的内存结构。

1. 十进制和二进制

十进制我们并不陌生，在我们的生活中经常使用十进制的表示方法来描述现实世界的数字，比如：年龄、人口数量、身高等。

十进制就是逢十进一的表示方式，同理二进制就是逢二进一的表示方式，三进制就是逢三进一的表示方式。

计算机中常用的表示方式有十进制、二进制、八进制和十六进制。

十进制

以下是我家里的天然气表显数字。

可见这个表显数字是 05589.60 ，其中 5589 是整数部分，.60 是小数部分。我们先来研究整数部分。整数部分的表示如下：

```
 万位    千位    百位    十位    个位
+-----+-----+-----+-----+-----+
|  0     5     5     8     9  |
+-----+-----+-----+-----+-----+
```

我们发现每一个位都是一个小的滚轮，滚轮的最小值是0，然后是1，..... 最大值是9。当这个表数再加上 1 后，个位就进位成为 0，十位变成 9。表显将变成 `05590`。

在十进制中，一个位能表示：`0~9`。 如果要表示更大的数，则要增加位数，两个位则能表示：`0~99`，三个位能表示： `0~999`。十进制每增大一个位，表示数字范围增大为原来的十倍。

二进制

二进制和十进制的原理是一样的，也就是表显的滚轮只有两个数字(`0`和`1`)。是要逢二进一。一个位能表示：`0~1`，两个位能表示：`00~11`，即十进制的 `0 ~ 3`，三个位能表示：`000~111`，即十进制的 `0 ~ 7`。二进制每增大一个位，表示数字范围增大为原来的二倍。

八个位的二进制的表示的最小值是`00000000`：

```
 7位 6位 5位 4位 3位 2位 1位 0位
+---+---+---+---+---+---+---+---+
| 0   0   0   0   0   0   0   0 |
+---+---+---+---+---+---+---+---+
```

八个位的二进制的表示的最大值是`11111111`（即十进制的255）：

```
 7位 6位  5位 4位 3位  2位 1位 0位
+---+---+---+---+---+---+---+---+
| 1   1   1   1   1   1   1   1 |
+---+---+---+---+---+---+---+---+
```

当数字为二进制时，**位**通常用 0 开始，第一个位是 0 位，第二个位是 1 位，..... 以此类推。

同理，三进制就是表显数字滚轮有三个数字，八进制就是表显数字滚轮有八个数字，十六进制就是表显数字滚轮有十六个数字。

由于计算机内部为了简化电路的复杂度，都统一使用二进制来表示数字，即低电平代表 0，高电平代表 1。

进制只是表示现实世界数值的表达方式，二进制的运算更简单，但表示位数会比较长，十进制计算相对复杂，表示同样的数，使用十进制位数较少。

思考

以下的二进制表示的数值是多少?

```
 7位 6位  5位 4位 3位  2位 1位 0位
+---+---+---+---+---+---+---+---+
| 0   1   0   0   1   0   1   1 |
+---+---+---+---+---+---+---+---+
```

它表达的数值是:

$$0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 75$$

注意:

二进制的底数是 2，十进制的底数则是 10，三进制的底数则是 3，以此类推。

人类为什么使用十进制，你看看自己的双手共有几个手指大概就明白了。如果有一天人类发现了外星文明，那外星文明用几进制表示数字呢?

练习:

计算以下的二进制表示数值用十进制表示是多少?

```
 7位 6位  5位 4位 3位  2位 1位 0位
+---+---+---+---+---+---+---+---+
| 0   1   1   0   1   0   1   0 |
+---+---+---+---+---+---+---+---+
```

2. 八进制和十六进制

由于**二进制**表达数字时位数比较多。因此在 C 语言中常使用**八进制**和**十六进制**来表示一个二进制的数值。

原因是 8 正好是 2 的三次方，16 正好是 2 的四次方。即 8 进制的一个位正好可以转换位二进制的三个位且没有余数。十六进制也是如此。

八进制

逢八进一，每个位用 `0~7` 的数字表示。

十六进制

逢十六进一，每个位的值如果在十以内则用 `0~9` 的数字表示。大于等于十的部分用英文字母表示，如`A` 或 `a` 表示 10，`B` 或 `b` 表示 11，……，`F` 或 `f` 表示 15。

二进制、八进制、十六进制互转

二进制转八进制时，将二进制的最右侧起的每相邻三位转为 八进制的位即可。

如：

```
二进制
+---+---+---+---+---+---+---+---+
| 0   1   0   0   1   0   1   1 |
+---+---+---+---+---+---+---+---+

拆解二进制
+---+---+ +---+---+---+ +---+---+---+
| 0   1 | | 0   0   1 | | 0   1   1 |
+---+---+ +---+---+---+ +---+---+---+

每三个二进制位转为一个八进制的位
+-------+ +-----------+ +-----------+
|   2   | |     1     | |     3     |
+-------+ +-----------+ +-----------+
八进制就是: 213
```

二进制转十六进制时，将二进制的最右侧起的每相邻四位转为十六进制的位即可。

如:

```
二进制
+---+---+---+---+---+---+---+---+
| 0   1   0   0   1   0   1   1 |
+---+---+---+---+---+---+---+---+
```

```
拆解二进制
+---+---+---+---+ +---+---+---+---+
| 0   1   0   0 | | 1   0   1   1 |
+---+---+---+---+ +---+---+---+---+

每四个二进制位转为一个十六进制的位。
+---------------+ +---------------+
|       4       | |       B       |
+---------------+ +---------------+
十六进制就是：4B
```

由八进制转为二进制时，只要将八进制的一个位拆解成三个二进制位，然后排列在一起即可。十六进制转二进制时要将一个位拆解成四个二进制位，然后排列在一起即可。

练习：

思考如何将十六进制的 `A3` 转为 二进制表示。

3. 整数数据类型

计算机的内存和磁盘都是用来存储数据的工具。内部存储着大量二进制的数据。

现代计算机系统通常把 8 个二进制的位（bit）作为一个最小的存储单元进行管理。由8个位组成的存储单元称为 **字节**（Byte）。在写法上，通常 `1b` 代表一个位，而 `1B` 代表一个字节。

计算机的内存逻辑上是以字节为单位的存储单元，通常最小的分配单元就是一个字节，如果存储更大的数字可能需要临近的两个或四个字节来保存。

在计算机的内部，每个字节都有它的地址编号，这个编号用于定位这个字节在内存中的地址，以便于计算机的 CPU 对其进行访问和修改。

计算机的内存编号是从零开始的，每个字节的地址依次加一。对于一个 32 位的计算机系统，这个地址通常是 `0~FFFFFFFF(十六进制表示)`的范围。对于一个 64 位的计算机系统，这个地址通常是 `0~FFFFFFFFFFFFFFFF`的范围。

内存地址以示图：

```
  +--------+--------+--------+     +--------+--------+--------+
  |        |        |        | ... |        |        |        |
  +--------+--------+--------+     +--------+--------+--------+
地址：  0        1        2      ...  FFFFFFFD FFFFFFFE FFFFFFFF
```

> 每个方块代表一个字节（8个位）。

说明：

在计算机的图书中，在表示位的分布情况时，通常最低位在最右侧，高位在左侧。在表示字节的分布情况时，通常低地址的字节在**左侧**或**下面**，高地址的字节在**右侧**或**上面**。

整数：

在 C 语言中，在使用计算机存储数据时，先要考虑存储的数据的取值范围和符号特征，然后再用相应的数据类型进行存储。不同的数据类型表示不同的存储的字节数和符号的表示方式。

在 C 语言中，整数的存储分为**无符号整数（unsigned integer）**和**有符号整数（signed integer）**两种。

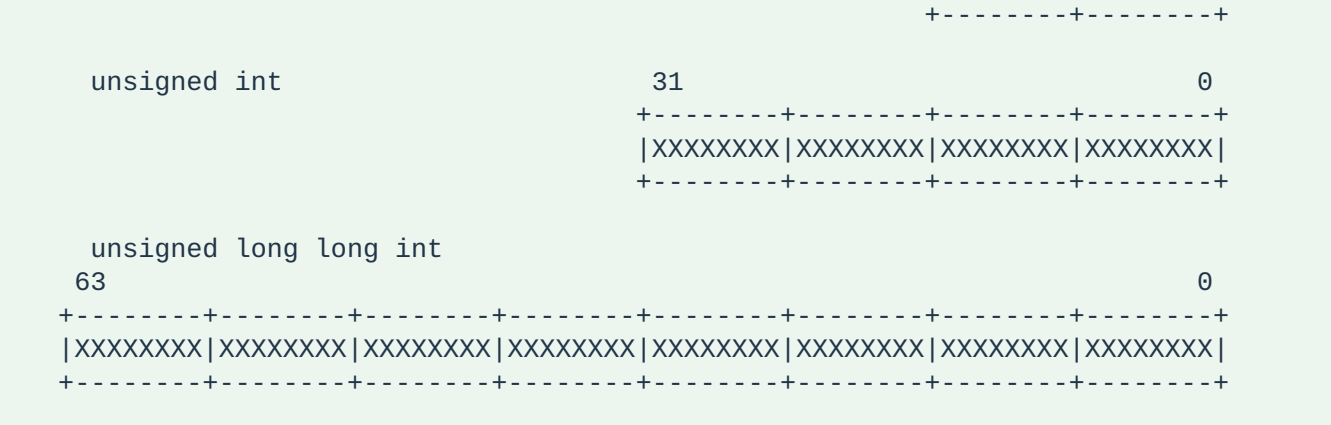
无符号整型

无符号整型只能存储大于等于零的数据。

数据类型	字节数	中文名	范围
unsigned char	1	无符号字符型	`0~255`
unsigned short int	2	无符号短整型	`0~65535`
unsigned int	4	无符号整型	`0~4294967296`
unsigned long int	4及以上	无符号长整型	`0~` 由操作系统和编译器决定字节数
unsigned long long int	8及以上	无符号长长整型	`0~18446744073709551616` 或更大

无符号整数内存表示如下：

```
unsigned char                                                  7      0
                                                               +--------+
                                                               |XXXXXXXX|
                                                               +--------+

unsigned short int                                          15              0
                                                           +--------+--------+
                                                           |XXXXXXXX|XXXXXXXX|
```



> x 表示一个位的值 0 或 1。

有符号整型

有符号整型能存储大于等于零的数据，也能存储小于零的负数。

数据类型	字节数	中文名	范围
signed char	1	有符号字符型	-128\~127
signed short int	2	有符号短整型	-32768\~32767
signed int	4	有符号整型	-2147483648\~2147483647
signed long int	4及以上	有符号长整型	0~由操作系统和编译器决定字节数
signed long long int	8及以上	有符号长长整型	-9223372036854775808\~9223372036854775807

对于用符号的整数中，内存的最高位为符号位，**0** 表示正数，**1** 表示负数。当数据存储是负数时，使用补码的方式进行存储。

补码

即原数的绝对值取反再加一就是补码。如有符号的字符型整数`-1`的补码是`11111111`，`-2`的补码是`11111110`。

补码计算方法

`-2`的补码计算，绝对值是`2`，八位二进制表示为`00000010`，取反后`11111101`，再加上一后为`11111110`。则`11111110`就是 -2 的补码。

补码的作用是方便计算机内部计算，如`-1 + 1`为零，计算方法为:

```
  11111111
+ 00000001
----------
  00000000
```

而 -1 的补码正是`00000000`再减一的结果，即二进制的滚轮向下滚动在借位得到的`11111111`。想象一下燃气表在为零时倒转一个数字的情况你就能明白补码的含义了。

有符号整数内存表示如下：

```
  signed char                                                 7      0
                                                              +--------+
                                                              |SXXXXXXX|
                                                              +--------+

  signed short int                                     15               0
                                                       +--------+--------+
                                                       |SXXXXXXX|XXXXXXXX|
                                                       +--------+--------+

  signed int                          31                                 0
                                      +--------+--------+--------+--------+
                                      |SXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|
                                      +--------+--------+--------+--------+

  signed long long int
 63                                                                      0
+--------+--------+--------+--------+--------+--------+--------+--------+
|SXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|
+--------+--------+--------+--------+--------+--------+--------+--------+
```

> `S`表示符号位的值，0（表示正数） 或 1（表示负数）。

在上述 C 语言的类型表示中 `signed` 可以省略不写， 当 int 类型有 `short` 或 `long` 修饰时， `int` 也可以不写。因此我们常写 `short` 是表示 `signed short int`，写`long` 表示 `signed long int`。

练习：

问答：

1. 有符号字符型 `signed char` 能保存多少种数字?
2. 无符号字符型 `unsigned char` 能保存多少种数字?
3. 无符号字符型 `unsigned char` 占用几个字节?
4. `short` 类型占用几个字节?
5. `long long int` 类型能用来保存天上星星的数量吗?

4. 浮点数类型

C 语言的数字类型大致有两种：整数和小数。

小数：

小数是指包含整数部分和小数部分的数字，在C语言中，小数通常用浮点数类型表示，包括 `float`、`double` 和 `long double`，共计三种。

例如：

- 圆周率：3.1415926
- 身高：1.73米

浮点数类型

数据类型	字节数	中文名	精度
float	4	单精度浮点型	6-7位十进制小数
double	8	双精度浮点型	15-16位十进制小数
long double	10或16(硬件相关)	长双精度浮点型	18-19位十进制小数

说明：

- 浮点数都是有符号的小数，其中最高位代表符号位。

- 浮点数的存储方式大都是小数加指数的存储方式，小数部分存储 0.5 ~ 1 之间的小数部分。 指数部分存储 2 的 n 次方中的n。
  - 例如：小数6.0 将会转化为 0.75 乘以 2 的 3 次方的形式。小数部分存储 0.75，指数部分存储 3。

浮点数的常用存储方法:

- float（单精度浮点数）
  - 1 位符号位。
  - 8 位指数部分。
  - 23 位小数部分（隐含开头的 1，实际精度为24位）。
- double（双精度浮点数）
  - 1 位符号位。
  - 11 位指数部分。
  - 52 位小数部分（隐含开头的 1，实际精度为53位）。
- long double（长双精度浮点数）
  - 1 位符号位。
  - 15 位指数部分。
  - 64 位或以上小数部分（具体依照硬件）。

以上数据浮点数类型的数据在 C 语言编译器中已经实现，一般开发者无需关注浮点数的具体存储细节。

练习：

下列内容使用什么数据类型存储比较合适?

1. 身高（厘米）
2. 体重（公斤）
3. 年龄
4. 央行黄金储量（克）
5. 上证指数

5. 变量

变量是 C 语言程序中用于存储数据的基本单元，它是内存中的一段连续的内存，其内存中的值可以在程序运行期间被修改。

变量通常有自己的名字，变量名是内存位置的标识符，通过这个名字可以访问这段内存。

C 语言中每个变量有特定的 **数据类型**，决定了它能存储的数据种类（如整数、小数等）和占用内存的大小。

变量的声明和初始化

变量必须先声明再使用。声明是告诉编译器，在程序运行到此处是要申请多大的内存空间用来保存何种类型的数据。

变量声明的语法：

```
数据类型 变量名1 [= 变量的初始值1],变量名2 [= 变量的初始值2];
```

> [] 表示其中的内容可以省略

声明：

本教程写语法的规则是：中文的部分需要使用对应的内容替换，英文部分和符号部分如果不加特殊说明，则是语法中必须书写的关键字和符号。

语法说明：

- 变量的声明必须以类型开始，以英文的分号（;）结束。
- 变量可以在声明时直接给出初始值，也可以在后续使用中通过赋值表达式给出新值。
- 变量在声明时没有给出初始值，则变量的初始值不确定（不一定是零，可能是任何数值）。

示例：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int age;      // 声明一个整型变量，初始值不确定。
    float price = 0.0;  // 声明一个浮点型变量，初始值为0.0
    int score1, score2 = 60; // 声明两个整型变量
    char gender = 'F';  // 声明一个字符型变量，初始值为'F'(值为70)。
                        // 字符 'F' 后续讲解。
    printf("age:%d, price:%f, score1:%d, score2:%d, gender:%d\n",
        age, price, score1, score2, gender
    );
    return 0;
}
```

运行结果

```
age:2113775720, price:0.000000, score1:32765, score2:6, gender:70
```

> 其中变量 age、score1 没有给出初始值，因此它的值充满不确定性。

上述程序中 printf 函数是为了打印各个变量的值，具体内容我们后续章节再解释。

上述程序中 age、price、score1、score2、gender 都是变量名。

C 语言的变量名不能随意书写，它有自己的命名规则，称之为：**标识符**（Identifier）。

标识符

标识符 用于在程序中自定义的名称，如：变量名（variable）、函数名（function）、数组名（array）等。

C 语言的标识符的命名规则

- 第一个字母必须是 英文字母 A-Z 、 a-z 或下划线（_）。
- 第二个字母起（如果有）则必须是 英文字母 A-Z 或、a-z 、下划线或数字0-9。
- 不能使用 C 的 关键字（keywords），如 int、long、return等。
- C 语言的标识符区分大小写，即：laowei 和 LaoWei 是两个不同的变量名。
- C89/C90标准规定编译器要至少支持 31 个字符以上作为标识符。
- C99及以后标准规定编译器要至少支持 63 个字符以上作为标识符。

合法的标识符示例

```
laowei      wei_ming_ze    ei123       a1b2c3
_abc        a1             x           y
```

不合法的标识符示例:

```
1a          $a
```

常用的标识符的命名方法：

1. 使用英文单词的名词等作为变量名（建议使用），也可是使用汉语拼音作为变量名。
2. 当有多个英文组合时使用小写加下划线。如：student_name_age。
3. 当有多个英文组合时使用**大驼峰命名法**，每个单词首字母大写。如：StudentNameAge。

4. 当有多个英文组合时使用**小驼峰命名法**，第一个单词小写，后续单词首字母大写。如：`studentNameAge`。

C 语言的关键字(保留字)不能用作标识符。

C11 的关键字和保留字

```
auto      if        unsigned
break     inline    void
case      int       volatile
char      long      while
const     register  _Alignas
continue  restrict  _Alignof
default   return    _Atomic
do        short     _Bool
double    signed    _Complex
else      sizeof    _Generic
enum      static    _Imaginary
extern    struct    _Noreturn
float     switch    _Static_assert
for       typedef   _Thread_local
goto      union
```

练习：

挑选出下列不合法的变量名

```
123L     snoopy_and_me       WHILE       volatile

__file__    printf      _1a2b3c4d        long_long

goto_room     longlongago
```

6. ASCII 编码

本节课我们来讲解关于英文编码的知识。

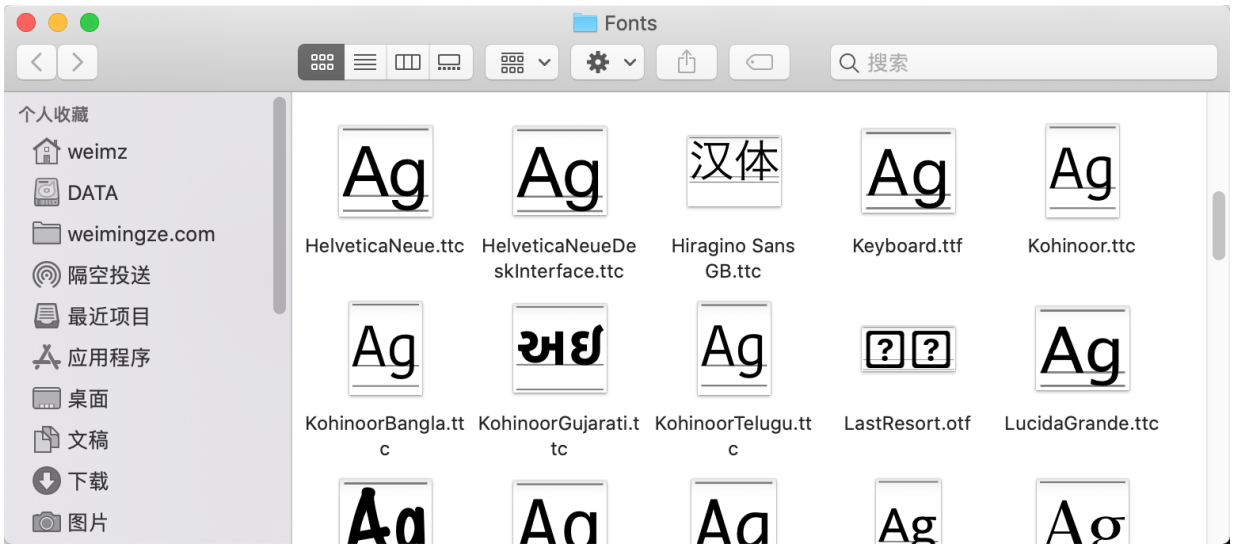
在计算机系统中，我们在屏幕上看到的每一个字就是一张图片。这些图片存储在字体库中。

各个系统字体库的位置：

- Windows 系统
  - 位于：`C:\Windows\Fonts\`
- MacOS 系统
  - 位于：`/System/Library/Fonts/`

- Ubuntu Linux 系统
  - 位于：`/usr/share/fonts/`

MacOS 系统的字体库如下图所示:



为了准确定位些图片的位置，计算机系统为每一张图片都定义了一个唯一的编号，我们把这个编号叫做**编码（Code）**。

字体的编码是一个无符号类型的整数，这个数值是从零开始的。

对于英文的编码，计算机中通常使用 **美国信息交换标准代码** 进行编码和通信。ASCII 全称是 American Standard Code for Information Interchange，中文翻译为：美国信息交换标准代码。

在 MacOS 或 Linux 的终端中，输入 `man ascii` 命令就可以查看到这个 128 个字符的编码值和对应的字符。如在Linux 下是使用如下命令:

```
man ascii
```

显示结果如下：

```
Oct   Dec   Hex   Char                        Oct   Dec   Hex   Char
────────────────────────────────────────────────────────────────────────
000   0     00    NUL '\0' (null character)   100   64    40    @
001   1     01    SOH (start of heading)      101   65    41    A
002   2     02    STX (start of text)         102   66    42    B
003   3     03    ETX (end of text)           103   67    43    C
004   4     04    EOT (end of transmission)   104   68    44    D
005   5     05    ENQ (enquiry)               105   69    45    E
006   6     06    ACK (acknowledge)           106   70    46    F
007   7     07    BEL '\a' (bell)             107   71    47    G
010   8     08    BS  '\b' (backspace)        110   72    48    H
011   9     09    HT  '\t' (horizontal tab)   111   73    49    I
012   10    0A    LF  '\n' (new line)         112   74    4A    J
```


013	11	0B	VT	'\v' (vertical tab)	113	75	4B	K			
014	12	0C	FF	'\f' (form feed)	114	76	4C	L			
015	13	0D	CR	'\r' (carriage ret)	115	77	4D	M			
016	14	0E	SO	(shift out)	116	78	4E	N			
017	15	0F	SI	(shift in)	117	79	4F	O			
020	16	10	DLE	(data link escape)	120	80	50	P			
021	17	11	DC1	(device control 1)	121	81	51	Q			
022	18	12	DC2	(device control 2)	122	82	52	R			
023	19	13	DC3	(device control 3)	123	83	53	S			
024	20	14	DC4	(device control 4)	124	84	54	T			
025	21	15	NAK	(negative ack.)	125	85	55	U			
026	22	16	SYN	(synchronous idle)	126	86	56	V			
027	23	17	ETB	(end of trans. blk)	127	87	57	W			
030	24	18	CAN	(cancel)	130	88	58	X			
031	25	19	EM	(end of medium)	131	89	59	Y			
032	26	1A	SUB	(substitute)	132	90	5A	Z			
033	27	1B	ESC	(escape)	133	91	5B	[			
034	28	1C	FS	(file separator)	134	92	5C	\	'\\'		
035	29	1D	GS	(group separator)	135	93	5D	]			
036	30	1E	RS	(record separator)	136	94	5E	^			
037	31	1F	US	(unit separator)	137	95	5F	_			
040	32	20	SPACE		140	96	60	`			
041	33	21	!		141	97	61	a			
042	34	22	"		142	98	62	b			
043	35	23	#		143	99	63	c			
044	36	24	$		144	100	64	d			
045	37	25	%		145	101	65	e			
046	38	26	&		146	102	66	f			
047	39	27	'		147	103	67	g			
050	40	28	(		150	104	68	h			
051	41	29	)		151	105	69	i			
052	42	2A	*		152	106	6A	j			
053	43	2B	+		153	107	6B	k			
054	44	2C	,		154	108	6C	l			
055	45	2D	-		155	109	6D	m			
056	46	2E	.		156	110	6E	n			
057	47	2F	/		157	111	6F	o			
060	48	30	0		160	112	70	p			
061	49	31	1		161	113	71	q			
062	50	32	2		162	114	72	r			
063	51	33	3		163	115	73	s			
064	52	34	4		164	116	74	t			
065	53	35	5		165	117	75	u			
066	54	36	6		166	118	76	v			
067	55	37	7		167	119	77	w			
070	56	38	8		170	120	78	x			
071	57	39	9		171	121	79	y			
072	58	3A	:		172	122	7A	z			
073	59	3B	;		173	123	7B	{			
074	60	3C	<		174	124	7C	|			
075	61	3D	=		175	125	7D	}			
076	62	3E	>		176	126	7E	~			
077	63	3F	?		177	127	7F	DEL			

这里对每一个英文字符进行了编码：如`z`的编码值是172（八进制）、122（十进制）、7A（十六进制）。

取值范围：

- 编码值在 `0~31`(十进制)的字符是控制字符，通常用于控制打印机和终端输出，如换行 `\n` 的值是 10，制表符 `\t` 的值是 `9`。
- 编码值为 `32`(十进制) 的的字符是**空格**。
- 编码值在 `48~57`(十进制)的字符是数字 `0~9`。
- 编码值在 `65~90`(十进制)的字符是英文字母 `A~Z`。
- 编码值在 `97~122`(十进制)的字符是英文字母 `a~z`。
- 其它编码值的都是各种符号。

英文的字母编码值在 0~127 之间，使用 7 个位就可以保存一个字符，在 C 语言中通常使用 `char` 类型来保存一个字符。最高位为 `0`。

这个编码表不需要死记硬背。我一般只记住了 `0`、`A` 和 `a` 的值十进制表示分别是 `48`、`65`和 `97`，十六进制是`30`、`41`和 `61`，其它都可以通过计算得到。

练习:

1. 字符 `B` 的 ASCII 编码值是多少（十进制和十六进制的值）？
2. 字符 `e` 的 ASCII 编码值是多少（十进制和十六进制的值）？
3. 字符 `8` 的 ASCII 编码值是多少（十进制和十六进制的值）？
4. 换行符 `\n` 的 ASCII 编码值是多少?

7. 字面值

字面值（Literal） 是直接在代码中表示的固定值（常量），它们的值和类型在编译时就已确定。字面值用于表示常量数据，例如整数、浮点数、字符或字符串等。

常量

常量是指在程序中永远不会改变的量，在整个程序中不允许修改。

在下面的程序中

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int age;       // 声明一个整型变量，初始值不确定。
    float price = 0.0;  // 声明一个浮点型变量，初始值为0.0
    int score1, score2 = 60; // 声明两个整型变量
    char gender = 'F';  // 声明一个字符型变量，初始值为'F'(值为70)。
                        // 字符 'F' 后续讲解。
```



```
    printf("age: %d, price: %f, score1: %d, score2: %d, gender: %d\n",
        age, price, score1, score2, gender
    );
    return 0;
}
```

其中的`0.0`、`60`、`'F'`和 return 后面的`0`都是字面值。

C 语言中的字面值类型有：

1. 整数字面值
2. 浮点数字面值
3. 字符字面值
4. 字符串字面值

1. 整数字面值

整数字面值有三种写法，即：**十进制写法**、**八进制写法** 和 **十六进制写法**，如：。

```
// 十进制写法:
100    -1     0

// 八进制写法，0开头。
0177

// 十六进制写法，以 0x 或 0X 开头，后跟 0-9, A-F或a-f（表示 10 ~ 15)
0x1    0X2    0xa    0xA     0xFF    0XFFFFFFFF
```

上述写法中，无论是 **十进制写法** 还是 **十六进制写法**，在计算机内存存储时都是用整数（int 类型）进行存储的。

在 C 语言中，整数字面值不但有值，而且还有存储这些值的方式方法，即：数据类型。默认整数字面值的数据类型都是整数。如果我们要告诉编译器这个类型不是整数，则需要再整数字面值的后面添加后缀来表示不同的类型。

整数字面值后缀写法如下表所示：

后缀	类型	示例
无后缀	int（默认类型）	`1234`或 0x1234
`u` 或 `U`	unsigned（无符号）	`1234U`
`l` 或 `L`	long（长整型）	`1234L`
`ll` 或 `LL`	long long（长长整型）	`1234LL`
`ul` 或 `UL`	unsigned long	`1234UL`
`ull` 或 `ULL`	unsigned long long	`1234ULL`
`lu` 或 `LU`	unsigned long（与 UL 相同）	`1234LU`
`llu` 或 `LLU`	unsigned long long（与 ULL 相同）	`1234LLU`

2. 浮点数字面值

浮点数字面值有两种写法，即：**小数写法**、**指数写法**。如：

```
// 小数写法:
3.14

// 指数写法:
0.314E1 或 0.314e1  // 表示 0.314 x 10 的 1 次方
```

浮点数字面值后缀写法如下表所示：

后缀	类型	示例
无后缀	double（默认类型）	`3.14`
`f` 或 `F`	float	`3.14F`
`l` 或 `L`	long double	`3.14L`

3. 字符字面值

字符型字面值用单引号（`'`）括起来的单个字符。默认返回整数数据类型。如：

```
// '' 中间放入一个 ascii 字符
'A'     // 值为 65（十进制）
' '     // 空格，值为 32（十进制）
'0'     // 字符 0，值为 48（十进制）
```

```
// 转义序列写法：使用 \ 开头
'\n'    // 换行符，值为 10（十进制）（详见ASCII编码表）
'\t'    // 制表符，值为 9（十进制）

// 八进制转义序列写法：一个 \ 后跟 1~3个八进制数字
'\0'    // 空字符，值为 0 (ASCII 0)
'\12'   // 换行符'\n'，值为 12（八进制）、10（十进制）
'\101'  // 字符 'A'，值为101（八进制）、65（十进制）、

// 十六进制转义序列写法：一个 \x 后跟 1~2个十六进制数字
'\x0'    // 空字符，值为 0 (ASCII 0)
'\x41'   // 字符 'A'，值为101（八进制）、65（十进制）、
'\xA'    // 换行符，值为 10（十进制）
'\xFF'   // 值为 255（十进制），字符型字面值的最大值。
```

> 字符型字面值默认返回整数数据类型。

4. 字符串字面值

字符串字面值用英文的双引号（"）括起来的字符序列。默认返回 `const char const *` 类型（后面会讲），如：

```
"Hello"   // 字符串，末尾自动加 '\0'，长度为 6 个字节。
"Hello\tWorld\n"     // 包含转义字符的字符串。
```

> 字符串字面值实际上是 `char` 数组，并以 `\0`（空字符）结尾。

关于字符串的内容后面会讲

总结：

字面值 是 C 语言中直接表示固定值的语法结构，编译器会根据其形式推断数据类型。理解字面值有助于正确使用常量数据。

练习：

下面这些字面值是那种类型的数据。

```
3.14     3.14f    '0'     200     200U    200L

200UL    1LL     6.18E-1   2.71828E0F     "hello"
```

第四章、基本输入输出函数

什么是函数

函数（Function）是实现了一个功能的代码块。函数有函数的名字，函数名代表一段代码块的起点。也代表一个功能。

函数都有自己的名字，如：`printf`、`main`等。不同的名字代表不同的功能。

> 函数对应英文是 Function，其实函数翻译成中文为 **功能** 会更好理解（个人见解）。

C 语言中已经实现了一些函数供编码人员调用，这些函数我们称之为标准库函数。

函数的调用

函数只有调用了，函数内部的代码才能够执行，才能够发挥作用。

函数调用的语法

```
函数名(表达式1, 表达式2, 表达式3, ...)
```

语法说明:

1. `函数名` 代表函数的功能。
2. `()` 代表调用此函数（让函数执行）。
3. `()` 括号内的表达式计算的结果是给函数提供数据（通常叫做实际调用参数），即函数的输入。多个数据给函数时需要用英文的逗号分隔符(`,`)分隔。一个函数需要多少个参数是由这个函数的声明者（编写函数的人）决定的。
4. 函数调用是一个表达式，它可以返回一个值。

示例:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello World!\n");
    printf("age:%d\n", 35);
```

```
    return 0;
}
```

上述程序中，第一个 `printf` 函数只有一个参数为 `"Hello World!\n"`，第二个 `printf` 函数有两个参数为 `"age:%d\n"` 和 `35`，中间用英文的逗号（`,`）分隔。

1. printf函数

`printf` 函数是用来在屏幕终端上打印一个标准输出，来让用户看到程序执行中运行的一些结果。

printf 的头文件

在使用库函数前，我们必须在当前的 `.c` 文件中使用 `#include <xxx.h>` 来包含库函数声明的头文件。C 语言的函数需要先声明再使用，而 printf 函数的声明在 头文件 `stdio.h`中。因此需要在 文件头部加上 `#include <stdio.h>`。

> `stdio` 含义为 标准输入输出（Standard Input/Output）。 `.h` 通常是头文件的后缀。

包含头文件的语法

```
#include <头文件名>
```

如，包含 *`stdio.h`* 这个文件的写法如下：

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

只要一次包含就可以多次使用此库中的所有函数了。

printf 的调用格式:

```
printf(格式字符串, 表达式1, 表达式2, ...);
```

例如:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("hello!\n");
    printf("name:%s, age:%d\n", "张三", 18);
```

```
    return 0;
}
```

执行结果如下：

```
hello!
name:张三, age:18
```

上述程序中第二个 `printf` 的第一个参数（格式字符串）中 `%s` 和 `%d`是**格式说明符**（也叫占位符），此处的数据需要使用第二个参数(`"张三"`)和第三个参数（`18`）填充。而第一个 `printf` 中 的第一个参数的字符串中没有`%`开头的**格式说明符**，则不需要额外的参数。

printf 格式说明符格式

```
%[flags][width][.precision][length]specifier
```

> 其中 中括号（`[]`） 括起来的部分表示可选项。

说明：

- `flags`：控制对齐、前缀等（如：`-`，`+`，`0`，`#`）。
- `width`：最小输出宽度（如：`%5d`）。
- `.precision`：精度（如：`%.2f` 保留两位小数）。
- `length`：数据类型长度修饰符（如：`%hd` 表示 `short`）。
- `specifier`：格式字符。

格式化字符串的 格式字符

格式字符	对应数据类型
%d	十进制整数（int）
%i	十进制整数（同 %d）
%u	无符号十进制整数
%o	八进制整数
%x	十六进制整数(字母小写)
%X	十六进制整数(字母大写)
%f	浮点数（默认保留小数点后六位）
%e	指数显示浮点数(字母小写)
%E	指数显示浮点数(字母大写)
%g	自动转换小数和指数格式(字母小写)
%G	自动转换小数和指数格式(字母大写)
%c	单个字符
%s	字符串
%p	指针（十六进制表示）
%%	表示一个百分号%

长度修饰符

修饰符	说明
%ld	用于 long int或 unsigned long int
%lld	用于 long long int或 unsigned long long int
%Lf	用于 long double

示例

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    float pi = 314.15926535;
    int x = 8815431;
    long long int lli= 1000;
```

```
    printf("---%d---\n", x);
    printf("---%u---\n", x);
    printf("---%i---\n", x);
    printf("---%o---\n", x);
    printf("---%x---\n", x);
    printf("---%X---\n", x);
    printf("---%f---\n", pi);
    printf("---%e---\n", pi);
    printf("---%g---\n", pi);
    printf("---%G---\n", pi);
    printf("---%lld---\n", lli);

    printf("======================\n");

    printf("---%10d---\n", x);
    printf("---%-10d---\n", x);
    printf("---%+10d---\n", x);
    printf("---%010d---\n", x);
    printf("---%hd---\n", x); // 转为short类型，丢失高位数据
    printf("---%#x---\n", x);
    printf("---%10.2f---\n", pi);
    printf("---%.3f---\n", pi);
    return 0;
}
```

执行结果：

```
---8815431---
---8815431---
---8815431---
---41501507---
---868347---
---868347---
---314.159271---
---3.141593e+02---
---314.159---
---314.159---
---1000---
======================
---   8815431---
---8815431   ---
---  +8815431---
---0008815431---
----31929---
---0x868347---
---    314.16---
---314.159---
```

练习：

1. 使用指数形式打印圆周率。
2. 使用小数形式打印圆周率，小数点后保留2位有效数字。

3. 使用小数形式打印圆周率，小数点后保留2位有效数字，共占据7个字符，数前空格补充`0`。

2. 取地址运算 &

运算符（Operator） 是一种特殊的符号，它用于表示运算规则。运算符可以将 C 语言中的字面值、变量、常量等结合在一起来进行运算并能够取得一个值。

& 运算符

作用：返回变量的内存地址。

计算机的内存是以字节(Byte) 位单位的，每个字节都对应一个不同的整数编号。这个编号称为地址。这个地址是从 0 开始的。 0 地址是第一个字节， 1 地址是第二个字节，以此类推。

对于 32 位的计算机，它的最大地址是 `0xFFFFFFFF`，即 4G 的大小。对于 64 位的计算机系统。它的最大地址是 `0xFFFFFFFFFFFFFFFF`。

取地址 & 运算符的语法:

```
& 变量
```

说明:

- `&` 用于返回变量在内存中的起始地址，如果使用 `int i;` 声明一个 int 类型的变量 i 占用 4 个字节，地址分别为 `0x????a5f4`、`0x????a5f5`、`0x????a5f6`、`0x????a5f7`。则 `&i` 返回的值为 `0x????a5f4`（这里 `????` 表示地址的高位字节的值，此处省略不写了）。

示例:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 0;
    int y = 100;
    double d1 = 3.14;
    double d2 = 2.71828;

    printf("&x:%p, &y:%p, &d1:%p, &d2:%p\n", &x, &y, &d1, &d2);

    return 0;
}
```

运行结果:


```
&x:0x7ffc6e94a5f0, &y:0x7ffc6e94a5f4, &d1:0x7ffc6e94a5f8, &d2:0x7ffc6e94a600
```

可见变量 x 的实际地址是 0x7ffc6e94a5f0，占用 4 个字节。变量 y 的实际地址是 0x7ffc6e94a5f4，占用 4 个字节。变量 d1 的实际地址是 0x7ffc6e94a5f8，占用 8 个字节。变量 d2 的实际地址是 0x7ffc6e94a600，占用 8 个字节。

各个变量的内存结构如下图所示：

```
                x        y           d1                d2
+----+     +--------+--------+-----------------+-----------------+     +----+
|    | ... |    0   |  100   |       3.14      |     2.71828     | ... |    |
+----+     +--------+--------+-----------------+-----------------+     +----+
0x0        a5f0     a5f4      a5f8              a600          a608 ...   0xFFFF
           &x       &y        &d1               &d2
```

> 这里用十六进制的后四位表示内存地址的具体值。

实验：

写程序定义多个字符型变量和多个短整型变量，然后打印器地址，画出它的内存结构图，

3. scanf函数

作用

从标准输入（终端）读取数据，根据指定的格式（如：%d、%f等）将读取的数据转换为相应的类型，放入到对应的变量中。

头文件：

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

调用格式:

```
scanf("格式字符串", 变量地址1, 变量地址2...);
```

返回值:

成功读取的数据的数量。若失败或输入结束则返回 `EOF`（`-1`）。

常用读取格式

格式符	说明
%d	读取整数（int）
%f	读取浮点数（float）
%lf	读取双精度浮点数（double）
%s	读取字符串（char*）
%c	读取单个字符串（char）
%u	读取无符号整数（unsigned int）

示例：

文件名: scanf_test.c

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int count = 0;
    float price = 3.1;

    printf("读取之前的count:%d, price:%f\n", count, price);
    printf("请输入整数: ");
    scanf("%d", &count);

    printf("请输入小数: ");
    scanf("%f", &price);
    printf("读取之后的count:%d, price:%f\n", count, price);

    return 0;
}
```

执行过程和结果

```
weimingze@mzstudio:~$ gcc -o scanf_test scanf_test.c
weimingze@mzstudio:~$ ./scanf_test
读取之前的count:0, price:3.100000
请输入整数: 10
请输入小数: 9.9
读取之后的count:10, price:9.900000
```

练习：

写一个程序，声明两个整数变量`x`、`y`，声明两个float 类型的变量`f1`和`f2`。读取两个整数放入x、y，读取两个小数放入`f1`和`f2`。

打印`x, y, f1, f2`的值。

第五章、运算符与表达式

运算符是 C 语言**表达式**的组成部分，每一个不同的运算符表示不同的运算规则。

表达式和语句的概念

表达式和**语句** 是编译原理里面的两个重要的概念。

什么是表达式

表达式是用于计算并一定能返回一个值的字面值、变量值或计算式，它一定能返回一个结果(数字、字符串、指针等)，如 `1 + 2` 返回 `3`。

什么是语句

语句 是程序执行的单位。通常一个语句作为一个完整的指令序列来执行，**表达式** 则是组成语句的基本单位。

白话文解释：

> **表达式**就相当如现代汉语里面的字、词或短语，是组成一句话的基本单位。**语句**就相当于用字和词组成的一个完整的一句话，它能表示一个完整的意思。

1. 算术运算符

C 语言中的算术运算符 有如下 5 个。

```
+     // 加法
-     // 减法
*     // 乘法
/     // 除法
%     // 求余数（取模）运算（不能用于浮点数运算）
```

算术运算符的语法

```
左操作数   算术运算符  右操作数
```

什么是操作数

操作数（operand） 是指参与运算的数据。

说明

- 上述**算术运算符** 是二元运算符，即必须有两个操作数运算后求值。
- **二元运算符** 是指由两个数据元素参加运算的运算符。
- 二元算术运算符是**先左后右**的结合性，即左侧的操作数先求值，然后是右侧的操作数在求值，然后再进行算术运算。

在 C 语言中，除**赋值运算符**（后面会讲）外，二元运算符都是**先左后右**的结合性，即左操作数先算。

示例

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 7;
    int y = 2;
    float f = 3.1;

    printf("%d + %d = %d\n", x, y, x + y);
    printf("%d - %d = %d\n", x, y, x - y);
    printf("%d * %d = %d\n", x, y, x * y);
    printf("%d / %d = %d\n", x, y, x / y);
    printf("%d %% %d = %d\n", x, y, x % y);

    printf("%d + %f = %f\n", x, f, x + f);
    printf("%d - %f = %f\n", x, f, x - f);
    printf("%d * %f = %f\n", x, f, x * f);
    printf("%d / %f = %f\n", x, f, x / f);
    printf("1 + 2 * 3.14 = %f\n", 1 + 2 * 3.14);

    return 0;
}
```

运行结果：

```
7 + 2 = 9
7 - 2 = 5
7 * 2 = 14
```

```
7 / 2 = 3
7 % 2 = 1
7 + 3.100000 = 10.100000
7 - 3.100000 = 3.900000
7 * 3.100000 = 21.699999
7 / 3.100000 = 2.258065
1 + 2 * 3.14 = 7.280000
```

上述程序中的`+`、`-`、`*`、`/`和数学中的意义是一样的，计算中也是乘、除先算，加、减后算。

求余数（%）是整数除法后得到整数的商后剩余的整数，比如7个苹果分给三个人，每个人分2个，剩余1个，这个`1`就是`7`对`3`求余数的结果。

说明：

1. 在 C 语言中，整数和整数运算的结果也是整数类型不会提升为浮点数。因此`7 / 3`的结果是`2`而不是浮点数`2.333333`。
2. 不同类型的数据类型进行运算，默认会类型提升，即：整数和浮点数运算结果是 浮点数，如：`7 / 3.0`的结果就是浮点数`2.333333`了。
3. 求余数运算符（%）不能用于浮点数运算。

练习

1. 写一个程序，输入两个整数，打印这个两个数相加、相减、相乘、相除和求余数的值。
2. 写一个程序，输入两个小数，打印这个两个数相加、相减、相乘、相除值。

2. 赋值运算符

赋值运算符 是构成**赋值表达式**的基本运算符。

什么是赋值表达式

赋值表达式 是用来修改变量（或其它数据）的值的一个表达式。通常在赋值表达式的末尾加一个分号（;） 形成一个可单独执行的表达式语句。

赋值运算符有如下几种：

```
=   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
```

赋值表达式的语法

```
左操作数    赋值运算符    表达式
```

说明：

- **左操作数** 通常是**变量**、 **数组索引**或**指针解引用**等（后面会讲）。
- 赋值表达式是先右后左的结合性，即右侧优先运算，将表达式运行完毕后的最终结果赋值给左操作数。

例如1

```
int x;

x = 100 + 200;
```

上述代码中 `x = 100 + 200` 是赋值表达式，最终变量 `x` 的值被改为 `300`。

例如2

```
int x;
int y;

y = x = 100 + 200;
```

上述代码中 `x = 100 + 200` 先计算，变量 `x` 的值被改为 `300`，然后表达式 `x = 100 + 200` 的返回值就是 x 的值 300，再使用赋值表达式赋值给 y ，最终变量 y 的值也是 300。

问题：

现在有一个变量 x 此时它的值是未知的。我现在要在它原有的值的基础上增加 5 我该怎么办呢？起始我们可以这样写代码如下：

```
x = x + 5;
```

右表达式 `x + 5` 先算出值后再赋值给 x，此时 x 的值就被修改为原来的值加上 5 了。当然上述赋值表达式可以写成如下形式：

```
x += 5;  // 等同于 x = x + 5;
```

赋值表达式的其它 运算符列举如下：


```
左操作数 += 表达式      // 等同于:  左操作数 = 左操作数 + 表达式
左操作数 -= 表达式      // 等同于:  左操作数 = 左操作数 - 表达式
左操作数 *= 表达式      // 等同于:  左操作数 = 左操作数 * 表达式
左操作数 /= 表达式      // 等同于:  左操作数 = 左操作数 / 表达式
左操作数 %= 表达式      // 等同于:  左操作数 = 左操作数 % 表达式
// 以下是位运算相关的运算符(后续小节内容学完才会理解)。
左操作数 &= 表达式      // 等同于:  左操作数 = 左操作数 & 表达式
左操作数 |= 表达式      // 等同于:  左操作数 = 左操作数 | 表达式
左操作数 ^= 表达式      // 等同于:  左操作数 = 左操作数 ^ 表达式
左操作数 <<= 表达式     // 等同于:  左操作数 = 左操作数 << 表达式
左操作数 >>= 表达式     // 等同于:  左操作数 = 左操作数 >> 表达式
```

示例

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 7,y = 2;
    int result;

    result = x + y; // 此时 result 值为 9
    printf("result: %d\n", result);

    result += 5; // 此时 result 值为 14
    printf("result: %d\n", result);

    return 0;
}
```

运行结果

```
result: 9
result: 14
```

练习:

写一个程序，输入矩形（长方形）的宽度和高度，打印矩形的面积和周长。

3. 自增、自减运算符

自增（`++`）和自减（`--`）运算符用于对整数变量的值进行加1或减1操作。它们是一元运算符（只需一个操作数），且会直接修改操作数的值。

作用:

将变量的值增加 1 或减少 1。常用来简化替代如用 `i++` 来替代 `i+=1`。

自增、自减运算符两种语法：

1. 前置自增、自减运算。
2. 后置自增、自减运算。

1、 前置自增、自减运算

作用

先对变量进行操作，然后返回变量操作后的值。

前置自增、自减运算语法：

```
++ 变量
-- 变量
```

前置自增、自减运算示例：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 5;
    int y;

    y = ++x;
    printf("x: %d, y: %d\n", x, y);

    y = --x;
    printf("x: %d, y: %d\n", x, y);

    return 0;
}
```

运行后的结果：

```
x: 6, y: 6
x: 5, y: 5
```

2、 后置自增、自减运算

作用

先对变量进行操作，然后**返回变量操作前的值**。

后置自增、自减运算语法：

```
变量 ++
变量 --
```

后置自增、自减运算示例：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 5;
    int y;

    y = x++;
    printf("x: %d, y: %d\n", x, y);

    y = x++;
    printf("x: %d, y: %d\n", x, y);

    return 0;
}
```

运行后的结果：

```
x: 6, y: 5
x: 7, y: 6
```

可见： 后置自增和自减运算会返回变量运算前的值。

注意：

在实际开发过程中应尽量减少在一个语句中多次使用 **前置自增和自减运算** 和 **后置自增和自减运算**。由于各个编译器的具体实现方式不同，可能会出现不可预知的结果。

如下程序在 MacOS 上使用 clang 编译器 和 Linux 下的 gcc 编辑器下运行结果会有所不同。程序如下：

文件名：`myprog.c`

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 5;
    int y;
```



```
    y = ++x + x++;
    printf("x: %d, y: %d\n", x, y);

    return 0;
}
```

Linux 下的 gcc 编译后运行的结果如下：

```
weimingze@mzstudio:~$ gcc -o myprog myprog.c
weimingze@mzstudio:~$ ./myprog
x: 7, y: 13
```

MacOS 下的 clang 编译后运行的结果如下：

```
weimz@MacBook-Pro c % clang -o myprog myprog.c
weimz@MacBook-Pro c % ./myprog
x: 7, y: 12
```

可见 此时 表达式 `++x + x++` 的运行结果不同。解决的方法是使用临时变量明确具体的语义。改写如下：

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 5;
    int y;
    int temp1;
    int temp2;

    temp1 = ++x;
    temp2 = x++;

    y = temp1 + temp2;
    printf("x: %d, y: %d\n", x, y);

    return 0;
}
```

gcc 编译后和 clang 编译后的运行的结果相同，如下：

```
x: 7, y: 12
```

练习：

写一个程序，输入一个人的年龄。

1. 计算并打印明年的年龄。

2. 计算并打印后年的年龄。

4. 关系运算符

关系运算符（Relational operators）用于比较两个操作数的值，返回它们是否成立的一个**布尔值**。

布尔值

布尔值 是由 19 世纪英国著名的数学家和逻辑学家乔治·布尔（George Boole，1815.11.2～1864.12.8）提出，用于表示逻辑关系。

布尔值分为两种：

- 真（true），表示成立。
- 假（false），表示不成立。

C 语言中的布尔值：

在C语言中用数值表示布尔值：

- **零值**表示假（false），如：0 或 0.0。
- **非零值**表示真（true），如 1 或 3.14。

关系运算符

```
<       // 小于运算符
>       // 大于运算符
<=      // 小于等于运算符
>=      // 大于等于运算符

==      // 等于运算符
!=      // 不等于运算符
```

其中 `<`、`>`、`<=`、`>=` 四个运算符的优先级高于 `==`、`!=` 的优先级，即先比较然后才判断是否相等。

语法

```
左表达式 关系运算符 右表达式
```

说明

关系运算符运算后用整数 0 代表**假**，用整数 1 表示**真**。

示例：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("100 < 200 的结果: %d\n", 100 < 200);
    printf("100 > 200 的结果: %d\n", 100 > 200);
    printf("100 <= 200 的结果: %d\n", 100 <= 200);
    printf("100 >= 200 的结果: %d\n", 100 >= 200);
    printf("100 == 200 的结果: %d\n", 100 == 200);
    printf("100 != 200 的结果: %d\n", 100 != 200);

    return 0;
}
```

运行结果

```
100 < 200 的结果: 1
100 > 200 的结果: 0
100 <= 200 的结果: 1
100 >= 200 的结果: 0
100 == 200 的结果: 0
100 != 200 的结果: 1
```

注意事项

- 等于运算符（`==`）是双等号是关系运算符，而复制运算符（`=`）是单等号，两者完全不同。但有时在表达式中会出现 `x == 100` 写成 `x = 100`，导致 x 的值被修改且条件一直成立，比较好的写法是 写成 `100 == x`，一旦少写一个等号，则会出现 赋值表达式中左侧出现常量，在编辑阶段就会报错。
- 由于浮点数精度问题，应避免直接用`==`比较浮点数，要比较浮点数是否相等通常使用将两个操作数相减，再判断结果的绝对值是否小于某个误差范围，如果小于误差范围则认为两个浮点数相等。否则就是不相等。
- 关系运算符的优先级低于算术运算符但高于赋值运算符。

练习：

写程序，假设一个商品在拍卖会上竞拍。现在有两个人竞拍。请输入你的竞拍价格，再输入另外一个人的竞拍价格。打印竞拍结果：1 表示你竞拍成功。0 表示你竞拍失败。

5. 逻辑运算符

什么是逻辑运算?

逻辑运算是基于布尔（真/假、1/0）的基本运算，用于在逻辑表达式中求值。

逻辑运算有三种:

- 逻辑与运算（运算符：`&&`）。
- 逻辑或运算（运算符：`||`）。
- 逻辑非运算（运算符：`!`）。

逻辑与运算（&&）：

逻辑与运算是参与运算的两个表达式都为真（非零）时，结果为真（`1`）；否则结果为假（`0`）。

语法

```
左表达式 && 右表达式
```

示例:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int score = 80;

    int result = score >= 60 && score <= 100;
    printf("%d\n", result);

    score = 59;
    result = score >= 60 && score <= 100;
    printf("%d\n", result);

    return 0;
}
```

运行结果:

```
1
0
```

说明:

逻辑与运算是短路运算，如果第一个表达式为假，则直接返回假值，不再计算第二个表达式的值。

注意：

判断 `score` 变量的值是否在 `60 ~ 100` 之间不能使用表达式 `60 <= score <= 100`，因为条件运算符的结合性是**先左后右**。因此会先算出 `60 <= score` 的值，假设结果为`x`，然后才计算 `x <= 100` 的值。这个逻辑不符合数学中判断区间的计算规则。

逻辑或运算（||）：

逻辑或运算是参与运算的两个表达式只要有一个为真（非零）时，结果就为真（`1`）；两个表达式都为假值时结果才为假值（`0`）。

语法

```
左表达式 || 右表达式
```

示例:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int score;

    printf("请输入成绩: ");
    scanf("%d", &score);

    int result = score < 0 ||  score > 100;
    printf("%d\n", result);

    return 0;
}
```

运行结果:

```
weimingze@mzstudio:~$ gcc -o myprog myprog.c
weimingze@mzstudio:~$ ./myprog
请输入成绩: 80
0
weimingze@mzstudio:~$ ./myprog
请输入成绩: -10
1
weimingze@mzstudio:~$ ./myprog
请输入成绩: 200
1
```

说明:

逻辑或运算是短路运算，如果第一个表达式为真，则直接返回真值，不再计算第二个表达式的值。

逻辑非运算（`!`）：

逻辑非运算是参与运算的一个表达式取**非操作**（真变假，假变真）。即表达式为真（非零）时，结果为假（`0`）；表达式为假（零值）时，结果为真（`1`）。

语法

```
! 表达式
```

示例:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 100;
    int y = 0;

    printf("!x: %d\n", !x);
    printf("!y: %d\n", !y);
    printf("!!x: %d\n", !!x);

    return 0;
}
```

运行结果:

```
!x: 0
!y: 1
!!x: 1
```

说明：

- 逻辑运算符的优先级是：`!` 大于 `&&`， `&&` 大于 `||`。
- 逻辑运算符通常用于**迭代语句**和**选择语句**中。
- C语言中，任何非零值都被视为 **真** ，只有零值（`0` 或 `0.0`）被视为 **假**。

练习

写一个程序，任意输入两个整数。

1. 求这两个整数 逻辑与运算、逻辑或运算 的值并打印结果。
2. 求每个数字的逻辑非运算的值并打印结果。

6. 位运算符

位运算是指对整数在二进制位级别上进行的运算。

位操作经常用在嵌入式系统和 Linux 内核驱动程序中。

C 语言位运算的运算符有如下 6 种：

```
&      // 按位与运算
|      // 按位或运算
^      // 按位异或运算
<<     // 左移运算
>>     // 右移运算
~      // 按位取反运算
```

其中 `&`、`|`、`^`、`<<`、`>>` 是二元运算符，且左右两个操作数一定是整数。

语法格式如下：

```
左表达式  二元位运算符  右表达式
```

`~` 是一元运算符

语法格式如下：

```
~ 表达式
```

6.1 按位与运算（&）

按位与运算是两个操作数的对应位都为 `1` 时，该位结果才为 `1`；否则该位结果为 `0`。

按位与运算的特点是一个位和 `0` 与则变为 `0`， 和 `1` 与则不变。

示例

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    short int x = 0x29;  // 00000000 00101001
    short int y = 0x27;  // 00000000 00100111
    short int z;

    z = x & y;
    printf("z: 0x%04x\n", z);

    return 0;
}
```

执行结果如下：

```
z: 0x0021
```

运算过程

```
z = x & y;

x:       00000000 00101001
y:    &  00000000 00100111
     --- -------- --------
z:       00000000 00100001
```

6.2 按位或运算（|）

按位或运算是两个操作数的对应位都为 `0` 时，该位结果才为 `0`；否则该位结果为 `1`。

按位或运算的特点是一个位和 `1` 或则变为 `1`，和 `0` 或则不变。

示例

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    short int x = 0x29;  // 00000000 00101001
    short int y = 0x27;  // 00000000 00100111
    short int z;

    z = x | y;
    printf("z: 0x%04x\n", z);

    return 0;
}
```

执行结果如下：

```
z: 0x002f
```

运算过程

```
z = x | y;

x:       00000000 00101001
y:    |  00000000 00100111
     --- -------- --------
z:       00000000 00101111
```

6.3 按位异或运算（^）

按位异或运算是两个操作数的对应位不相同时，该位结果为 `1`；相同该位结果为`0`。

按位异或运算的特点是一个位和 `1` 异或则翻转（`1`变 `0`，`0`变 `1`） ， 和 `0` 异或则不变。

示例

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    short int x = 0x29;  // 00000000 00101001
    short int y = 0x27;  // 00000000 00100111
    short int z;

    z = x ^ y;
    printf("z: 0x%04x\n", z);

    return 0;
}
```

执行结果如下：


```
z: 0x000e
```

运算过程

```
z = x ^ y;

x:       00000000 00101001
y:    ^  00000000 00100111
     --- -------- --------
z:       00000000 00001110
```

6.4 左移运算（<<）

左移运算（<<）是将操作数的所有位向左侧移动指定的位数，高位溢出丢掉，低位补 0。

示例

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    short int x = 0x29;  // 00000000 00101001
    short int z;

    z = x << 1;
    printf("z: 0x%04x\n", z);

    z = x << 2;
    printf("z: 0x%04x\n", z);

    return 0;
}
```

执行结果如下：

```
z: 0x0052
z: 0x00a4
```

运算过程

```
z = x << 1;

x:       00000000 00101001
     <<  00000000 00000001
     --- -------- --------
z:    0  00000000 01010010  <-- 补1个 0
```

```
z = x << 2;

x:       00000000 00101001
     <<  00000000 00000010
     --- -------- --------
z:   00  00000000 10100100  <-- 补2个 0
```

6.5 右移运算（>>）

右移运算（>>）是将操作数的所有位向右侧移动指定的位数，低位溢出丢掉，对于高位有这两种不同的填补方法：

- 对于无符号数：高位补 0。
- 对于有符号数：高位补符号位（算术右移）。

示例

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    short int x = 0x29;  // 00000000 00101001
    short int z;

    z = x >> 1;
    printf("z: 0x%04x\n", z);

    z = x >> 2;
    printf("z: 0x%04x\n", z);

    return 0;
}
```

执行结果如下：

```
z: 0x0014
z: 0x000a
```

运算过程

```
z = x >> 1;

x:       00000000 00101001
     >>  00000000 00000001
     --- -------- --------
z:       00000000 00010100  1（溢出1个位）
```

```
z = x >> 2;

x:       00000000 00101001
     >>  00000000 00000010
     --- -------- --------
z:       00000000 00001010  01（溢出2个位）
```

6.6 按位取反运算（~）

按位取反运算（`~`）是将操作数对应的位翻转（`1` 变 `0`，`0` 变 `1`）。

示例

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    short int x = 0x29;  // 00000000 00101001
    short int z;

    z = ~x;  // z = 11111111 11010110
    printf("z: 0x%04x\n", z);

    return 0;
}
```

执行结果如下：

```
z: 0xffffffd6
```

运算过程

```
z = ~x;

x:    ~  00000000 00101001
     --- -------- --------
z:       11111111 11010110
```

练习1:

输入两个无符号的整数，计算两个整数位与、位或、位异或的值并打印结果。

练习2:

输入一个整数，用有符号变量 x 绑定。

1. 将 x 的第 0 位置0 后打印结果。提示: `x &= (~1);`。

2. 将 x 的第 1 位置1 后打印结果。提示: `x |= 0x01 << 1;`。
3. 将 x 的第 2 位取反后打印结果。提示：任意为和 `1` 异或翻转，和 `0` 异或不变。

7. 条件运算符

条件运算符用于根据布尔表达式的值，执行不同的表达式并返回结果。

运算符：`?:`

> C 语言中唯一的一个三元运算符

语法格式:

```
表达式1 ? 表达式2 : 表达式3
```

> 表达式 2 或 表达式 3 最多只能运行一个。

说明:

1. 执行 `表达式1`，判断真或假。
2. 如果 `表达式1` 返回真值，则执行 `表达式2` 并返回 `表达式2` 的结果。否则返回 `表达式3` 的结果。

示例:

写一个程序，输入成绩（0~100分），如果成绩大于等于 60 则成绩为 100 分，否则成绩为0分。打印最终成绩。

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int score;

    printf("请输入成绩: ");
    scanf("%d", &score);

    score = score >= 60? 100 : 0;

    printf("成绩是: %d\n", score);
    return 0;
}
```

条件运算符有自右向左的结合性，即当有多个条件运算符嵌套时自右向左运算，如：

```
score = score >= 60? 100 : score < 58 ? 0: 60;
```

则 `score < 58 ? 0: 60` 先进行运算。

练习：

写一个程序，输入一个学生的成绩，如果成绩大于 60 分打印及格。否则打印不及格。

> 提示： 使用表达式 `score >= 60 ? "及格": "不及格"` 返回不同的字符串。

8. 正负号运算符

正负号运算符（`+` 或 `-`）用于对数值型的操作数进行符号操作。

正负号运算符是一元运算符（只需一个操作数），通常正好不会改变其原有值，负号会改变原有操作数的符号。

语法格式如下：

```
+ 表达式
- 表达式
```

示例:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 9;
    int y = +x;
    int z = -x;

    printf("x:%d, y:%d, z:%d\n", x, y, z);

    x = -x;
    y = +x;
    z = -x;
    printf("x:%d, y:%d, z:%d\n", x, y, z);
    return 0;
}
```

运行结果：


```
x:9, y:9, z:-9
x:-9, y:-9, z:9
```

练习：

写程序，输入任意一个整数x，使用条件运算符 和 正负号运算符，计算并打印 x 的绝对值。

> **绝对值**：正数的绝对值是原数字，负数的绝对值是原数在取负值。

9. 逗号运算符

逗号运算符

逗号运算符用于将多个表达式一起，自左向右执行多个表达式，并返回最后表达式的值。

语法

```
表达式1, 表达式2, 表达式3...
```

> 逗号优先级最低。

示例：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x;

    // 赋值运算符优先级高于逗号优先级，使用括号提高逗号的优先级。
    x = (100 + 1, 200, 1+2);
    printf("%d\n", x);  // 打印 3
    return 0;
}
```

10. sizeof运算符

sizeof 运算符是一个用于计算变量、数据类型或表达式的结果的类型所占用的内存大小（以字节为单位）的运算符。

sizeof 运算符是编译时运算符，即在编译阶段计算结果，而非运行时计算。

语法

```
sizeof 一元表达式
// 或
sizeof (类型)
```

示例

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char ch = 'a';
    short int si = 100;
    int i = 1000;
    long int li;
    long long int lli;
    float f = 3.14;
    double d;

    printf("sizeof ch: %ld\n", sizeof ch);
    printf("sizeof si: %ld\n", sizeof si);
    printf("sizeof i: %ld\n", sizeof i);
    printf("sizeof li: %ld\n", sizeof li);
    printf("sizeof lli: %ld\n", sizeof lli);
    printf("sizeof f: %ld\n", sizeof f);
    printf("sizeof d: %ld\n", sizeof d);
    printf("sizeof 100: %ld\n", sizeof 100);
    printf("sizeof(char): %ld\n", sizeof(char));
    printf("sizeof(short int): %ld\n", sizeof(short int));
    printf("sizeof(int): %ld\n", sizeof(int));
    printf("sizeof(long int): %ld\n", sizeof(long int));
    printf("sizeof(long long int): %ld\n", sizeof(long long int));
    printf("sizeof(float): %ld\n", sizeof(float));
    printf("sizeof(double): %ld\n", sizeof(double));
    printf("sizeof(long double): %ld\n", sizeof(long double));
    return 0;
}
```

运行结果

```
sizeof ch: 1
sizeof si: 2
sizeof i: 4
sizeof li: 8
sizeof lli: 8
sizeof f: 4
sizeof d: 8
sizeof 100: 4
sizeof(char): 1
sizeof(short int): 2
sizeof(int): 4
```

```
sizeof(long int): 8
sizeof(long long int): 8
sizeof(float): 4
sizeof(double): 8
sizeof(long double): 16
```

11. 类型转换运算符

隐式类型转换

隐式类型转换是编译器按着默认规则将数据类型进行转换，如：

```c
float pi = 3.14159
int x = pi;  // 隐式将浮点数转化为整数。
```

在隐式类型转换过程中，小数转为整数会丢失小数部分，整型转为更小的整型数时会丢失高位字节。如：

```c
int x = 258;
unsigned char uc = x;  // 丢失高位的3个字节。
printf("%d\n", uc);  // 打印 2
```

类型转换运算符

类型转换运算符（也称为 强制类型转换）用于显式地将一种数据类型转换为另一种数据类型。它允许程序员手动控制类型转换的过程，避免编译器隐式转换可能带来的负面问题（如精度丢失等）。

语法

```
(类型) 表达式
```

示例

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int number = 5;
    float error_result = number / 2;  // 整数相除类型为整数。
    float result = (float)number / 2; // 浮点数和整数相除，类型为浮点数。

    printf("error_result: %f\n", error_result);
    printf("result: %f\n", result);
```

```
    return 0;
}
```

运行结果

```
error_result: 2.000000
result: 2.500000
```

练习：

如何修改以下代码，使得 result 的值为 3.5（而不是 2.0）？

```
int a = 7, b = 2;
float result = a / b;
```

选择：（）

- A) float result = (float)a / b;
- B) float result = a / (float)b;
- C) float result = (float)(a / b);
- D) float result = (float)a / (float)b;

12. 运算符优先级与结合性

C 语言的运算符有十五个**优先级**，优先级数值越小，优先级越高。当一个表达式有多个运算符时，**高优先级的运算符先进行运算**。对于相同的优先级的运算符，具体要看运算符的**结合性**来决定哪个运算符先计算。

C 语言运算符优先级表

优先级	运算符	说明	结合性
1	`++` `--` `()` `[]` `.` `->` `(type){list}`	后置自增/减、函数调用、数组下标、结构体(联合体)成员访问、结构体(联合体)指针成员访问、复合字面值(C99)	自左向右
2	`++` `--` `+` `-` `!` `~` `(type)` `*` `&` `sizeof` `_Alignof`	前置自增/减、正负号、逻辑非、按位取反、强制类型转换、解引用、取地址、求占用字节数、查询对齐(C11)	自右向左
3	`*` `/` `%`	乘、除、取模（求余数）	自左向右
4	`+` `-`	加、减	自左向右
5	`<<` `>>`	左移、右移	自左向右
6	`<` `<=` `>` `>=`	关系运算符比较	自左向右
7	`==` `!=`	关系运算符等于、不等于	自左向右

8	`&`	按位与	自左向右
9	`^`	按位异或	自左向右
10	`|`	按位或	自左向右
11	`&&`	逻辑与	自左向右
12	`||`	逻辑或	自左向右
13	`?:`	条件运算符（三元运算符）	自右向左
14	`= += -= *= /= %=` `<<= >>= &= ^= |=`	赋值运算符	自右向左
15	`,`	逗号运算符	自左向右

对于上述优先级表比较复杂，在不确定的情况下可以使用 括号`()`来提高表达式的优先级。因为括号的优先级最高。

在复杂的表达式中，可以使用括号嵌套的方式来提高运算符的计算顺序，最内侧的括号先进行进行计算，如：

```
int x = 2
int y;

y = (1 + (x << 1)) * 3;
```

练习：

请问如下程序中，变量 x、y、z 的值是多少？

```
int x = 1 + 2 / 3 * 4  - 5 + 6 * 7;
int y = 256 | 1 << 2;
int z = 5;
z *= z << 1;
printf("x:%d, y:%d, z:%d\n", x, y, z);
```

第六章、选择语句

什么是语句

语句 是程序执行的单位。通常一个语句作为一个完整的指令序列来执行。

C11 标准为我们提供的以下几种语句，供开发人员使用。它们分别是：

- 标签语句（labeled-statement）。
- 复合语句（compound-statement）。
- 表达式语句（expression-statement）。
- 选择语句（selection-statement）。
- 迭代语句（iteration-statement）。
- 跳转语句（jump-statement）。

以下几章我们先来研究 C 语言的这些语句。

选择语句

选择语句（Selection Statements），也称为条件语句或分支语句。选择语句用于根据条件决定程序的执行路径。它们允许程序在不同的条件下执行不同的代码块。

C 语言中选择语句的种类：

1. if 语句
2. switch 语句

选择语句的语法如下：

```
if ( expression ) statement
if ( expression ) statement else statement
switch ( expression ) statement
```

1. if 语句

作用：

根据条件表达式的值来选择性的执行语句。

if 语句的语法格式有两种:

```
// 第一种
if (表达式) 语句1
// 第二种
if (表达式) 语句1 else 语句2
```

> 语法中：`if` 和 `else` 是关键字。

if 语句语法说明

1. 括号中的 **表达式** 先进行计算，如果表达式结果为真值（非零值）则执行**语句1**（仅执行一条语句）。
2. 如果表达式结果为假值（零值）则如果有 else 子句（第二种语法）则执行**语句2**（也仅执行一条语句）。

示例1

写一个程序，输入你的年龄，如果年龄小于 18 岁，提示：**你还未满18岁，不能在我驾校报名！**，否则什么都不做。

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int age;

    printf("请输入年龄: ");
    scanf("%d", &age);

    if (age < 18)
        printf("你还未满18岁，不能在我驾校报名！\n");
    printf("程序退出\n");
    return 0;
}
```

运行结果1如下：

```
请输入年龄: 16
你还未满18岁，不能在我驾校报名！
程序退出
```

运行结果2如下：

```
请输入年龄：21
程序退出
```

可见语句 `printf("你还未满18岁，不能在我驾校报名！\n");` 实际受 if 语句控制。有 if 语句决定是否执行。而 `printf("程序退出\n");` 则不受 if 语句控制。因为 if 语句执行控制离它最近的一条语句。

if 语句实现二分支结构

使用 if 语句的第二种 `if - else` 的语法，可以实现二选一的分支结构。如：

写一个程序，输入你的年龄，如果年龄小于 18 岁，则提示：**你是未成年人!**，否则提示：**你是成年人!**。

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int age;

    printf("请输入年龄：");
    scanf("%d", &age);

    if (age < 18)
        printf("你是未成年人！\n");
    else
        printf("你是成年人！\n");
    printf("程序退出\n");
    return 0;
}
```

运行结果1如下：

```
请输入年龄：12
你是未成年人！
程序退出
```

运行结果2如下：

```
请输入年龄：20
你是成年人！
程序退出
```

练习：

写一个程序，输入一个学生的二科成绩:

1. 打印出最高分是多少?
2. 打印出最低分是多少?

2. if 语句中嵌入复合语句

上一节我们学习了 if 语句，细心的朋友会发现当我们在 if 的后面或 else 子句的后面写入多条语句时编译器就会报错。原因是 if 和 else 后面的语句部分只能放入一条语句。那么如何放入多条语句呢？这时我们需要使用 **复合语句**。本节我们先简单讲解复合语句的基本写法，后面我们再详细的介绍复合语句全部的语法和具体细节。

复合语句

复合语句 是使用 大括号 `{}`，将内部的0条、1条或多条语句组合成为一个整体、C 语言中将其作为一条语句放在语法中的某个位置。

复合语句可以写成如下的形式。

```
{
    int x = 100;
    x ++;
    printf("%d\n", x);
}
```

上述三条语句 `int x = 100;`、`x ++;`、`printf("%d\n", x);` 作为一条复合语句存在。

在复合语句内部可以声明变量，在复合语句内部声明的变量是复合语句内部的局部变量（后面会讲）（如上述：`int x = 100;`），这些局部变量在复合语句内部有效，复合语句执行完毕后会自动销毁。在此复合语句外部将无法使用。如上面的示例可以改写如下，且执行逻辑和结果完全一样：

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int age;

    printf("请输入年龄: ");
    scanf("%d", &age);

    if (age < 18) {
        printf("你是未成年人! \n");
    } else {
        printf("你是成年人! \n");
    }
```



```
    printf("程序退出\n");
    return 0;
}
```

练习：

写一个程序，输入一个整数x，判断这个整数是正数、负数还是零并打印结果。

3. if 语句实现多分支结构

if 语句也是语句，它可以放入到 C 语言语法中能放入语句的任何语句中，如（if 语句、for 语句等）。

前面我们已经学过了 if 语句的语法。在如下的语法中

```
if (表达式) 语句1 else 语句2
```

语句2 部分也可以放入一条 if 语句。这样一层一层的嵌套写法就形成了 C 语言的多分支结构。如将如下 if 语句的语法嵌套到上述语法的语句2部分。

```
if (表达式2) 语句2 else 语句3
```

这样就形成了如下的语法结构：

```
if (表达式1) 语句1 else if (表达式2) 语句2 else 语句3
```

将上述语法结构进行整理折行后就形成了如下的三分支的结构的语法（三个语句三选一执行）。

```
if (表达式1)
    语句1
else if (表达式2)
    语句2
else
    语句3
```

其实 if 语句还可以进一步嵌套，形成4分支、5分支或更多分支的应用结构。

这样嵌套的 if 语句的语法结构的最终执行顺序是程序自上而下来执行表达式，去找最先成立的一个，然后执行对应的语句后结束 if 语句。

示例：

写一个程序，输入一年中的月份(1~12)，打印这个月在哪儿个季度，如输入 2 就打印**春季**。如果输入非1~12的其它数字，则提示您输错了。

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int month;

    printf("请输入月份: ");
    scanf("%d", &month);

    if (month >= 1 && month <= 3)  {
        printf("春季\n");
    } else if (month >= 4 && month <= 6) {
        printf("夏季\n");
    } else if (month >= 7 && month <= 9) {
        printf("秋季\n");
    } else if (month >= 10 && month <= 12) {
        printf("冬季\n");
    } else {
        printf("您输入的月份不合法\n");
    }

    return 0;
}
```

练习:

写一个程序，输入一个学生的三科成绩: 1. 打印出最高分是多少？ 2. 打印出最低分是多少？ 3. 打印出均分是多少?

4. switch 语句

switch 语句用于多分支选择，根据表达式的值跳转到匹配的 case 标签语句处执行代码。

作用：根据表达式具体的取值，执行对应的语句（语句块）。

语法:

```c
switch (表达式) {
    case 常量表达式1: 语句1;
    case 常量表达式2: 语句2;
    case 常量表达式3: 语句3;
    ...
    default: 语句（other）;
}
```

> 语法中：`switch`、`case` 和 `default` 是关键字。

语法说明：

- `case` 标签语句可以有零个、一个、或多个。
- `default:` 标签语句只能有一个且只能放在最后。
- `case` 标签后面的值必须是常量表达式（通常是字面值），不能是含有变量的非常量表达式。
- `switch` 语句先计算 表达式 的值，然后用此值自上而下比较 `case` 中的常数值，直至找到一个相等的值，然后执行匹配的语句块。
- `default:` 标签的语句块会在上述所有的 case 标签都没有匹配成功的情况下执行。
- 如果 `case` 对应的语句块内没有用 break 语句终止执行 switch，则会向下执行所有的语句块（包括 `default:` 标签后面的语句块），直至结束。

示例：

请输入一年中的季度(1/2/3/4)，打印这个季度对应的名称。如果输入其它值，则提示：`您的输入有误！`。

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int season;

    printf("请输入一年中的季度(1/2/3/4): ");
    scanf("%d", &season);
    switch (season) {
        case 1:
            printf("春季\n");
        case 2:
            printf("夏季\n");
        case 3:
            printf("秋季\n");
        case 4:
            printf("冬季\n");
        default:
            printf("您的输入有误！\n");
    }

    return 0;
}
```

执行结果：

```
weimingze@mzstudio:~$ ./a.out
请输入一年中的季度(1/2/3/4): 100
```

```
您的输入有误!
weimingze@mzstudio:~$ ./a.out
请输入一年中的季度(1/2/3/4): 4
冬季
您的输入有误!
weimingze@mzstudio:~$ ./a.out
请输入一年中的季度(1/2/3/4): 1
春季
夏季
秋季
冬季
您的输入有误!
```

从上述执行结果中可以看出，当输入 100 时，case 没有匹配则值，则直接执行 `default:` 后面的语句。当输入 1 时，则匹配 `case 1:` 标签，然后执行后面的语句，接下来执行`case 2:` 标签后面的语句，…… ，最后执行 `default:` 标签后面的语句。这种执行方式叫做**穿透**。如何能够避免穿透，使逻辑的正常呢？下面我们来说下 **break 语句**。

实验：

完成上述示例的编写和运行，仔细查看运行结果是否与讲述内容相符。

5. break 语句

break 语句用于 switch 语句或迭代语句中，用于终止包含它的 `switch`、`for`、`while`、`do-while`语句的执行。

break 语句是跳转语句（后面会讲）之一，它不是选择语句。

语法

```
break;
```

> 语法中：`break` 是关键字。

语法说明:

- break 只能用在 `switch`、`for`、`while`、`do-while`语句的内部。
- 当 `break` 语句执行后，则包含它的上述语句中会立即终止执行，转去执行后面的语句。
- break 语句只能终止包含它的上述四条语句中的一个，当有上述语句嵌套时，只能终止离它最近的一个语句。

改写 switch 语句中的示例如下：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int season;

    printf("请输入一年中的季度(1/2/3/4): ");
    scanf("%d", &season);
    switch (season) {
        case 1:
            printf("春季\n");
            break;
        case 2:
            printf("夏季\n");
            break;
        case 3:
            printf("秋季\n");
            break;
        case 4:
            printf("冬季\n");
            break;
        default:
            printf("您的输入有误! \n");
    }

    return 0;
}
```

执行结果如下：

```
weimingze@mzstudio:~$ ./a.out
请输入一年中的季度(1/2/3/4): 100
您的输入有误!
weimingze@mzstudio:~$ ./a.out
请输入一年中的季度(1/2/3/4): 4
冬季
weimingze@mzstudio:~$ ./a.out
请输入一年中的季度(1/2/3/4): 1
春季
```

练习

写一个程序，输入成绩的级别(`A`、`B`、`C`、`D`、`E`):A 级（90-100分)，B级（80-89分)，C级（70-79分）， D级（60-69分)，E 级（0-59分)。要求：输入级别，打印分数值范围。

第七章、迭代语句

迭代的概念：

迭代（Iteration）就是通过不断重复和改进来逐步接近或达成目标的过程。迭代通常每次重复都会根据前一次的反馈进行调整优化，最终实现目标。

迭代语句（Iteration statements）也叫循环语句。迭代语句用于将一条语句（可以是一条复合语句）在满足特定条件时重复执行多次，避免代码重复来实现复杂的功能。

C 语言中的迭代语句有 3 种：

1. for 语句
2. while 语句
3. do-while语句

迭代语句的语法如下：

```
while ( expression ) statement
do statement while ( expression ) ;
for ( expression ; expression ; expression ) statement
for ( declaration expression ; expression ) statement
```

以下会详细讲解上述语法。

1. for 语句

for 语句通常用于控制一段代码（一条语句或复合语句）的执行次数，直到满足某个条件为止的循环。

for 语句的语法有两种

1、没有变量声明的 for 语句

```
for (表达式1; 表达式2; 表达式3)
    语句
```

2、带有变量声明的 for 语句

```
for (变量声明1 表达式2; 表达式3)
    语句
```

> 语法中：`for` 是关键字。

for 语句的执行过程：

1. 先执行 **表达式1** 或 **变量声明1** 此部分通常对循环中需要使用的变量进行创建（仅适用于带有变量声明的语法2）和初始化。
2. 计算 **表达式2**，根据计算结果值来决定是否执行**语句**：
    1. 当**表达式2**的计算结果为非零值时，则执行**语句**。
    2. 当**表达式2**的计算结果为零值时（整数:`0`、浮点数：`0.0`、字符：`'\0'`或空指针:`(void*)0`），此 for 语句执行结束。
3. 当 **语句** 部分正常执行完毕后则执行 **表达式3**，然后继续回到第二步计算 **表达式2**。

总结：

1. **表达式1** 或 **变量声明1** 通常用来初始化循环变量。
2. **表达式2** 通常用来判断是否继续循环。
3. **表达式3** 通常用来更新循环变量。
4. **语句** 是循环的执行体。

说明：

1. **表达式1** 或 **变量声明1** 会在 for 语句运行时最先运行，且只运行一次。
2. **变量声明1** 中可以声明变量，此处声明的变量的生命周期属于此 for 语句内，不能在此 for 语句之外使用。
3. **变量声明1** 必须以分号结束，且只能有一个分号作为结束，如果有多个变量声明则使用逗号（`,`）分隔。
4. **表达式1** 可以为空。
5. **变量声明1** 可以为空，为空时需要补充一个分号（`;`）。
6. **表达式2** 可以为空。为空时表示此表达式则只为真值（`1`），等同于添加一个 `1` 作为表达式2。
7. **表达式3** 可以为空。
8. **语句** 不能为空，如果不需要执行任何语句，需要添加空语句 `;` 或 空复合语句 `{}`。

示例

打印 1 到 10 十个整数。

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int number;  // 用于控制循环次数的变量

    for (number = 1; number <= 10; number ++) {
        printf("%d\n", number);
    }
    printf("程序退出前，number:%d\n", number); // 此时 number 的值时多少?
    return 0;
}
```

也可以写成

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int number = 1; number <= 10; number ++) {
        printf("%d\n", number);
    }
    // printf("程序退出前，number:%d\n", number); // 此处打印number 会报错。
    return 0;
}
```

练习1:

打印 1 - 101 之间所有的偶数。

练习2:

打印 1 - 100 之间所有的奇数的和。

2. while 语句

while 语句用于根据某个条件控制一段代码（一条语句或复合语句）重复执行。当循环条件不满足或在执行语句中执行了 break 语句才会终止循环。

语法：

```
while (表达式)
    语句
```

> 语法中：`while` 是关键字。

while 语句的执行过程：

1. 计算 **表达式**，根据计算结果值来决定是否执行**语句**：
    1. 当**表达式**的计算结果为非零值时，则执行**语句**。
    2. 当**表达式**的计算结果为零值时（整数:`0`、浮点数：`0.0`、字符：`'\0'`或空指针:`(void*)0`），此 while 语句执行结束。
2. 当 **语句** 部分正常执行完毕后则回到上一步再次执行 **表达式** 对循环条件进行判断。

示例

写程序，使用 while 语句打印 5 行 `hello world`

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int times = 1;
    while (times <= 5) {
        printf("hello world\n");
        times += 1;
    }
    printf("while 语句结束: times=%d\n", times);
}
```

练习：

写一个程序，输入一个整数n，写程序打印如下 n 行文字。

如:

```
请输入: 10
这是第 1 行
这是第 2 行
这是第 3 行
...
这是第 10 行
```

3. do-while 语句

do-while 语句同样是根据某个条件控制一段代码（一条语句或复合语句）重复执行。但与 while 不同之处在于 do-while 内部的语句是先执行再判断。即先执行语句部分，然后再判断循环条件来决定是否再执行下一次。也就是说 do-while语句内的语句至少要执行一次。

语法：


```
do {
    语句
} while (表达式);
```

> 语法中：`do`、`while`是关键字。
>
> 注意：语法中的最后一定要有一个英文的分号（`;`）结束。

do-while 语句的执行过程：

1. 执行 **语句** 部分。
2. 计算 **表达式**，根据计算结果值来决定是否再次执行**语句**：
    1. 当**表达式**的计算结果为非零值时，则回到第一步，再次执行**语句**。
    2. 当**表达式**的计算结果为零值时（整数:`0`、浮点数：`0.0`、字符：`'\0'`或空指针:`(void*)0`），此 do-while 语句执行结束。

示例

写一个程序：输入六位数字密码，如果密码不是`123456`则继续输入。否则提示`登录成功`，退出程序。

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int pwd;

    do {
        printf("请输入密码: ");
        scanf("%d", &pwd);
    } while( pwd != 123456);

    printf("登陆成功!");

    return 0;
}
```

练习

- 写一个程序：输入六位数字密码，如果密码不是`123456`则继续输入，最多重复输入 3 次。如果超过 3 次则提示`登陆失败`，否则提示`登录成功`，退出程序。

4. 死循环

死循环（death-loop） 是指循环条件一直成立的循环。在 C 语言中可以使用 `for` 语句、`while` 语句和 `do-while` 语句实现死循环。

死循环不是一个语法，它只是 C 语言中的一种特殊用法。死循环通常用于循环次数无法确定的循环。

> C 语言中也可以使用 goto 跳转语句（后面会讲）实现死循环（不推荐）。

死循环的写法:

写法1

```
for (;;) {
    // 循环体部分
}
```

写法2

```
while (1) {
    // 循环体部分
}
```

写法3

```
do {
    // 循环体部分
} while(1);
```

说明:

- 死循环通常使用 break 语句来终止循环;
- 在终端中，可以使用 **Control + c** 快捷键来终止正在执行的程序。

示例

写一个程序，输入任意个学生的成绩，当输入负数时结束输入。打印这些学生的总成绩。

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int total_score = 0;
    int score = 0;
    while (1) {
        printf("请输入成绩: ");
        scanf("%d", &score);
        if (score < 0)
            break;
        total_score += score;
    }
    printf("总成绩是:%d\n", total_score);
}
```

练习：

- 写一个程序，输入多个人的年龄，当输入的年龄小于等于 0 时结束输入。计算这些人的平均年龄并打印结果。

5. 语句嵌套

在 C 语言中所有的 **迭代语句** 和 **选择语句** 本身也是语句，它可以放入到其它的 **迭代语句** 和 **选择语句** 中形成一种嵌套结构。用这种嵌套的结构可以实现复杂的功能。

一、for 语句中嵌套 if 语句可以这样写：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i = 1; i <= 20; i++) {
        printf("%2d ", i);
        if (i % 5 == 0)
            printf("\n");
    }
    return 0;
}
```

二、if 语句中嵌套 for 语句可以这样写：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x;
    printf("please input a number:");
    scanf("%d", &x);
    if (x > 20)
        for (int i = 1; i <= x; i++) {
```

```
            printf("%2d ", i);
            if (i % 5 == 0)
                printf("\n");
        }
    return 0;
}
```

三、for 语句中嵌套 for 语句可以这样写：

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int y = 0; y < 4; y++) {
        for (int x = 1; x <= 5; x++) {
            printf("%2d ", y * 5 + x);
        }
        printf("\n");
    }
    return 0;
}
```

同理，while 语句、do-while 语句、for 语句都可以任意嵌套，但嵌套一定要复合业务逻辑，不要为了嵌套而嵌套，不要玩语法糖。

编写高质量程序的原则是越简单越好。有时候编写复杂的程序是为了逻辑更清晰或执行效率高。而不是为了炫技。

练习：

编写一个程序，打印小学我们学过的 9 行 9 列的九九乘法表。打印结果如下：

```
1x1=1
1x2=2 2x2=4
1x3=3 2x3=6 3x3=9
...
1x9=9 2x9=18 3x9=27 ... 9x9=81
```

第八章、跳转语句

跳转语句（Jump statements）可以在程序执行过程中打破原有自上而下以语句为单位依次执行的顺序，无条件的跳转到指定的位置。

C 语言为我们提供了四种跳转语句，它们分别是：

- break 语句
- continue 语句
- goto 语句
- return 语句

跳转语句的语法

```
break;
continue;
goto 标识符(标签);
return [表达式];
```

1. break 语句

前面我们已经介绍 break 语句在 switch 语句中的用法。
如果你还需要了解之前 break 语句的内容，请点击这里！

break 语句用于 switch 语句或迭代语句中，用于终止包含它的 `switch`、`for`、`while`、`do-while`语句的执行。

语法

```
break;
```

语法说明:

- break 只能用在 `switch`、`for`、`while`、`do-while`语句的内部。
- 当 `break` 语句执行后，则包含它的上述语句会立即终止执行，转去执行后面的语句。
- break 语句只能终止包含它的上述四条语句中的一个，当有上述语句嵌套时，只能终止离它最近的一个语句。

以下我们来用示意性的代码来说明 break 语句在各个语句中的跳转位置。

一、break 在 switch 语句中执行后的跳转位置。

```
...
switch (/* ... */) {
    case /* ... */:
        /* ... */
        break;
        /* ... */
    case /* ... */:
        /* ... */
        break;
        /* ... */
}
// <--- break 跳转到此处
```

二、break 在 for 语句中执行后的跳转位置。

```
for (/* ... */) {
    /* ... */
    break;
    /* ... */
}
// <--- break 跳转到此处
```

三、break 在 while 语句中执行后的跳转位置。

```
...
while (/* ... */) {
    /* ... */
    break;
    /* ... */
}
// <--- break 跳转到此处
```

四、break 在 do-while 语句中执行后的跳转位置。

```
do {
    /* ... */
    break;
    /* ... */
} while(/* ... */);
// <--- break 跳转到此处
```

五、break 在 for 语句嵌套中执行后的跳转位置。

```
for (/* ... */) {
    for (/* ... */) {
        /* ... */
        break;
        /* ... */
    }
    // <--- break 跳转到此处（只结束包含它的最近的 for 语句）
}
// 注意不是跳转到这里。
```

练习：

写程序，任意输入一些正整数，当输入负数时结束输入。当输入完成后，打印您输入的这些正整数的和。如:

```
请输入: 1
请输入: 2
请输入: 3
请输入: 4
请输入: -1
您刚才输入的正整数之和是: 10
```

2. continue 语句

continue 语句用于迭代语句(for 语句、while 语句和 do-while 语句)中，当continue 语句执行后则不再执行本次循环内 continue 之后的语句，重新开始一次新的循环。

语法

```
continue;
```

> 语法中：`continue` 是关键字。

语法说明:

- continue 只能用在 `for`、`while`、`do-while`语句的内部。
- 在迭代语句中，当 continue 语句执行后，continue 之后的语句将不再执行。程序跳转到指定位置后继续执行。
- 在 for 语句中，当 continue 语句执行后，程序跳转 for 语句括号内的**表达式3**处执行。然后计算**表达式2**判断循环条件再决定是否再次循环。

- 在 while 语句中，当 continue 语句执行后，程序跳转到 **表达式** 处。待计算**表达式**结果后再根据结果决定是否再次循环。
- 在 do-while 语句中，执行 continue 语句后，程序跳转到 **表达式** 处。待计算**表达式**结果后再根据结果决定是否再次循环。

以下我们来用示意性的代码来说明 continue 语句在各个语句中的跳转位置。

一、continue 在 for 语句中执行后的跳转位置。

```
for (/* ... */) {
    /* ... */
    continue;
    /* ... */
    // <--- continue 跳转到此处
}
```

二、continue 在 while 语句中执行后的跳转位置。

```
...
while (/* ... */ )
{
    /* ... */
    continue;
    /* ... */
    // <--- continue 跳转到此处
}
```

三、continue 在 do-while 语句中执行后的跳转位置。

```
...
do {
    /* ... */
    continue;
    /* ... */
    // <--- continue 跳转到此处
} while(/* ... */);
```

示例：

编写一个程序，输出 1 到 20 之间的所有数字，但跳过所有能被 3 和 7 整除的数字。

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i = 1; i <= 20; i++) {
        if (i % 7 == 0)
            continue;
        if (i % 3 == 0)
```

```
            continue;
        printf("%d\n", i);
    }
    return 0;
}
```

练习：

写一个程序，打印 1~100 范围内个位、十位、百位都不包含 7，且不能被 7 整除的所有整数。

3. goto 语句

goto 语句用于无条件跳转到同一函数内指定**标签语句（Labeled statements）**的位置开始执行。

语法

此语法用于跳转到指定标签处的语句。

```
goto 标识符（标签）;
```

标签语句的语法:

此语法会在语句处打一个标签，此标签会通过编译器编译为此语句的绝对位置。

```
标签： 语句
```

说明：

1. **标签** 必须是标识符。
2. 标签定义可以是本函数内的任意位置，可以不遵守先定义后使用的规则，且标签只能是函数内有效。
3. goto 语句不能跳转到函数之外。
4. goto语句不得从具有可变修改类型标识符的作用域外部跳转至该标识符的作用域内部。

示例:

使用 goto 语句实现迭代语句的功能。

```
#include <stdio.h>

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



```
    int i = 0;

mylabel:
    printf("i: %d\n", i);
    i += 1;
    if (i < 5)
        goto mylabel;

    printf("程序结束\n");
    return 0;
}
```

注意：

goto语句不得从具有可变修改类型标识符的作用域外部跳转至该标识符的作用域内部。这可能会导致变量创建失败或没有初始化，是及其危险的行为。为了减少类似的错误发生，C 语言建议使用迭代语句和其它跳转语句组合使用来代替 goto 语句。尽可能少的使用 goto 语句可以避免很多难以控制的错误。

如，错误示例如下：

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i = 0;

    goto myprint;
    int x = 200; // 错误，跳过了此变量的初始化。

myprint:
    printf("i: %d, x: %d\n", i, x);
    return 0;
}
```

练习：

- 尝试在两重嵌套的 for 语句中，在最内层的 for 语句使用 goto 语句直接结束最外层 for 语句的执行而结束这两重 for 循环。

4. return 语句

return 语句 的作用是用来结束当前函数的执行，返回到调用此函数的地方。同时它可以返回一个值（表达式的结果）作为调用此函数的返回值。

语法:


```
return [表达式];
```

说明：

1. return 语句后跟表达式计算结果的类型要和函数返回值的类型一致，如果不一致则尝试使用隐式类型转换，如果不能转换则会报错。
2. 如果函数的返回类型为 void 类型，则 return 语句不需要此**表达式**。
3. 在 UNIX/Linux 操作系统中，main 函数的返回值的最低 8 位会返回给 Shell 作为程序执行的结果，在 Shell 中可以使用特殊变量 `$?` 获取该值。

C 语言函数的常用的约定：

- 函数返回 `0` 值表示成功
- 函数返回非 `0` 值 表示失败。

示例：

```c
// filename: test_return.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("1\n");
    return 10;  // 此return 语句将结束 main 函数的执行。
    printf("2\n");  // 此语句及以下语句不会执行。
    return 0;
}
```

上述程序在 Linux 操作系统下的运行结果：

```
weimingze@mzstudio:~$ gcc -o test_return test_return.c
weimingze@mzstudio:~$ ./test_return
1
weimingze@mzstudio:~$ echo $?  # 打印 test_return 的运行结果
10
weimingze@mzstudio:~$ ./test_return || echo "OK"
1
OK
```

> 上述运行结果显示 `./test_return` 的返回结果为假，`echo "OK"` 命令才会运行

练习：

- 练习使用 return 语句从主函数的任意位置返回，尝试返回 **零值** 、 **非零值**以及大于 `255` 的值，尝试在 UNIX/Linux 终端下用 `$?` 打印程序运行的结果。

第九章、表达式语句和空语句

前面在 **运算符与表达式** 一章已经介绍过表达式的概念。这一章我们在继续讲解一下表达式语句。

1. 表达式语句

表达式语句（Expression statements）是由一个表达式后跟一个英文的分号（`;`）构成。

语法:

```
表达式;
```

说明

- 表达式可以是一个字面值，如: `0`。
- 表达式可以是一个变量，如: `x`。
- 表达式可以是一个计算式，如: `1+2`。
- 表达式可以是一个赋值表达式，如: `x = 100`。
- 表达式可以是一个函数调用，如: `printf("hello")`。

> 注意：变量声明不是表达式语句，如：`int x = 100;`、`float pi;` 都不是表达式语句。

示例:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 1, y = 2, z;  // 这是变量声明，不是表达式语句

    z = x + y;  // 是表达式语句
    printf("z:%d\n", x);  // 是表达式语句

    return 0;   // 这是 return 语句，不是表达式语句
}
```

2. 空语句

空语句(Null statements) 是指表达式为空的语句。正确的表达式语句的语法格式是 表达式后跟一个英文的分号（`;`），当表达式语句中的表达式省略掉，只剩下一个英文的分号（`;`）时，即为空语句。

空语句的主要作用是**填充语法空白**。

空语句的语法

```
;
```

示例：

写一个程序，如果成为在 0 ~ 100 的范围内，则正常打印成绩，如果成绩不在 0 ~ 100 范围内，则提示您的输入有误。

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int score;
    printf("请输入成绩(0~100):");
    scanf("%d", &score);
    if (score >= 0 && score <= 100)
        printf("您的成绩是:%d\n", score);
    else
        printf("成绩有错!\n");
    return 0;
}
```

上述程序已经完成，但现在需要按需求做如下的修改。

如果成为在 0 ~ 100 的范围内则什么都不做，如果成绩不在 0 ~ 100 范围内，则提示您的输入有误。

基于上面的需求，我删除 `if` 后面的语句 `printf("您的成绩是:%d\n", score);` 修改后的代码如下：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int score;
    printf("请输入成绩(0~100):");
    scanf("%d", &score);
    if (score >= 0 && score <= 100)  // <--  此处会报语法错误
```

```
    else
        printf("成绩有错!\n");
    return 0;
}
```

上述程序在编译时会出错。原因是 `if (score >= 0 && score <= 100)` 后面一定要跟着一条语句。虽然我们不需要执行任何语句，但也必须要填充语句才能符合语法规则。那我们可以填充一个空语句。修改如下：

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int score;
    printf("请输入成绩(0~100):");
    scanf("%d", &score);
    if (score >= 0 && score <= 100)
        ;  // <-- 此处添加了一个空语句，语法和逻辑都正确了。
    else
        printf("成绩有错!\n");
    return 0;
}
```

实验：

- 修改上述示例程序。尝试在上述程序中 `if (score >= 0 && score <= 100)` 后添加两个空语句是否会出错，为什么？

第十章、其它语句

1. 复合语句

复合语句（Compound statements） 也常被称为代码块，是用一对大括号 `{}` 将 **零条** 或 **多条语句** 括起来所形成的一个整体。

复合语句的语法:

```
{
    语句块列表
}
```

语句块的语法

```
声明
语句
```

语法说明：

- 声明可以是变量声明，也可以是函数声明。声明部分可以省略不写。
- 语句部分可以有一条或多条语句，也可以省略不写。
- C89/C90 的C语言标准规定复合语句的语句块列表内只有一条语句块。即只能在最上面有一个声明。不能在任意位置声明变量。
- C99、C11的C语言标准规定复合语句内可以有多条语句块。即可以在任意位置声明变量。
- 在语法上，C 语言将复合语句视为一条单独的语句。

示例

输入语文成绩和数学成绩，计算并打印两科成绩的和。

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int total_score = 0;
    {  // 这对大括号是复合语句
        int score;
        printf("请输入语文成绩(0~100):");
        scanf("%d", &score);
```

```
        if (score < 0 || score > 100) {  // 这对大括号是复合语句
            printf("语文成绩有错\n");
            return 1;
        }
        total_score += score;
    }
    {  // 这对大括号是复合语句
        printf("请输入数学成绩(0~100):");
        int score;  // C99 及之后版本的编译器才可以在语句后声明变量。
        scanf("%d", &score);
        if (score < 0 || score > 100) {  // 这对大括号是复合语句
            printf("数学成绩有错\n");
            return 2;
        }
        total_score += score;
    }
    printf("您的总成绩是: %d\n", total_score);
    return 0;
}
```

注意：

一个空的**复合语句** `{}` 等同于一条由单个的分号(`;`)组成的**空语句**。如下面的示例中。两种写法效果相同。

```
int x;
for (x = 0; x < 100; x++) {}  // {} 是空复合语句
```

也可以写成

```
int x;
for (x = 0; x < 100; x++) ;  // ; 是空语句
```

实验

- 随便写一个程序，自己定义一个复合语句，在复合语句内部声明变量，在复合语句后面打印这个变量的值，看是否会报错。

2. 标签语句

标签语句（Labeled statements） 仅用于 goto 语句 或 switch 语句中。标签语句的特点是在执行的语句前都有一个英文冒号（`:`）结尾的标签，冒号前面就是标签。

标签语句的作用是用于程序中语句位置的定位。

标签语句的语法有三种

```
identifier : statement
case constant-expression : statement
default : statement
```

中文表示:

```
标识符 : 语句
case 常量表达式 : 语句
default : 语句
```

说明:

1. 其中 `case` 或 `default` 标签仅能出现在 switch 语句中。
2. **常量表达式**是指在编译阶段就能确定结果的表达式。此表达式的值在运行时不可修改。
3. 标签的名字必须是**标识符**，且此**标识符**在当前函数中必须唯一。
4. `identifier : statement` 的语法格式通常用于为 goto 语句定义跳转标签。

示例:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int season;

    printf("请输入一年中的季度(1/2/3/4): ");
    scanf("%d", &season);
    switch (season) {
        case 1:  // <--- 此处是标签语句
            printf("春季\n"); goto exit;
        case 2:  // <--- 此处是标签语句
            printf("夏季\n"); goto exit;
        case 3:  // <--- 此处是标签语句
            printf("秋季\n"); goto exit;
        case 4:  // <--- 此处是标签语句
            printf("冬季\n"); goto exit;
        default:  // <--- 此处是标签语句
            printf("您的输入有误! \n");
    }
    printf("判断结束\n"); // 正确输入时不会执行此语句;
exit:  // <--- 此处是标签语句
    printf("程序退出\n");

    return 0;
}
```

第十一章、指针

这一章我们来学习 C 语言中最难理解的知识点**指针**。

指针是 C 语言的灵魂。C 语言有了指针才使得它变得更加强大、灵活和高效。如果你不懂 C 语言的指针就等于你没有掌握 C 语言。如果你不打算搞懂指针。那建议你学习其它的编程语言，比如 python。

指针 是 C 语言的重点难点，它理解起来比较令人困惑。要理解指针我们先要了解内存以及变量在内存中的存放方式。

下面我先来讲解一下内存和变量地址。

1. 内存和变量地址

先来说一下内存。内存是用来存放计算机运行时数据的重要存储单元。 我们声明的变量在运行时都占用一段连续的内存空间。

在计算机内，内存是以字节为最小单位的存储单元组成的。这些字节在逻辑上是顺序排列的，每个字节都有自己的地址。第一个字节的地址为 0，第二个字节的地址为1，以此类推。一般要存储比较小的数字，比如年岭我们用一个字节就够了。但我们要存储你的豪车的里程数，显然我们需要两个或四个字节才能存得下。因此我们存储不同的数据就有了不同的数据类型。也就是说，数据类型是内存结构中的数据对外（编译器）的具体表现形式。

在现实中，你可以把内存理解成依山傍海的海边的一条狭长的平原空地。你打算在这里建造一座城市。于是作为城市规划师的你自西向东每隔 10 米划分成一个地段，如下图所示：

```
^^ ^^ ^^ 山 ^^ ^^ ^^ 山 ^^ ^^ ^^ https://weimingze.com ^^ ^^ ^^ 山 ^^ ^^ ^^
 +-------+-------+-------+-------+-------+-------+-------+-------+-------+
 | house | house | house |            school             | house |   ...
 +-------+-------+-------+-------+-------+-------+-------+-------+-------+
     1       2       3       4       5       6       7       8       9
                           道             路
~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~
 ~~ ~~ ~~ 海 ~~ ~~ ~~ ~~ ~~ ~~ 海 ~~ ~~ ~~ ~~ ~~ ~~ 海 ~~ ~~ ~~ ~~ ~~ ~~ ~
~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~
```

你为每个地块都定义的一个数字 1、2、3 这就是地址。其中 居民房子（house）比较小占一个地块，学校（school）比较大占四个地块。房子和学校就是不同的类型（对应C 中的数据类型）。但有个问题出现了，学校占用了4、5、6、7 四个地块，为了邮寄和管理方便，我们如何确定学校

的地址呢？聪明的你想到了用最小的数值表示跨越多个地块的建筑的地址（C 语言中也是这样规定地址的），即学校的地址是 4 而不是 5、6、7。

在 C 语言中，我们声明的变量，在运行时都会存在于内存中。我们用 `&标识符` 的语法就可以获取一个变量的地址。下面我们用程序打印这些变量的地址。

```c
// filename: memory_addr.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char ch1 = 'a', ch2 = 'b';
    short s = 10;
    int i = 1000;

    printf("&ch1:%p\n", &ch1);
    printf("&ch2:%p\n", &ch2);
    printf("&s  :%p\n", &s);
    printf("&i  :%p\n", &i);

    return 0;
}
```

说明：

- 格式化字符串中的 `%p` 是以十六进制的格式输出内存地址。

运行结果：

```
weimingze@mzstudio:~$  gcc -o memory_addr memory_addr.c
weimingze@mzstudio:~$  ./memory_addr
&ch1:0x7ffdd8ac3ae0
&ch2:0x7ffdd8ac3ae1
&s  :0x7ffdd8ac3ae2
&i  :0x7ffdd8ac3ae4
```

根据上述输出结果，可以推断其内存分布结构如下：

```
          ch1     ch2          s                       i
   ----+-------+-------+-------+-------+-------+-------+-------+-------+----
   ... |  'a'  |  'b'  |      10       |              1000             | ...
   ----+-------+-------+-------+-------+-------+-------+-------+-------+----
       ^       ^       ^               ^                               ^
       | &ch   | &ch2  | &s            | &i                            |
0x7ffdd8ac3ae0 |   0x7ffdd8ac3ae2      |                         0x7ffdd8ac3ae8
         0x7ffdd8ac3ae1           0x7ffdd8ac3ae4
```

计算内的内存地址的范围

- 一般在 32 位的操作系统中，内存地址是 `0x00000000 ~ 0xFFFFFFFF` 的 32 位的无符号整数数值。
- 一般在 64 位的操作系统中，内存地址是 `0x0000000000000000 ~ 0xFFFFFFFFFFFFFFFF` 的 64 位的无符号整数数值。

> 具体内存的取值范围由操作系统的位宽和编译器共同决定。

现在我们了解了地址的概念。那么我们如何来用变量来保存这些地址呢？这就需要用到我们下一节课讲到的**指针**。

实验：

写一段程序，定义几个变量，打印这些变量的地址，分析这些变量在内存中的分布情况。

2. 指针的声明

指针（Pointer）是用来保存另外一个变量地址的变量，其值是另一个变量的内存地址。

指针的作用：

1. 通过指针可以访问和修改指针指向的内存地址的数据。
2. 指针可以进行运算，可以通过运算改变指向的地址。

指针变量声明的语法

```
变量类型 *指针变量名称1 [= 初始值1] [, *指针变量名称2 [= 初始值2]][, ...];
```

语法说明：

1. 中括号(`[]`) 括起来的内容是可选的内容，可以省略不写。
2. **变量类型** 不是指针的类型，是此指针可以指向的变量的类型。
3. 指针的类型是 **变量类型** 后根一个 星号（`*`）， 即：**变量类型*** 类型。
4. 一个变量的类型如果最右侧有一个星号（`*`），则这个变量一定是指针类型的变量。
5. 第二个或之后的 **变量名称** 如果声明为指针，则前面必须加一个 `*`，否则它的类型是 **变量类型**， 不是 **变量类型*** 类型。

指针说明

1. 指针的值实质是一个整数，这个整数是指向变量的内存地址。
2. 在同一个程序中，无论什么类型的指针变量，它的内存长度都是一样的（4 字节或 8 字节）。
   - 一般32位编译器和32位操作系统下指针变量的占用内存为 4 字节。
   - 一般64位编译器和64位操作系统下指针变量的占用内存为 8 字节。
   - 其它情况是 4 字节或 8 字节依具体编译器和操作系统来确定。
3. 指针变量如果在声明时没有赋初始值，则指针的指向是不确定的。我们把这种指针通常称之为野指针。

示例:

```c
// file: pointer_declaration.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 1, y = 2;  // x, y都是普通的变量，类型为: int
    int *p1 = &x, *p2, *p3; // p1、p2、p3是指针，类型为: int*。

    p2 = &y;  // p2 指向 y, 即 p2 的值为 y 的地址

    // 打印 x、y 的地址
    printf("&x:%p, &y:%p\n", &x, &y);
    // 打印 p1、p2、p3的值
    printf("p1:%p, p2:%p, p3:%p\n", p1, p2, p3);

    // 打印 p1 指针占用内存的字节数
    printf("sizeof(p1):%ld\n", sizeof(p1));

    return 0;
}
```

说明:

- 上述程序中指针变量 p1 是在初始化时给定了初始值是 x 变量的地址，即`&x`。
- p2 开始没有给定地址，后来通过赋值表达式将 y 的地址赋值给 `p2`。
- p3 在声明时没有进行初始化，则 p3 的值可能是任意值（不确定的值），即**野指针**。

运行结果如下:

```
weimingze@mzstudio:~$  gcc -o pointer_declaration pointer_declaration.c
weimingze@mzstudio:~$  ./pointer_declaration
&x:0x7ffe32b3fba8, &y:0x7ffe32b3fbac
p1:0x7ffe32b3fba8, p2:0x7ffe32b3fbac, p3:0x7ffe32b3fcb0
sizeof(p1):8
```

运行结果说明

1. x 的地址 `&x` 和 p1 的值 都是 `0x7ffe32b3fba8`，说明 p1 的值就是 &x。
2. y 的地址 `&y` 和 p2 的值 都是 `0x7ffe32b3fbac`，说明 p2 的值就是 &y。
3. 上述结果 p3 可能是任意值，只是这里正好是 0 而已，请不要对未初始化的指针进行使用。
4. p1 占用的内存数是 8 字节（这个值和操作系统有关）。

上述程序的内存结构如下：

```
     x      y               p1             p2             p3
  ------+------+------+--------------+--------------+--------------+----
    ... |   1  |   2  |0x7ffee40ae95c|0x7ffee40ae958|      0x0     |
  ------+------+------+--------------+--------------+--------------+----
        ^      ^      ^              ^              ^              ^
        |      |      |              |              |              |
0x7ffe32b3fba8 |  0x7ffe32b3fbb0     |       0x7ffe32b3fbc0        |
        0x7ffe32b3fbac        0x7ffe32b3fbb8              0x7ffe32b3fbc8
```

实验：

- 声明一个 `char` 类型的指针 `pch1`，打印此指针变量占用内存的字节数。
- 声明一个`short int`类型的指针 `psi1`，打印此指针变量占用内存的字节数。
- 声明一个`long int`类型的指针 `pli1`，打印此指针变量占用内存的字节数。

3. 指针解引用

上节课我们讲解了如何来声明一个指针变量。这节课我们来讲解如何通过 **指针解引用** 来操作指针指向的变量。

引用（Reference） 原本是在 C++ 中才有的数据类型，在 C++ 中，一个引用相当于一个变量的别名，用引用可以操作实际关联的变量，从而实现对原有关联变量的访问和修改。

指针解引用（Dereference） 是通过星号（`*`）运算符后跟一个指针（指针变量或指针常量）的运算。通过 **指针解引用** 运算，你可以对指针指向的内存内的数据进行取值或赋值等操作。

在 C 语言中 通过 **指针解引用** ，可实现对指针指向的变量的引用操作。即通过 **指针解引用** 可以操作指针指向的变量，即指向变量的**引用**操作。

语法:


```
* 指针
```

说明

1. **指针解引用** 返回的数据是指针指向的变量的引用。
2. **指针解引用** 后能够引用的内存数据范围和指针本身的类型相关。
3. **指针解引用** 后能够引用的数据类型是指针类型去掉一个星号（`*`）后的类型，如：指针 `p` 的类型为 `char*`，则解引用后 `*p` 的类型为 `char`。

示例：

```c
// file: pionter_dereference.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 1;
    char ch = 'A';
    int *p1 = &x;
    char * p2 = &ch;

    // 打印x,和 *p1 值
    printf("x:%d, *p1:%d\n", x, *p1);
    // 通过指针 p1 修改 x 的值
    *p1 = 100;
    printf("x:%d, *p1:%d\n", x, *p1);

    // 修改 ch 的值
    ch = 'B';
    printf("ch:%c, *p2:%c\n", ch, *p2);

    // 打印指针 p1 和 p2 占用的内存长度。
    printf("sizeof(p1):%ld, sizeof(p2):%ld\n", sizeof(p1), sizeof(p2));
    // 打印指针 p1 和 p2 解引用后的数据占用的内存长度。
    printf("sizeof(*p1):%ld, sizeof(*p2):%ld\n", sizeof(*p1), sizeof(*p2));

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$  gcc -o pionter_dereference pionter_dereference.c
weimingze@mzstudio:~$  ./pionter_dereference
x:1, *p1:1
x:100, *p1:100
ch:B, *p2:B
sizeof(p1):8, sizeof(p2):8
sizeof(*p1):4, sizeof(*p2):1
```

通过上述运行结果可以得出。

1. 要修改 x 变量的值有两种方法，
    1. 直接对变量 x 赋值。
    2. 使用 指向 x 变量的指针 p1 再对其解引用后赋值，即 `*p1 = 100` 也可以修改 x 变量。
2. 变量 x 的值变化，则指向此变量的指针解引用的值也会随之变化，因为p1解引用后 `*p1` 就是 x 变量。
3. 指针解引用后的实际是指向的变量，因此 `sizeof(*p1)` 的值是 4， `sizeof(*p2)` 的值是 1。

练习：

- 完成下面的程序，并达到预期的结果。

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 100, y = 200;
    int *px = &x, *py = &y;
    int temp;
    printf("x:%d, y:%d", x, y); // 打印: x:100, y:100
    // 使用指针交换 变量 x, y 的值(禁止对 x, y 变量进行赋值运算),
    // ...

    printf("x:%d, y:%d", x, y); // 打印: x:200, y:100
    return 0;
}
```

4. void 类型的指针

在 C 语言中，有这样一个用于描述数据类型的关键字 `void`，它不同于 `int`、`double` 等能够描述详细的内存结构，正好相反，它表示没有类型的类型（或者叫空类型）。

void 的字面含义是**空白的、空的**。

void 类型的三个作用：

1. 作为指针的类型，表示不关心指针指向地址的数据类型，仅用于保存内存地址。
2. 作为函数的返回类型（后面会讲）。
3. 作为函数的参数列表（后面会讲）。

有时候我们只需要一个指针来保存一个变量的内存地址，但是并不关心这段内存中存储的数据是什么类型，这个时候我们可以用 **void 类型的指针**，即：`void*` 类型。

void 类型的变量声明的语法

```
void *变量名1, *变量名2, ...
```

说明:

1. void 不能声明普通变量，只能用于声明指针类型的变量，如 `void x;` 的类型声明是错误的。
2. `void*` 类型的指针可以指向任何类型的指针，有时也叫通用指针。但要转到其它数据类型需要使用**类型转换运算符**进行强制类型转换。
3. `void*` 类型的指针通常只用来保存地址，不关心指向地址内的数据类型。
4. `void*` 类型的指针不能使用解引用运算符对内容进行访问。需要转化成其它类型的指针才能访问指向的数据。

示例:

```c
// filename: void_pointer.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char ch = 'a';
    short s = 10;
    int i = 1000;

    // 声明三个 void* 类型的指针，用于保存不同类型变量的地址。
    void *pv1 = &ch, *pv2 = &s, *pv3 = &i;

    //打印三个变量的地址
    printf("&ch:%p, &s:%p, &i:%p\n", &ch, &s, &i);
    //打印三个void*类型的指针的值
    printf("pv1:%p, pv2:%p, pv3:%p\n", pv1, pv2, pv3);
    // 打印 void 类型的指针的占用的内存空间的大小
    printf("sizeof(pv1):%ld\n", sizeof(pv1));

    // 使用强制类型转换将 pv3 的值转换为 int* 类型的指针，饭后解引用
    printf("int value at %p is %d\n", pv3, *((int*)pv3));

    return 0;
}
```

运行结果如下:

```
weimingze@mzstudio:~$ gcc -o void_pointer void_pointera.c
weimingze@mzstudio:~$ ./void_pointer
&ch:0x7ffdbf5c9189, &s:0x7ffdbf5c918a, &i:0x7ffdbf5c918c
pv1:0x7ffdbf5c9189, pv2:0x7ffdbf5c918a, pv3:0x7ffdbf5c918c
```

```
sizeof(pv1):8
int value at 0x7ffdbf5c918c is 1000
```

根据打印结果，可见

1. 各个 void* 类型的指针的值是各个变量的内存地址，与对应的变量的内存地址相同。
2. void* 类型的指针占用的字节数也是 8 字节（也可能是4字节），它和普通的指针的区别就在于不关心指向数据的类型。
3. 对 pv3 使用强制类型转换可以再次获取 i 变量的值。

实验

- 改写上述示例，使用 一个指针 `void * pv1` 来依次保存各个变量地址数据。并打印依次打印指针的值。

5. 空指针

空指针（Null pointer）是指内存指向 零地址 的指针，即指针的值为 `0`。

在 C 语言中，指针是非常灵活和高效的一种数据类型，它允许我们通过指针保存的地址来直接操作一段内存。但由于过分的灵活也给不熟悉或不能完全掌控指针的软件开发者带来不必要的困惑。因此为了避免不必要的麻烦。建议将没有使用的指针或已经用过但不再使用的指针的值设置为 **零值**，即**空指针**。

一般来说，计算机的内存零地址都不会存放任何的数据，如果一个指针指向零地址，在解引用对其修改时一般不会造成任何的破坏，甚至有些系统会通过异常的方式来阻止内存的访问，进而保证了程序和数据的安全。

将一个指针置空的写法1

```
int * p = 0;
```

上述写法中，由于 `p` 是 `int*` 类型， 而字面值 `0` 是 `int` 类型，因此有些编译器可能会报告警告或者错误。因此可以改写如下：

```
int * p = ((void*)0);
```

上述写法中，由于 `p` 是 `int*` 类型， `((void*)0)` 是将`0`转为 指针类型，这样写就不会报告任何警告和错误了。

在大多数系统中，为了简化上述写法，通常在头文件 `stddef.h` 中使用宏 `NULL` 来代替 `((void*)0)`，写起来更直观，可读写更好。

改写后声明空指针的示例如下：

```
// filename: null_pointer.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int *p = NULL;
    char * pch;
    pch = NULL;

    printf("p:%p, pch:%p\n", p, pch);

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o null_pointer null_pointer.c
weimingze@mzstudio:~$ ./null_pointer
p:(nil), pch:(nil)
```

> 注意： 在 gcc 编译器中会将 空指针显示为 `(nil)`，而非 `NULL`。

6. 野指针

野指针（Wild pointer）是指指针变量所指向的地址是不可用地址或指向非预想地址的指针。

野指针通常是在声明指针变量时没有初始化或指针指向的内存地址已经被释放不再可用而引起的。野指针为程序运行带来不确定性，通常会产生灾难性的后果。

为避免野指针的出现给程序造成破坏。建议在声明指针变量时初始化为空指针，在程序运行中指针指向的内容不再可用时也将指针置空。在使用指针时先判断是否指针为空指针，如果为空则放弃操作，不为空再进行操作。此种做法虽然会多了指针的判断而损失效率，但可以提高程序的安全性和可靠性。

示例:

```
// file: wild_pointer.c
#include <stdio.h>
```



```c
int main(int argc, char *argv[]) {
    int * pi1; // pi1 为野指针;
    int * pi2 = NULL;

    for (int i = 0; i < 10; i++) {
        pi2 = &i;
    }

    // 此时 pi2 也为野指针，因为 变量 i 已经不存在，pi2的值已不再有效。
    printf("pi1:%p, pi2: %p\n", pi1, pi2);

    return 0;
}
```

运行结果:

```
weimingze@mzstudio:~$ gcc -o wild_pointer wild_pointer.c
weimingze@mzstudio:~$ ./wild_pointer
pi1:0x7ffc1af453a0, pi2: 0x7ffc1af452a4
```

由运行结果可见指针 `pi1`和 `pi2` 不是空指针，而是不确定的值，因此对此指针解引用可能会出现不可预见的结果。

7. 指针的运算

前面我们已经学过了指针的解引用运算。这一小节我们学习指针的其它常用运算。

指针的运算有：

1. 指针解引用
2. 赋值运算
3. 算术运算
4. 关系运算
5. 取地址运算

7.1 指针的赋值运算

使用 赋值运算符（=）对一个指针变量赋值是改变指针的指向。赋值运算符是改变变量的值，但对于指针来讲，它的值就代表指针的指向，因此对指针赋值就是改变指针的指向。

语法：

```
指针 = 表达式
```

说明

1. 表达式的返回的类型最好和指针的类型相匹配，如果类型不匹配可能会引发隐式类型转换或编译器报警告（Warning）或错误（Error）。
2. 两个指针的值相同。说明两个指针指向的地址相同，如果两个指针的类型也相同，则这两个指针解引用操作是操作同一块内存。

示例：

```c
// filename: pointer_assignment.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 100;
    int *p1, *p2;
    p1 = &x;  // 将 x 的地址赋值给 p1，则 p1 指向 x 变量
    p2 = p1;  // 将 p1 的值赋值给 p2，p2 也指向 x 变量

    *p1 = 200;
    printf("x:%d, *p1:%d, *p2:%d\n", x, *p1, *p2);
    *p2 = 300;
    printf("x:%d, *p1:%d, *p2:%d\n", x, *p1, *p2);

    return 0;
}
```

运行结果

```
weimingze@mzstudio:~$ gcc -o pointer_assignment pointer_assignment.c
weimingze@mzstudio:~$ ./pointer_assignment
x:200, *p1:200, *p2:200
x:300, *p1:300, *p2:300
```

从运行结果可以看出。赋值语句将两个指针指向了同一个变量 x，用两个指针解引用都可以操作这个变量 x 的值。

实验

练习改写上述程序。

7.2 指针的算术运算

在 C 语言中，指针的值是一个无符号的整数，它表示一个内存块的起始地址（最低位置的地址）。这个整数可以用如下的运算符进行计算。

能用于指针运算的算术运算符有：

```
++      // 前置和后置自增运算自减
--      // 前置和后置自减运算
+       // 加法运算，只能和整数相加
-       // 减法运算，只能和整数或同类型的指针（用于计算间隔元素个数）相减。
```

说明：

1. 指针的算术运算都是基于其指向数据的类型的运算。指针的加一运算实际得到的地址是原地址加上指向数据占用的字节数。
2. `void*` 类型的指针可以指向任何数据类型，但不允许进行算术运算，因为编译器不知道其指向类型的大小。

例如：`char*` 类型的指针 `p1` 加一值会真正的加一，因为 `p1` 指向的类型是一个字节。但 `short int *`类型的指针 `p2` 加一时值会加二，因为 `p2` 指向的类型是二个字节。以此类推。

示例：

```c
// filename: pointer_arithmetic.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char ch = 'A';
    short int si = 100;
    int i = 999;

    char *pch = &ch;
    short int * psi = &si;
    int * pi = &i;

    printf("pch:%p, pch+1: %p, pch+4:%p\n", pch, pch + 1, pch + 4);
    printf("psi:%p, psi+1: %p, psi+4:%p\n", psi, psi + 1, psi + 4);
    printf("pi :%p, pi+1 : %p, pi+4 :%p\n", pi, pi + 1, pi + 4);

    psi++;
    pch++;
    pi++;

    printf("after: psi++; pch++; pi++;\n");
    printf("pch:%p, psi:%p, pi:%p\n", psi, pch, pi);

    return 0;
}
```

运行结果:

```
weimingze@mzstudio:~$ gcc -o pointer_arithmetic.c pointer_arithmetic.c.c
weimingze@mzstudio:~$ ./pointer_arithmetic.c
pch:0x7ffe14fe3529, pch+1: 0x7ffe14fe352a, pch+4:0x7ffe14fe352d
psi:0x7ffe14fe352a, psi+1: 0x7ffe14fe352c, psi+4:0x7ffe14fe3532
```

```
pi :0x7ffe14fe352c, pi+1 : 0x7ffe14fe3530, pi+4 :0x7ffe14fe353c
after: psi++; pch++; pi++;
pch:0x7ffe14fe352c, psi:0x7ffe14fe352a, pi:0x7ffe14fe3530
```

使用场景：

指针的算数运算对操作内存连续的多个数据的时候非常实用，比如对数组（后面会学）的数据元素进行操作是非常方便。对非数组类型的指针进行算术运算虽然语法允许，但通常很危险且容易出错。

实验：

编写程序练习指针的算术运算。

7.3 指针的关系运算

指针的关系运算符是比较指针的值的大小，即：谁在前谁在后。其实质是比较两个指针的数值（无符号整数）的大小，即：地址的大小。

指针的关系运算

```
<   // 小于：判断 左表达式 的值（地址）是否小于 右表达式 的值。
<=  // 小于等于
>   // 大于
>=  // 大于等于
==  // 相等
!=  // 不相等
```

示例：

```
// filename: pointer_relational.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i = 0;

    int *pi;

    for (pi = &i; pi < &i + 5; pi++) {
        printf("pi: %p\n", pi);
    }

    return 0;
}
```

运行结果:

```
weimingze@mzstudio:~$ gcc -o pointer_relational pointer_relational.c
weimingze@mzstudio:~$ ./pointer_relational
pi: 0x7fff8bbebdac
pi: 0x7fff8bbebdb0
pi: 0x7fff8bbebdb4
pi: 0x7fff8bbebdb8
pi: 0x7fff8bbebdbc
```

从运行结果可以看出循环内每次指针向后走 4 个字节，共循环 5 次，最后比较不成立而退出循环。

8. 二级指针

二级指针就是指向指针的指针。

前面我们学习了指针，它是一个变量，它的内部保存的是所指向的变量的地址。那么有没有一个变量能够指向一个指针呢？答案就是二级指针。

在说二级指针之前我们先来说一下指针取地址运算符（`&`）。

当我们有个变量如下：

```
int x = 100;
```

变量 x 的类型是 `int`，当我们对变量 `x` 取地址时，表达式 `&x` 返回的值是一个数字代表内存地址，表达式返回的类型是 `int*`，我们为了保存这个 `int*` 类型的数值我们需要定义一个 `int*` 类型的指针。如下面代码所示。

```
int *px = &x;
```

现在我们程序中已经声明了两个变量 `x` 和 `px`。现在我们对变量 `px` 取地址，表达式 `&px`的返回值依旧是一个地址，但因为 `px` 的类型为 `int*`，当我们再次取地址为 `&px`是，要在 `int*`类型的基础上再加一个星号才是 `&px` 的类型，表示为 `int**`。我们要在程序中保存这个地址，需要定义一个 `int**` 类型的变量，这个变量就是二级指针。代码表示如下：

```
int ** ppx = &px;
```

变量 `ppx` 就是二级指针，它保存了 变量 `px` 的地址。

那么如何使用这个二级指针呢？答案是**解引用**。当对指针 `ppx` 一次解引用（`*ppx`）则得到的是它指向的变量 `px` ，如果对 `*ppx` 再次解引用（`**ppx`）则得到了 px 指向的变量 x。即：`*ppx` 为变量 `px` 的引用， `**ppx` 为变量 x 的引用。

让我们写一个完整可运行的示例来看一下吧。

```c
// filename: pointer_to_pointer.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 100;
    int *px = &x;
    int ** ppx = &px;

    // 打印三个变量的值
    printf("x:%d, px:%p, ppx:%p\n", x, px, ppx);

    // 打印三个变量的地址
    printf("&x:%p, &px:%p, &ppx:%p\n", &x, &px, &ppx);

    // 打印两个指针解引用后的值
    printf("*px:%d, *ppx:%p\n", *px, *ppx);

    // 打印指针ppx 二次解引用后的值
    printf("**ppx:%d\n", **ppx);
}
```

运行结果如下:

```
weimingze@mzstudio:~$ gcc -o pointer_to_pointer pointer_to_pointer.c
weimingze@mzstudio:~$ ./pointer_to_pointer
x:100, px:0x7ffe58bc0af4, ppx:0x7ffe58bc0af8
&x:0x7ffe58bc0af4, &px:0x7ffe58bc0af8, &ppx:0x7ffe58bc0b00
*px:100, *ppx:0x7ffe58bc0af4
**ppx:100
```

从上述运行结果可知 变量 `px` 保存的是 `x` 的地址， `ppx` 保存的是 `px` 的地址。 通过解引用运算符 `*ppx` 的值为 `0x7ffe58bc0af4`， 即 `*ppx` 就是 `px` 的引用。 `**ppx` 的值为 100， 即 `**ppx` 就是 `x` 的引用。

二级指针同一级指针一样遵循着指针的各种运算规则。

基于以上原理，同样还可以有三级指针、四级指针、..... 。只是我们很少使用罢了。

实验：

尝试使用二级指针对多个一级指针和多个变量进行操作。

9. 指针访问内存的高级用法

本节课我们讲解指针访问内存的高级用法，本节的内容在 C 语言中非常常用，但难于理解。为了能让我的朋友们快速理解本节内容，我们先来复习一下我们前面说过的几个概念。

1. 指针的是一个变量，它的内部保存的是指向变量的内存地址。
2. **指针解引用** 返回的数据是指针指向的变量的引用。
3. **指针解引用** 后能够引用的内存数据范围和指针本身的类型相关。
4. **指针解引用** 后能够引用的数据类型是指针类型去掉一个星号（`*`）后的类型，如：指针 `p` 的类型为 `char*`，则解引用后 `*p` 的类型为 `char`。

既然 **指针解引用** 后能够引用的内存数据范围和指针本身的类型相关，那么我们就可以使用这个规则对内存进行重新定义类型，这样可以方便对内存的操作。

下面举例说明

在 IP 协议中，网络层传输的 IPv4 的地址通常用一个无符号型的 32 位整数表示，如要表示 `192.168.1.100` 的 IP 地址可以用无符号型 int 类型存储。如：

```
unsigned int ipv4 = (192 << 24) | (168 << 16) | (1 << 8) | (100 << 0);
```

那么我们如何得到每个字节的呢？我们知道 `unsigned char *` 类型的指针解引用后每次只能访问一个字节。基于这个思想。我们定义指针如下：

```
unsigned char * pc = (unsigned char *) &ipv4;
```

这样我们就使用指针 pc 指向了 变量 ipv4 的起始地址。再通过指针的运算，我们就可以得到每个字节数据了。比如：`ipv4` 最高字节的数据可以使用 `*(pc+3)` 获取；`ipv4` 最高字节的数据可以使用 `*pc` 获取。

完整示例如下：

```
// filename: pointer_adv.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    unsigned int ipv4 = (192 << 24) | (168 << 16) | (1 << 8) | (100 << 0);
    unsigned char * pc = (unsigned char *) &ipv4;

    // 以点分隔的方式打印 IPv4 的地址。
    printf("ipv4 address: %d.%d.%d.%d\n", *(pc+3), *(pc+2), *(pc+1), *(pc+0));
```



```
    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o pointer_adv pointer_adv.c
weimingze@mzstudio:~$ ./pointer_adv
ipv4 address: 192.168.1.100
```

练习：

已知一个 unsigned short int 类型的变量 s 的值为 0x0102，即十进制的 258。使用指针将 s 的高位字节修改为 0x02，低位字节修改为 0x04，最后打印修改后的值。

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    unsigned short int s = 0x0102;
    // ... 完成此处的代码。
    printf("s: %#06x\n", s);  // 打印 s: 0x0204
    return 0;
}
```

10. const 关键字

const 关键字 是 C 语言中的一个类型说明符（也叫类型限定符），它限制一个变量在操作的时候只能以只读的方式进行操作，不允许修改。

const 的中文含义是 `常量`、`常数` 的意思，但并不是说被 const 修饰的变量是常量，因为常量通常是不可变的字面值，而被 const 修饰的变量我们通常称之为 **常变量** 或 **只读变量**，而不是**常量**。

在 C 语言中，对常变量的修改编译器会报错。

常变量声明的语法:

```
const 变量类型 变量名1 = 初始值1, 变量名2 = 初始值2, ...
// 或
变量类型 const 变量名1 = 初始值1, 变量名2 = 初始值2, ...
```

说明:

1. const 关键字可以放在 **变量类型** 的前面或后面，效果是一样的。

2. 常变量在声明时必须进行初始化，否则编译器会报错。
3. const 关键字是编译器层面的类型限定符。这个类型限定符可以通过强制类型转换为非常变量的指针进行修改，但不建议这样做。

示例

```
// filename: const_variable.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    const double pi = 3.141592653589793, e = 2.718281828459045;
    int const max_size = 1024;

    printf("pi:%f, e:%f\n", pi, e);
    printf("max_size:%d\n", max_size);

    pi = 3.14; // 对常变量的修改会报错

    return 0;
}
```

编译报错如下：

```
weimingze@mzstudio:~$ gcc const_variablea.c
a.c: In function ‘main’:
a.c:10:8: error: assignment of read-only variable ‘pi’
   10 |     pi = 3.14; // 对常变量的修改会报错
      |        ^
weimingze@mzstudio:~$
```

const 关键字的作用：

1. 提高程序的健壮性，从编译器层面进行类型检查，防止变量被无意修改，减少运行时错误。
2. 提高代码可读性，开发人员看到 const 修饰的变量就知道这个变量是不应该被改变的。
3. 有助于编译器优化，编译器知道某些变量的值不会改变，可能会进行一些优化。

下面我们来学习 const 说明符在指针上的应用。

当我们声明一个指针时，比如：`char *pch;` 其实说明了两种数据类型：

1. `pch` 是指针变量，类型是 `char*`；
2. `pch` 指针指向的数据类型为 `char` 类型，即 `*pch` 解引用后是一个 `char` 类型的数据。

当我们要用 const 修饰指针时，我们要想好是指针的指向不变，还是指针指向的数据内容不变，还是指针指向和指向的内容都不变，这几种写法在语法层面是不一样的。下面我们来举例说明。

1、指针和指向的数据都可变

```
char ch1 = 'A';
char ch2 = 'B';
char *pch1 = &ch1;
```

上述写法中 `pch1 = &ch2;`是合法的， `*pch1 = 'C';` 也是合法的。即指针和指针指向的数据都可变。

2、指针指向的数据不可变

```
char ch1 = 'A';
char ch2 = 'B';
const char *pch1 = &ch1;
char const *pch2 = &ch2;
```

上述写法中 `pch1 = &ch2;`是合法的， `*pch1 = 'C';` 是不合法的。即指针可变，指针指向的数据不可变。

3、指针不可变

```
char ch1 = 'A';
char ch2 = 'B';
char * const pch1 = &ch1;
```

上述写法中 `pch1 = &ch2;`是不合法的， 但 `*pch1 = 'C';` 是合法的。即指针不可变，指针指向的数据可变。

4、指针和指针指向的数据都不可变

```
char ch1 = 'A';
char ch2 = 'B';
const char * const pch1 = &ch1;
char const * const pch2 = &ch2;
```

上述写法中 `pch1 = &ch2;`是不合法的， `*pch1 = 'C';` 也是不合法的。即指针不可变，指针指向的数据也不可变。

另外还需要说写一下 const 说明符的一个细节，就是可以为指针添加 const 说明符是允许的。但要去掉 const 说明符则需要显式调用强制类型转换。否则将在编译时报警告或错误。如下面的示例中。可以将没有 const 说明符的指针赋值给有 const 说明符的指针，但反之则会报告警告。

示例：

```c
// filename: const_alarm.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char ch1 = 'A';
    char * pch1 = &ch1;
    const char * pch2;
    char * pch3;

    pch2 = pch1; // 没有 const 修饰符的指针可以赋值给有const修饰符的指针。
    pch3 = pch2; // 此处编译报告警告

    printf("%c, %c, %c\n", *pch1, *pch2, *pch3);
    return 0;
}
```

编译运行结果如下:

```
weimingze@mzstudio:~$ gcc -o const_alarm const_alarm.c
const_alarm.c: In function ‘main’:
const_alarm.c:10:10: warning: assignment discards ‘const’ qualifier from
pointer target type [-Wdiscarded-qualifiers]
   10 |     pch3 = pch2; // 此处编译报告警告
      |          ^
weimingze@mzstudio:~$ ./const_alarm
A, A, A
```

实验：

按上述讲述内容写程序验证 const 修饰指针是的各种情况是否正确。

第十二章、数组

前面我们研究了基础数据类型和指针类型，这一章我们来学习更高级的数据类型**数组**。

在前面我们所书写的程序中，声明一个变量只能保存一个数据，比如要保存一个人的年龄信息通常这样来声明变量`int age = 35;`。但当数据量比较多的时候，比如说现在有 1000 个人的年龄信息需要保存，如果我们创建 1000 个变量，这对于程序的书写，变量的命名，程序的维护和修改都带来极大的困难。那解决此问题最好的办法是使用 **数组**。

数组（Array） 是一系列相同类型变量顺序存储的集合。它允许将一系列具有相同数据类型的数据顺序的排列在内存中，使用统一的名称对这段内存内的数据进行访问和修改。也就是说数组中的所有成员的数据类型都是相同的。

数组按着维度分类分为**一维数组**、**二维数组**和**多维数组**。这一节我们先来学习**一维数组**。

1. 一维数组

一维数组声明的语法

```
数据类型 数组名[整数表达式] = {初始值1, 初始值2, ...}, ...;
```

说明：

1. 前面的数据类型是数组内各个成员（也称为数据元素）的**数据类型**，不是数组的数据类型。
2. 数组名必须是标识符。
3. 数组名是一个常量，不能使用赋值语句对其进行赋值操作。
4. 中括号内**整数表达式**是数组内数据元素的个数，有时也称为数组长度。数组长度必须是大于等于零的整数。
5. 中括号内数组长度一般使用整数常量表达式。
    1. 在`C89`版本的编译器中，此**整数表达式**必须是编译时能确定的整数常量值。
    2. 在`C11`版本的编译器中，此**整数表达式**可以是动态运行得到的结果，此种方法创建的数组也称为**变长数组**，且变长数组必须在栈内创建（不能是全局变量或静态局部变量）。
    3. 在`C99`版本的编译器中，部分编译器如：GCC、Clang支持变长数组，MSVC 不支持变长数组。
6. `= {初始值1, 初始值2, ...}`是数组的初始化列表，此部分可以省略不写，如果不给定初始化列表，数组内的数据元素的值可能是任意值。

7. **初始化列表** 必须是用一对大括号（`{}`）括起来的表达式列表，初始化列表内表达式计算结果的类型需要能够隐式类型转化为**数据元素的数据类型**或者同**数据元素的数据类型**一致。
8. 如果 **初始化列表** 内表达式的个数小于数据元素的个数，则后面其余的数据补充 **零值**。
9. 中括号内**整数表达式**可以省略不写，如果不写整数表达式则编译器使用初始化列表来推断数组内数据元素的个数。
10. **整数表达式** 和 **初始化列表** 不能同时丢失，否则编译器则因无法确定数据元素个数而报错。
11. `, ...` 表达式后面还可以声明变量，指针，数组等其它变量。
12. 数组有自己的数据类型，对于一维数组的数据类型是 `数据类型[元素个数]`，如: `int arr[100]` 的类型是 `int[100]`。

示例：

```c
// filename: 1d_array.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    // 声明含有 5 个数据元素的数组，每个数组内的数据类型是 int 类型。
    int arr1[5];  // 数组的类型是 int[5]。

    // 声明一个含有 5 个字符型数据的数组，第一个数据是 'a', 其它都是 '\0'
    char arr2[5] = {'a'};  // 数组的类型是 char[5]。

    // 声明一个含有两个短整型数据的数组。元素个数为 2。
    short arr3[] = {100, 200};  // 数组的类型是 short int[2]。

    // 声明含有 3 个 int 型指针的数组
    int * arr4[3] = {};  // 三个空指针，数组的类型是 int*[3]。

    printf("sizeof(arr1): %ld\n", sizeof(arr1));
    printf("sizeof(arr2): %ld\n", sizeof(arr2));
    printf("sizeof(arr3): %ld\n", sizeof(arr3));
    printf("sizeof(arr4): %ld\n", sizeof(arr4));

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o 1d_array 1d_array.c
weimingze@mzstudio:~$ ./1d_array
sizeof(arr1): 20
sizeof(arr2): 5
sizeof(arr3): 4
sizeof(arr4): 24
```

由运行结果可知，数组占用的空间是 `数据元素的个数` ✖ `每个数据元素占用的内存长度`。

变长数组示例

```c
// filename: vla.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int element_count = 0;
    printf("please input student count: ");
    scanf("%d", &element_count);

    int scores[element_count];

    printf("sizeof(scores): %ld\n", sizeof(scores));

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o vla vla.c
weimingze@mzstudio:~$ ./vla
please input student count: 10
sizeof(scores): 40
```

使用变长数组需要注意以下问题：

1. 不是所有的编译器都能使用变长数组，为了程序的可移植性，尽可能少使用变长数组。
2. 变长数组只能在栈上声明，也就是说只能在函数内部声明且不可以是静态局部变量（后面会讲）。
3. 变长数组由于在栈上运行，因此可能会因数组个数的不确定导致栈溢出，导致程序运行不稳定。

实验：

1. 尝试写程序声明变长数组，看你的编译器是否支持变长数组的编译。
2. 运行上述 `vla.c` 编译后的程序，输入一个比较大的数字，查看运行结果。

2. 一维数组的索引

上一节我们学习了数组的定义。这节课我们来学习如何访问和修改数组中的数据元素。要使用数组，我们必须要了解数组的**索引**操作。

数组可以理解成现实中的火车，火车是一个整体，它是由每一节车厢组成的，每一节车厢是可以单独作为事物存在的。火车上每一节车厢都有一个唯一的编号：1、2、3、4、……，这个编号用

于标记每一节车厢。我们在乘火车出行时，用车厢号可以快速定位一节车厢来乘车。我们可以认为车厢号就是火车的索引值。

索引（Index）是引用数组中单个元素的方法，通过索引我们可以得到数组中每个数据元素的引用（即其中的单个数据）。使用索引可以访问和修改数组中的数据元素。

索引的语法

```
数组[整数表达式]
```

说明：

1. 此 **整数表达式** 叫索引值，用于表示数组中的数据元素的位置。
2. 索引是一定是大于等于零的整数。`0` 表示第一个数据元素的位置，`1` 表示第一个数据元素的位置，以此类推。
3. 数组的索引由运算符(`[]`)、数组名和整数表达式组成。`[]` 是一个二元运算符，此运算的返回值数组中的数据元素的引用。
4. 数组的索引的整数表达式可以越界（大于等于数组长度），这时可能会访问到数组以外的数据，编译器不会报错，但可能会出现运行时的错误。

示例：

写一个程序，输入任意个学生的成绩，当成绩为负数时结束输入。打印已经输入的学生的成绩。

```c
// filename: 1d_array_index.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int student_count = 0; // 用来记录学生人数
    unsigned int scores[100];  // 最多保存 100个学生的信息。

    for (int i =0; i < 100; i++) {
        int scor;
        printf("请输入成绩:");
        scanf("%d", &scor);
        if (scor < 0) // 停止输入
            break;
        scores[i] = scor;  // 保存成绩到数组中
        student_count++;// 人数+1;
    }
    // 打印学生的编号，成绩和年龄
    for (int i =0; i < student_count; i++) {
        printf("第 %d 个学生的成绩是: %d\n", i+1, scores[i]);
    }
```

```
    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o 1d_array_index 1d_array_index.c
weimingze@mzstudio:~$ ./1d_array_index
请输入成绩:100
请输入成绩:99
请输入成绩:98
请输入成绩:-1
第 1 个学生的成绩是: 100
第 2 个学生的成绩是: 99
第 3 个学生的成绩是: 98
```

练习：

写一个程序，使用数组存储多个学生的年龄（学生数量不超过 100 人），当输入学生年龄为 0 时结束输入。然后计算并打印如下信息：

1. 打印平均年龄。
2. 打印最小的学生年龄。
3. 打印最大的学生年龄。

3. 一维数组的内存结构

要真正了解 C 语言，你要了解每一种类型的内存结构。这一节我们来介绍一下一维数组的内存结构。

以整数类型的一维数组为例，比如我们声明如下的数组：

```
int myarr[3] = {100, 200, 300};
```

我们通过 sizeof 运算符计算 `sizeof(myarr)` 得到数组占用的内存数是 12 个字节。那这三个整数又是如何排列的呢?

我们先来说结论。

1. 数组的起始地址就是第一个数据元素的地址。
2. 数组中每个数据元素的起始地址是：`数组的起始地址 + 数据元素类型占用的字节数 * 索引`。

我们通过取地址运算符，对上述数组，以及数组中数据元素依次取地址，这样就可以的到每个数据在内存中的地址了。让我们一起来写程序验证吧。

示例程序

```c
// filename: 1d_array_momory.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int myarr[3] = {100, 200, 300};

    printf("&myarr: %p\n", &myarr); // 获取数组的起始地址
    printf("&myarr[0]: %p\n", &myarr[0]); // 第一个数据元素的起始地址
    printf("&myarr[1]: %p\n", &myarr[1]); // 第二个数据元素的起始地址
    printf("&myarr[2]: %p\n", &myarr[2]); // 第三个数据元素的起始地址
    printf("&myarr[3]: %p\n", &myarr[3]); // 第三个数据元素的末尾地址

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o 1d_array_momory 1d_array_momory.c
weimingze@mzstudio:~$ ./1d_array_momory
&myarr: 0x7ffdde2519dc
&myarr[0]: 0x7ffdde2519dc
&myarr[1]: 0x7ffdde2519e0
&myarr[2]: 0x7ffdde2519e4
&myarr[3]: 0x7ffdde2519e8
```

由此可见， 数组 myaddr 的内存结构如下图所示

```
           myarr[0]            myarr[1]            myarr[2]
 ----+----+----+----+----+----+----+----+----+----+----+----+----+----
 ... |        100        |        200        |        300        | ...
 ----+----+----+----+----+----+----+----+----+----+----+----+----+----
     ^                   ^                   ^                   ^
     | &myarr[0]         | &myarr[1]         | &myarr[2]         | &myarr[3]
     | &myarr            |                   |                   |
0x7ffdde2519dc    0x7ffdde2519e0      0x7ffdde2519e4      0x7ffdde2519e8
(数组的起始地址)
```

练习：

声明一个 `double` 类型的数组，通过打印地址的方式来分析数组的内存结构并在纸上画图说明。

4. 变量复制和memcpy

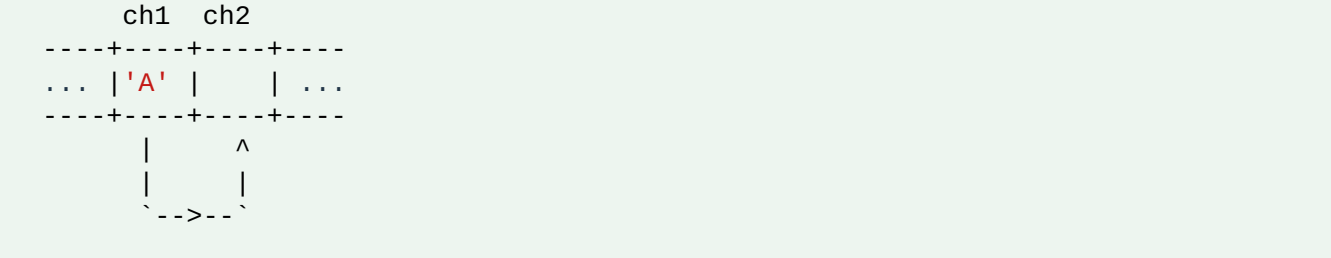
这一节我们来讲解变量的赋值和复制。本节我们将要理解赋值运算符的原理，以及 memcpy 函数的原理和用法。同时也为理解下一节《数组的复制》做准备。

我们先来了解赋值表达式的执行过程和运行结果。先看下面这段代码。

char 类型的两个变量的复制

```
char ch1 = 'A';
char ch2;

ch2 = ch1;
printf("ch1:%c, ch2:%c\n", ch1, ch2);
```

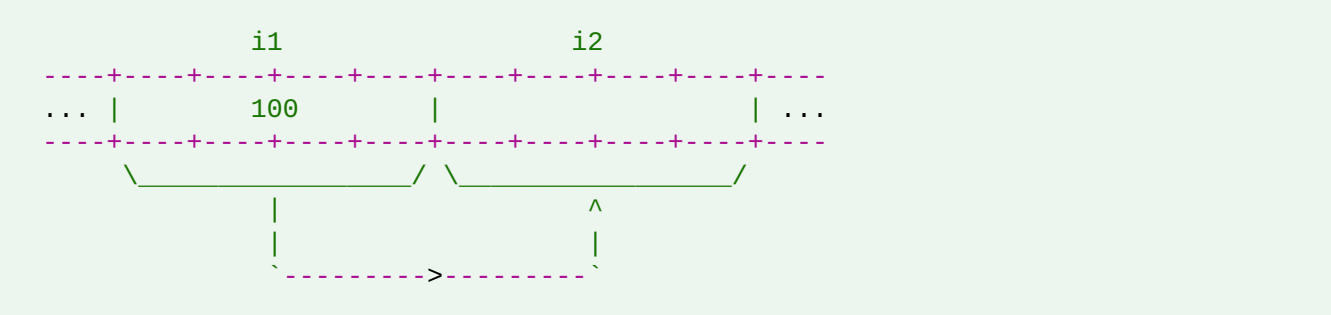
从上面的代码可以看出，其实相同类型的变量的复制的实质是将 `ch1` 变量所占内存的一个字节的值复制到了 `ch2` 变量所占用的一个字节的值。如下图所示:



同样，我们再来看一下 **int 类型的两个变量的复制**

```
int i1 = 100;
int i2;

i2 = i1;
printf("i1:%d, i2:%d\n", i1, i2);
```

在 C 语言中，整数类型的赋值语句实质是将整数（如:`i1`）占用的四个字节的内容，一次性的复制到另外一个变量（如：`i2`）的四个字节中。这就形成了如下的结果。



其实我们还有别的办法来实现内存的复制，同样可以实现整数赋值的效果，那就是 C 语言的标准库函数 `memcpy`。 `memcpy` 函数可以将开始地址的 n 个字节复制到另外一个内存地址的 n 个字节中。

memcpy 函数的声明定义在头文件 `string.h` 中。在使用前需要使用预处理指令 `#include <string.h>` 将头文件 `string.h` 的函数声明加入到当前程序中。

memcpy 函数的调用格式

```
// 头文件 string.h
void * memcpy(void * dst, const void * src, size_t n);
```

参数说明：

1. 参数 `dst` 是目标内存地址。
2. 参数 `src` 是源内存地址。
3. n 是需要复制的字节数。

返回值:

- 这个函数始终返回 dst 作为返回值。

示例：

使用 memcpy 实现整数变量的复制来代替 赋值语句。

```
// filename: int_assignment.c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int i1 = 100;
    int i2;

    // i2 = i1;  取消变量赋值，使用 memcpy 代替。
    memcpy(&i2, &i1, sizeof(i1));
    printf("i1:%d, i2:%d\n", i1, i2);

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o int_assignment int_assignment.c
weimingze@mzstudio:~$ ./int_assignment
i1:100, i2:100
```

从上面的例子可以看出，赋值运算符其实就是编译器内部实现的内存复制，不同类型的变量的赋值编译器内部会调用不同的方式的内存复制来实现赋值。

实验：

1. 使用 memcpy 函数实现不同 double 类型的两个变量的赋值。
2. 思考：能否使用 memcpy 函数实现两个不同类型的变量的赋值，如 double 和 float 类型变量的赋值?

5. 数组的复制

这一节我们来讲解数组的复制。

先说结论，数组不允许直接使用 **赋值表达式** 直接复制。

我们先看下面这段代码。

示例程序

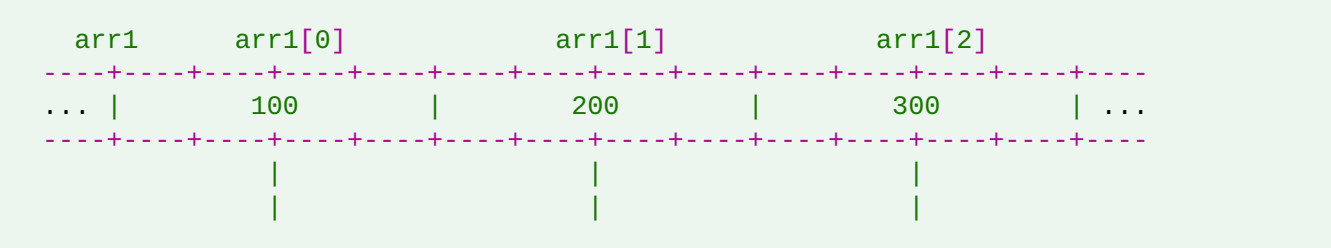
```c
// filename: array_copy.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int arr1[3] = {100, 200, 300};
    int arr2[3];

    arr2 = arr1;  // 报错，数组不允洗直接复制。

    return 0;
}
```

编辑时报错，结果如下：

```
weimingze@mzstudio:~$ gcc -o array_copy array_copy.c
temp.c: In function ‘main’:
temp.c:7:10: error: assignment to expression with array type
    7 |     arr2 = arr1;  // 报错，数组不允洗直接复制。
      |          ^
```

可见数组使用赋值表达式是不符合语法规则的。不能使用赋值表达式直接复制。

其实要实现数组的复制就是要让两个数组占用的内存的内容进行完整的复制。如下图所示：




```
  arr2       V                  V                  V
 ----+----+----+----+----+----+----+----+----+----+----+----+----
 ... |                   |                   |                   | ...
 ----+----+----+----+----+----+----+----+----+----+----+----+----
```

关于数组的复制有如下两种常用的方法：

1. 使用 memcpy 直接对两段内存的内容进行复制。
2. 使用循环对数组内的数据元素依次复制。

示例1

使用 memcpy 实现数组的复制；

```c
// filename: array_copy_by_memcpy.c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int arr1[3] = {100, 200, 300};
    int arr2[3];

    // arr2 = arr1;  // 报错，数组不允洗直接复制。
    memcpy(&arr2, &arr1, sizeof(arr1));

    // 打印复制后的结果
    for (int i = 0; i < sizeof(arr2)/sizeof(arr2[0]); i++) {
        printf("arr2[%d]: %d\n", i, arr2[i]);
    }
    return 0;
}
```

运行结果如下

```
weimingze@mzstudio:~$ gcc -o array_copy_by_memcpy array_copy_by_memcpy.c
weimingze@mzstudio:~$ ./array_copy_by_memcpy
arr2[0]: 100
arr2[1]: 200
arr2[2]: 300
```

关于上述程序的解释：

1. `memcpy(&arr2, &arr1, sizeof(arr1));` 也可以写成 `memcpy(arr2, arr1, sizeof(arr1));`，即数组取地址`&arr2` 和 直接使用数组的返回值 `arr2` 都会返回数组的起始地址，地址的值相同，只是返回的指针的数据类型不同（下一节我们在详细解释）。

2. `sizeof(arr2)/sizeof(arr2[0])` 是计算数组的元素个数。即：整个数组占用内存的字节数 除以 第一个数据元素占用的字节数。这样做的好处是当数组长度在声明时发生改变，则这段代码的返回值会动态计算，从而减少代码的改动。

示例2

使用循环对数组内的数据元素依次复制，实现数组的复制；

```c
// filename: array_copy_by_loop.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int arr1[3] = {100, 200, 300};
    int arr2[3];

    // arr2 = arr1;  // 报错，数组不允洗直接复制。
    for (int i = 0; i < sizeof(arr2)/sizeof(arr2[0]); i++) {
        arr2[i] = arr1[i]; // 依次复制数据元素。
    }

    // 打印复制后的结果
    for (int i = 0; i < sizeof(arr2)/sizeof(arr2[0]); i++) {
        printf("arr2[%d]: %d\n", i, arr2[i]);
    }
    return 0;
}
```

运行结果如下

```
weimingze@mzstudio:~$ gcc -o array_copy_by_loop array_copy_by_loop.c
weimingze@mzstudio:~$ ./array_copy_by_loop
arr2[0]: 100
arr2[1]: 200
arr2[2]: 300
```

实验：

写代码，分别用两种方法实现对 double 类型的数组的复制。

6. 指向数组的指针

这一小节我们来学习 C 语言中指向数组的指针。

关于数组，我们在使用其内部数据时通常使用索引来访问内部的数据，其实 C 语言中万能的指针也可以担此重任。有时使用指针比索引更高效。

提起指针，我们知道指针使用两部分组成的：

1. 指针的值，表示指针指向的内存地址。
2. 指针的类型，表示指针指向的地址里的数据是什么样的结构。

那我们用什么类型的指针才能指向数组呢？我们先来讲解一下数组的类型。

在数组的声明中，如果去掉变量名部分，则得到的类型就是数组的类型，如数组定义：`double data[10];` 数组 data 的类型就是去掉标识符后的类型 `double[10]`，这个我们可以通过 `printf("%ld\n", sizeof(double[10]));` 来验证并得到结果。那我们用什么类型的指针来指向这个数组 `data` 呢？即 `&data` 返回的类型是什么呢？答案是 `double[10]` 类型的指针，类型写作 `double(*)[10]`，如果声明这种类型的指针`pdata` 应当把指针变量名称写在星号的后面，写成: `double(*pdata)[10];`，你是不是很崩溃呢？但这确实是一维数组类型的指针的定义方式，我们写代码来验证一下。

示例1

指向数组的指针的声明和使用。

```
// filename: pointer_to_array.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    double data[10] = {3.1415, 2.7182};
    double(*pdata)[10];  // pdata 的类型为: double(*)[10]
    pdata = &data;

    printf("&data: %p\n", &data);
    printf("pdata: %p\n", pdata);
    printf("sizeof(data): %ld\n", sizeof(data));
    printf("sizeof(*pdata): %ld\n", sizeof(*pdata));
    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o pointer_to_array pointer_to_array.c
weimingze@mzstudio:~$ ./pointer_to_array
&data: 0x7fffb93e4330
pdata: 0x7fffb93e4330
sizeof(data): 80
sizeof(*pdata): 80
```

可见 数组 `data` 的地址 `&data` 和指针 pdata 的值相同，且数组 `data` 占用的字节数和 `pdata` 解引用后占用的字节数完全相同。即 `*pdata` 就是数组 `data` 的引用。

上述 指向数组的指针 虽然可用，但实际使用起来比较麻烦。因此我们实际在写程序的时候常用**指向数组第一个数据元素的指针**来代替**指向数组的指针**。

下面我们来讲解数组的返回值和返回值的类型。

1. 数组名称取地址的返回值是数组的起始地址，类型是数组类型的指针，即声明 `int score[5];` 中 `&score` 的返回值类型是 `int(*)[5]`;
2. 数组名称表达式的返回值是数组的起始地址，类型是数组中数据元素的类型的指针，即声明 `int score[5];` 中 `score` 的返回值类型是 `int*`;
3. 数组中索引为0的元素取地址的表达式的返回值是数组的起始地址，类型是数组中数据元素的类型的指针，即声明 `int score[5];` 中 `&score[0]` 的返回值类型也是 `int*`;

综合以上第二、第三条我们总结如下：

- 一维数组名称表达式的返回值是数组的起始地址，也是数组中第一个数据元素的起始地址。
- 一维数组名称表达式的返回类型是数组中数据元素的类型的指针。

因此我们也常使用 上述第二、第三种指针对数组进行操作。

一维数组的名字的类型

示例2

指向数组的指针的声明和使用。

```c
// filename: pointer_to_array2.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int score[5] = {100, 90, 80};
    int(*parr)[5];  // 指向数组的指针
    int * ps1;
    int * ps2;

    parr = &score;  // 取数组的地址
    ps1 = score;  // 数组名表达式的返回值
    ps2 = &score[0];  // 取第一个数据元素的地址

    // 下面我们使用多种方法对数组的第一个数据元素进行取值
    printf("score[0]: %d\n", score[0]);
    printf("(*parr)[0]: %d\n", (*parr)[0]);
    printf("*ps1: %d\n", *ps1);
    printf("*ps2: %d\n", *ps2);
    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o pointer_to_array2 pointer_to_array2.c
weimingze@mzstudio:~$ ./pointer_to_array2
score[0]: 100
(*parr)[0]: 100
*ps1: 100
*ps2: 100
```

实验：

- 使用各种指针对一个 double 类型的数组进行遍历

7. 指针的索引运算

在 C 语言中，可以用数组中数据元素类型的指针来指向数组的起始地址，也可以指向任何一个数据元素的地址。那指针能否向数组一样使用索引运算符（`[]`）来进行索引运算呢，如果能这样用那真是太方便了。答案是可以这样做。我们来看 **指针的索引运算** 的语法

语法

```
指针[整数表达式]
```

说明

1. **指针的索引运算** 的规则和一维数组索引运算的规则完全相同。
2. 在语法中，指针 p 的整数表达式索引 n 组成的表达式 `p[n]` 等同于 `*(p+n)`。
3. **指针的索引运算** 返回的是指针指向的位置起的第 n 个数据元素的引用。

示例

```c
// filename: pointer_index.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int scores[10] = {10, 20, 30, 40};
    int *p = scores;  // 用指针指向数组的起始位置

    printf("---访问数组中第一个数据元素---\n");
    printf("scores[0]: %d\n", scores[0]);
    printf("p[0]: %d\n", p[0]);
    printf("*p: %d\n", *p);

    printf("---访问数组中第三个数据元素---\n");
    printf("scores[2]: %d\n", scores[2]);
    printf("p[2]: %d\n", p[2]);
```

```
    printf("*(p+2): %d\n", *(p+2));

    return 0;
}
```

运行结果如下

```
weimingze@mzstudio:~$ gcc -o pointer_index pointer_index.c
weimingze@mzstudio:~$ ./pointer_index
---访问数组中第一个数据元素---
scores[0]: 10
p[0]: 10
*p: 10
---访问数组中第三个数据元素---
scores[2]: 30
p[2]: 30
*(p+2): 30
```

从上面的示例的运行结果可以看出。指针也可以象数组一样使用索引表达式运算。并且规则完全相同。

数组索引和指针索引的区别:

1. 数组的名称是常量，它的类型依然数组。只是数组名表达式的返回值是指针。
2. 指针变量，它的类型是指针，指针可以随时改变指向的位置。
3. sizeof 运算符对数组求长度得到的是数组占用内存的字节数。
4. sizeof 运算符对指针求长度得到的是指针的长度（4 或 8 字节）。

实验:

- 练习使用指针的索引运算来操作数组中的数据元素。

8. 二维数组

前面我们学习了一维数组，一维数组是具有相同数据类型的变量排列在一起组成的数据结构。

这一节我们来学习 C 语言中的 二维数组。二维数组你可以看成是多个相同长度的一维数组组成的数组，即：数组的数组。

在软件开发的过程中经常会用到二维数组来表示所需要的数据结构，比如：数学中的矩阵、黑白照片、五子棋或围棋的棋盘等。

在 C 语言中，我们可以认为二维数组使用行和列组成的网格结构。每个网格中的数据元素类型必须一致。

8.1 二维数组声明

二维数组声明的语法：

```
数据类型 数组名[行整数表达式m][列整数表达式n] = {...}, ...;
```

语法说明：

1. **数据类型**是数组内所有成员的**数据类型**，不是数组的数据类型。
2. **数组名**必须是标识符。
3. **数组名**是一个常量，不能使用赋值语句对其进行赋值操作。
4. 在逻辑上 **二维数组** 是 m 行 n 列的二维网格的结构，每一行的数据个数都必须相等，每一列的数据个数也必须相等。
5. 在物理存储时实际是以每一行为单位的 m 个 一维数组依次顺序摆放在内存中。
6. **行整数表达式m** 是整数常量表达式，表示二维数组的行数，在有初始化列表时可以省略不写，由初始化列表推导出行数m。
7. **列整数表达式n** 是整数常量表达式，表示二维数组的列数，此数值必须给出。
8. `= {...}` 是数组的初始化列表，此部分可以省略不写，如果不给定初始化列表，数组内的数据元素的值可能是任意值。
9. **初始化列表** 必须是用一对大括号（`{}`）括起来的表达式列表，该初始化列表用来初始化数组中每一个数据的初始值。
10. 二维数组的 **初始化列表** 的写法是 `{{第一行数据的初始化列表},{第二行数据的初始化列表}, ...}` 的格式。
11. 如果 **初始化列表** 内表达式的个数小于数据元素的个数，则后面其余的数据补充 **零值**。
12. **行整数表达式m** 和 **初始化列表** 不能同时丢失，否则编译器则因无法确定数组的行数而报错。
13. `, ...` 表达式后面还可以声明变量，指针，数组等其它变量。
14. 数组的每一行有自己的数据类型，每一行的数据类型是 `数据类型[列整数表达式n]`。
15. 数组有自己的数据类型，数组的数据类型是 `数据类型[行整数表达式m][列整数表达式n]`。

示例:

```c
// filename: 2d_array.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    // 声明 2 行 3 列一共含有 6 个整型数据元素的二维数组。
    int arr2d_1[2][3] = {{1, 2, 3}, {4, 5, 6}};

    // 声明 10 行 20 列的字符型数据的二维数组，内部数据元素都是 '\0'。
    char arr2d_2[10][20] = {};
```

```
    // 声明 3 行 4 列的短整型数据的二维数组，内部数据元素是如下结构：
    // 1 2 0 0     //(初始化列表没有给出部分补充 0)
    // 5 6 7 0
    // 0 0 0 0
    short int arr2d_3[][4] = {{1, 2}, {5, 6, 7}, {}};
    // 声明 5 行 2 列整数数据的二维数组，不进行初始化：
    int arr2d_4[5][2];

    printf("sizeof(arr2d_1): %ld\n", sizeof(arr2d_1));
    printf("sizeof(arr2d_2): %ld\n", sizeof(arr2d_2));
    printf("sizeof(arr2d_3): %ld\n", sizeof(arr2d_3));
    printf("sizeof(arr2d_4): %ld\n", sizeof(arr2d_4));
    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o 2d_array 2d_array.c
weimingze@mzstudio:~$ ./2d_array
sizeof(arr2d_1): 24
sizeof(arr2d_2): 200
sizeof(arr2d_3): 24
sizeof(arr2d_4): 40
```

下面我们来看一下 二维数组 `int arr2d_1[2][3] = {{1, 2, 3}, {4, 5, 6}};` 的逻辑结构和物理存储结构。

上述二维数组逻辑结构是 2 行 3 列的二维网格如下：

```
+-----+-----+-----+
|  1  |  2  |  3  |
+-----+-----+-----+
|  4  |  5  |  6  |
+-----+-----+-----+
```

物理存储结构是 连续的 6 个短整型数据进行存储的一段内存，如下：

```
----+-----+-----+-----+-----+-----+-----+----
... |  1  |  2  |  3  |  4  |  5  |  6  | ...
----+-----+-----+-----+-----+-----+-----+----
```

8.2 二维数组的索引

在逻辑上二维数组的是由多个一维数组组成的。因此二维数组的第一次索引会拿到某一行的一维数组，再次索引才能的获取到这一行内部的数据。

二维数组获取第 m 行数据的语法

```
二维数组名[行整数表达式m]
```

示例1

```c
// filename: 2d_array_index.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int arr2d_1[2][3] = {{1, 2, 3}, {4, 5, 6}};

    int *line1 = arr2d_1[0];  // line1 指向第一行一维数组的起始地址
    int *line2 = arr2d_1[1];  // line2 指向第二行一维数组的起始地址

    printf("*line1:%d\n", *line1);
    printf("*line2:%d\n", *line2);

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o 2d_array_index 2d_array_index.c
weimingze@mzstudio:~$ ./2d_array_index
*line1:1
*line2:4
```

有上面的运行结果可知，arr2d_1[0] 为第一行数组，arr2d_1[1]是第二行数组。如下图所示。

```
 +-----+-----+-----+
 |  1  |  2  |  3  |    <--  二维数组的第一行，即 arr2d_1[0]
 +-----+-----+-----+
 |  4  |  5  |  6  |    <--  二维数组的第一行，即 arr2d_1[1]
 +-----+-----+-----+
```

上述示例中指针 line1 和 line2 分为指向如下的物理内存的位置，如下图所示:

```
     |    arr2d_1[0]   |    arr2d_1[1]   |
 ----+-----+-----+-----+-----+-----+-----+----
 ... |  1  |  2  |  3  |  4  |  5  |  6  | ...
 ----+-----+-----+-----+-----+-----+-----+----
     ^                 ^
     | &arr2d_1        |
     | &arr2d_1[0]     | &arr2d_1[0]
     | &arr2d_1[0][0]  | &arr2d_1[1][0]
   line1             line2
```

对于二维数组使用一个索引运算符（[]）实际得到的是一维数组，那么再次使用索引运算符就可以得到内部数据的引用了。

二维数组获取第 m 行，第 n 列 数据的语法

```
二维数组名[行整数表达式m][列整数表达式n]
```

示例2

使用两重 for 循环嵌套获取遍历二维数组内的全部数据

```c
// filename: trival_2d_array.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int arr2d_1[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int row, col;

    for (row=0; row < sizeof(arr2d_1)/sizeof(arr2d_1[0]); row++) {
        for (col=0; col < sizeof(arr2d_1[0])/sizeof(arr2d_1[0][0]); col++) {
            printf("第%d行，第%d列的值为: %d\n", row+1, col+1, arr2d_1[row][col]);
        }
    }

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o trival_2d_array trival_2d_array.c
weimingze@mzstudio:~$ ./trival_2d_array
第1行，第1列的值为: 1
第1行，第2列的值为: 2
第1行，第3列的值为: 3
第2行，第1列的值为: 4
第2行，第2列的值为: 5
第2行，第3列的值为: 6
```

实验：

研究经典 2048 游戏中二维数组的算法，2048 游戏的官方网站是 https://2048game.com/。

如果你学过 Python 语言，可以下载我用 Python 语言写的 2048 游戏，并将其代码改写成 C 语言。

Python 2048 游戏下载地址：

https://gitee.com/weimz/py2048

9. 多维数组

我们把二维数组及以上维度的数组称为**多维数组**。

前面讲过了二维数组，二维数组可以认为是由多个相同长度的一维数组组合而成。这节课我们来讲解 三维数组以及更多维度的数组。

理解了二维数组，那理解三维数组、四维数组也就变的简单了。三维数组你可以认为是由多个相同长度的二维数组组合而成。同样四维数组你也可以认为是由多个相同长度的三维数组组合而成，以此类推。

三维数组在图片领域非常常用，比如一张彩色图片宽度为 w，高度为 h，有红绿蓝三种颜色组成，每一颜色用一个字节表示，那这个图片就可以用 `unsigned char` 类型的三维数组表示为 `unsigned char pic[w][h][3];`。

三维数组声明的语法：

```
数据类型 数组名[页整数表达式p][行整数表达式m][列整数表达式n] = {...}, ...;
```

四维数组声明的语法：

```
数据类型 数组名[本整数表达式b][页整数表达式p][行整数表达式m][列整数表达式n] = {...}, ...;
```

语法说明：

1. 多维数组的语法和二维数组的语法规则相同。
2. 多维数组最高维度的整数表达式可以省略不写，不写则必须由初始化列表推导得到。
3. 多维数组的索引规则同二维数组的索引规则。

示例：

```c
// filename: 3d_array.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    // 声明一个 4 页 2 行 3 列的三维数组并初始化。
    int arr3d[4][2][3] = {
        {{111, 112, 113}, {121, 122, 123}},
        {{211, 212, 213}, {221, 222, 223}},
        {{311, 312, 313}, {321, 322, 323}},
        {{411, 412, 413}, {421, 422, 423}}
    };
```



```
    printf("sizeof(arr3d):%ld\n", sizeof(arr3d));
    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o 3d_array 3d_array.c
weimingze@mzstudio:~$ ./3d_array
sizeof(arr3d):96
```

上述三维数组的逻辑结构如下：

```
       +---+---+---+
      /411 412 413/|
     +---+---+---+ |
    /311 312 313/| +
   +---+---+---+ |/
 / 211 212 213/| +
+---+---+---+  |/
|111 112 113|  +
|121 122 123| /
+---+---+---+
```

其物理存储结构在内存中依旧是连续存储（相当于多个二维数组依次排列）。

```
      _________ 第一页 _______   _________ 第二页 _______   ___
     /                       \ /                         \ /
----+---+---+---+---+---+---+---+---+---+---+---+---+---+----
... |111 112 113|121 122 123|211 212 213|221 222 223| ...
----+---+---+---+---+---+---+---+---+---+---+---+---+---+----
    |第一页第一行 |第一页第二行 |第二页第一行 |第二页第二行 |
```

基于上述的物理存储结构。通常我们使用一级指针就可以引用所有的数据元素。

实验：

自己定义不同长度的三维数组，使用取地址的方式打印行、列、页的地址，研究三维数组在计算机内存中的结构。

第十三章、字符串

字符串 是用来记录人类的文字信息的一种数据类型，在 C 语言中没有内置定义字符串类型，我们通常使用字符型一维数组来存储字符串信息。

1. 字符串和存储结构

字符串有两种存储方式：

1. 字面值
2. 字符型数组

字符串的字面值

字符串字面值用双引号（`"`） 括起来的字符序列。如：

```
"Hello World!"
"name:%s, age:%d\n"
```

字符串的字面值中，每一个字符都是使用字符的字面值构成，如上述示例中的 `\n`就是换行符的转义。

如果有些字符无法表示，则需要使用转义字符，如：

```
"I'm \"laowei\""
"这是UTF8编码的\"中文\"两个字(\xe4\xb8\xad\xe6\x96\x87)"
"这是GBK编码的\"中文\"两个字(\xd6\xd0\xce\xc4)"
```

C 语言的字符串在内存中的存储方式如下：

1. 字符串在内存中是以字符型一维数组进行存储的。
2. C 语言的字符串规定以 零字符 `'\0'` 作为结束标志。

如字符串 `laowei` 在内存中的物理存储结构，如下：

```
----+-----+-----+-----+-----+-----+-----+-----+----
... | 'l' | 'a' | 'o' | 'w' | 'e' | 'i' | '\0'| ...
----+-----+-----+-----+-----+-----+-----+-----+----
```

可见字符串 `"laowei"` 实际占用的字节数是 7 个字节（尾零占用一个字节）。

字符串的字面值的返回值是字符串字面值的起始地址，类型是`const char *`类型的指针。字符串的字面值在编译过程中它会尽可能放在只读存储区，因此对字符串字面值进行修改可能会产生不可预知的结果。

字符型数组

如果需要对字符串进行存储和修改，则需要用字符型数组来存储字符串。下面我们来说一下字符型数组的定义方式和初始化方法。

```
char s1[] = "hello";  // s1 包含尾零，共6字节。
char s2[30] = "hello";  // s2 前 5 个字节为"hello" 后面为字符0，共30字节，
char s3[] = {'h', 'e', 'l', 'l', 'o'};  // s3 不包含尾零，共5字节。
char s4[10];  // s4 长度为 10，每个字节的数值不确定。
```

示例：

```
// filename : char_array.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char s1[] = "hello";
    char s2[30] = "hello";
    char s3[] = {'h', 'e', 'l', 'l', 'o'};
    char s4[10];

    printf("sizeof(s1): %ld\n", sizeof(s1));
    printf("sizeof(s2): %ld\n", sizeof(s2));
    printf("sizeof(s3): %ld\n", sizeof(s3));
    printf("sizeof(s4): %ld\n", sizeof(s4));

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o char_array char_array.c
weimingze@mzstudio:~$ ./char_array
sizeof(s1): 6
sizeof(s2): 30
sizeof(s3): 5
sizeof(s4): 10
```

2. 字符串的输入输出

这节课我们来学习 C 语言中字符的输入输出操作。

在任何的编程语言中，字符串的输入输出都是常用的功能。我们先来学习字符串的输出。

1、字符串的输出

字符串的输出需要给出字符串的起始地址，然后将起始地址之内的所有字节的内容以文字的形式输出到控制台终端中，当遇到零值的字符时结束输出。常用于输出的标准库函数有 `printf` 和 `puts`，下面是它们的参数和用法说明。

函数	说明	备注
`int printf(const char * format, ...);`	格式化输出，占位符是 `%s`	需要手动添加换行符 `\n`
`int puts(const char *s);`	输出一行单行的字符串	自动添加换行符 `\n`

> 以上两个函数都在头文件 `stdio.h` 中声明。

示例:

```c
// filename: string_output.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    const char *website = "weimingze.com";
    char buf[100] = "hello world!";

    printf("website: %s\n", website);
    puts(website);
    printf("buf: %s\n", buf);
    puts(buf);

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o string_output string_output.c
weimingze@mzstudio:~$ ./string_output
website: weimingze.com
weimingze.com
buf: hello world!
hello world!
```

2、字符串的输入

字符串的输出需要先声明一个能够存放该字符串的缓冲区（Buffer），然后将缓冲区的起始地址交给相应的输入函数。完成输入。常用于字符串输入的标准库函数有 `scanf` 和 `fgets`，下面是它们的参数和用法说明。

函数	说明	备注
`int scanf(const char * format, ...);`	格式化输入，占位符是 `%s`	遇到空格等空白字符则会断开输入，并在读入数据后追加尾零 `'\0'`。
`char *fgets(char *s, int size, FILE * stream);`	读取一行字符串放入缓冲区 `s`，最多读入 `size` 个字符，将标准输入 `stdin` 放入 `stream` 参数	遇到换行符 `\n` 或内容超过 `size` 长度则结束输入并自动追加尾零 `'\0'`。

> 以上两个函数都在头文件 `stdio.h` 中声明。

示例：

```c
// filename: string_input.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char buf1[100];
    char buf2[100];
    char buf3[100];
    char buf4[100];

    printf("请输入空格分隔的两个字符串：");
    scanf("%s", buf1);
    scanf("%s", buf2);

    printf("请输入一行文字：");
    fgets(buf3, sizeof(buf3), stdin);  // stdin 是标准输入，默认为键盘

    printf("请输入一行文字：");
    fgets(buf4, sizeof(buf4), stdin);

    printf("buf1：%s\n", buf1);
    printf("buf2：%s\n", buf2);
    printf("buf3：%s\n", buf3);
    printf("buf4：%s\n", buf4);
```

```
    return 0;
}
```

运行结果如下

```
weimingze@mzstudio:~$ gcc -o string_input string_input.c
weimingze@mzstudio:~$ ./string_input
请输入空格分隔的两个字符串: hello world!
请输入一行文字: 请输入一行文字: hello china!
buf1: hello
buf2: world!
buf3:

buf4: hello china!

weimingze@mzstudio:~$
```

运行结果说明：

1. 在提示 `请输入空格分隔的两个字符串:` 后输入 `hello world!` 并回车，实际输入给程序的字符为 `"hello world!\n"`。
2. 这一行 `scanf("%s", buf1);` 是将 `"hello"` 交给缓存区 `buf1`, buf1 的内容为 `hello\0`，刚才输入的内容剩余 `" world!\n"`
3. 这一行 `scanf("%s", buf2);` 将 `"world!"` 交给缓存区 `buf2`, buf2 的内容为 `world!\0`，刚才输入的内容还剩余一个换行符 `"\n"`
4. 这一行 `printf("请输入一行文字: ");` 执行，打印 `请输入一行文字:` 。
5. 这一行 `fgets(buf3, sizeof(buf3), stdin);` 执行，将输入缓冲区内的`"\n"` 交给缓存区 `buf3`, buf3 的内容为 `\n\0`，刚才输入的内容经过上述两个 `scanf` 和一个 `fgets` 全部读走。此时输入缓冲区为空。
6. 这一行 `printf("请输入一行文字: ");` 执行，打印 `请输入一行文字:` 并连接在上一行之后。
7. 这一行 `fgets(buf3, sizeof(buf3), stdin);` 执行，输入 `hello china!` 后并回车，实际将输入为 `"hello china!\n"` 交给 buf4，并在后面追加尾零。

练习：

- 使用 fgets 获取一段文字的内容，计算这段字符中空格的个数并打印。

3. 字符串运算

字符串种主要是存储文字信息。对应的操作也是文字相关的操作，主要有**求长度**、**复制**、**拼接**、**比较**、**查找**等。字符串的运算我们可以自己编写算法来完成。一般最快捷的方式使用 C 语言的标准库函数进行操作。

字符串常用的运算和对应的函数如下：

1. 求长度：strlen。
2. 复制：strcpy、strncpy。
3. 拼接：strcat、strncat。
4. 比较大小：strcmp、strncmp。
5. 字符串查找：strstr。

如下列出了字符串运算常用的函数和对应的说明。

函数	说明
`size_t strlen(const char *s);`	计算字符串的长度并返回这个长度（不包含尾零 `'\0'`）
`char *strcpy(char *dst, const char *src);`	将 `src` 开始的字节复制到 `dst` 位置（包含 `src` 的尾零 `'\0'`），返回 `dst`
`char *strncpy(char *dst, const char *src, size_t n);`	将 `src` 开始的最多 n 个字节复制到 `dst` 位置，返回 `dst`，如果 strlen(src) < n 则包含尾零 `'\0'`，否则不复制尾零。
`char *strcat(char *dst, const char *src);`	将 `src` 开始的字符串复制到 `dst` 字符串尾零 `'\0'`开始的位置实现拼接，`dst` 末尾添加尾零 `'\0'`并返回 `dst`
`char *strncat(char *dst, const char *src, size_t n);`	将 `src` 开始的最多 n 个字节复制到 `dst` 字符串尾零 `'\0'`开始的位置实现拼接，`dst` 末尾添加尾零 `'\0'`并返回 `dst`
`int strcmp(const char *s1, const char *s2);`	比较字符串 `s1` 和 `s2` 的大小，相等返回 0，s1 小于 s2 返回负值， s1 大于 s2 返回正值。
`int strncmp(const char *s1, const char *s2, size_t n);`	比较字符串前n个字符 `s1` 和 `s2` 的大小，相等返回 0，s1 小于 s2 返回负值， s1 大于 s2 返回正值。
`char *strstr(const char *haystack, const char *needle);`	在 字符串 `haystack` 中查找 `needle` 是否存在，如果存在返回 `needle` 第一个字符在`haystack` 中的位置。失败返回 `NULL`。

> 以上这些函数都声明在 `string.h` 头文件中。

以下将以示例形式详细讲解上述函数。

1、求长度：strlen

示例:

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

```c
int main(int argc, char *argv[]) {
    char s1[100] = "ABC";  // 长度不包含尾零'\0'

    printf("strlen(s1): %ld\n", strlen(s1));
    printf("hello's length: %ld\n", strlen("hello"));
    return 0;
}
```

运行结果如下：

```
strlen(s1): 3
hello's length: 5
```

2、复制：strcpy、strncpy。

示例:

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

int main(int argc, char *argv[]) {
    char s1[7] = "ABCDEF";

    strcpy(s1, "123");
    printf("s1: %s\n", s1);

    strncpy(s1, "abcdefg", 4);
    printf("s1: %s\n", s1);

    return 0;
}
```

运行结果如下：

```
s1: 123
s1: abcdEF
```

上述运行结果内存结构示意图。

```
// char s1[7] = "ABCDEF";
    +-----+-----+-----+-----+-----+-----+-----+----
s1  | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | '\0'| ...
    +-----+-----+-----+-----+-----+-----+-----+----

// 执行 strcpy(s1, "123"); 后
    +-----+-----+-----+-----+-----+-----+-----+----
s1  | '1' | '2' | '3' | '\0'| 'E' | 'F' | '\0'| ...
    +-----+-----+-----+-----+-----+-----+-----+----
```

```
// 执行 strncpy(s1, "abcdefg", 4);  后
    +-----+-----+-----+-----+-----+-----+-----+----
s1  | 'a' | 'b' | 'c' | 'd' | 'E' | 'F' | '\0'| ...
    +-----+-----+-----+-----+-----+-----+-----+----
```

3、拼接：strcat、strncat

示例:

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

int main(int argc, char *argv[]) {
    char s1[9] = "AB";

    strcat(s1, "CD");
    printf("s1: %s\n", s1);

    strncat(s1, "EF", 8);
    printf("s1: %s\n", s1);

    strncat(s1, "GHIJK", 2);
    printf("s1: %s\n", s1);

    return 0;
}
```

运行结果如下：

```
s1: ABCD
s1: ABCDEF
s1: ABCDEFGH
```

上述运行结果内存结构示意图。

```
// char s1[9] = "AB";
    +-----+-----+-----+-----+-----+-----+-----+-----+-----+----
s1  | 'A' | 'B' | '\0'| '\0'| '\0'| '\0'| '\0'| '\0'| '\0'| ...
    +-----+-----+-----+-----+-----+-----+-----+-----+-----+----

// 执行 strcat(s1, "CD"); 后
    +-----+-----+-----+-----+-----+-----+-----+-----+-----+----
s1  | 'A' | 'B' | 'C' | 'D' | '\0'| '\0'| '\0'| '\0'| '\0'| ...
    +-----+-----+-----+-----+-----+-----+-----+-----+-----+----

// 执行 strncat(s1, "EF", 8); 后
    +-----+-----+-----+-----+-----+-----+-----+-----+-----+----
s1  | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | '\0'| '\0'| '\0'| ...
    +-----+-----+-----+-----+-----+-----+-----+-----+-----+----
```

```
// 执行 strncat(s1, "GHIJK", 2); 后
    +-----+-----+-----+-----+-----+-----+-----+-----+-----+----
s1  | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | '\0'| ...
    +-----+-----+-----+-----+-----+-----+-----+-----+-----+----
```

4、比较大小：strcmp、strncmp

字符串比较是两个字符串的第一个字符的值先比较，如果相同才会比较下一个，否则直接返回结果。

示例:

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

int main(int argc, char *argv[]) {
    char s1[] = "1234";
    char s2[] = "125";

    printf("strcmp(s1, s2): %d\n", strcmp(s1, s2));
    printf("strcmp(s1, s2): %d\n", strncmp(s1, s2, 2));

    printf("strcmp(\"abc\", \"acb\": %d\n", strcmp("abc", "acb"));
    printf("strcmp(\"23\", \"123\": %d\n", strcmp("23", "123"));

    return 0;
}
```

运行结果如下：

```
strcmp(s1, s2): -2
strcmp(s1, s2): 0
strcmp("abc", "acb": -1
strcmp("23", "123": 1
```

5、字符串查找：strstr

示例:

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

int main(int argc, char *argv[]) {
    char s1[100] = "hello world!";
    char * p_ret = NULL;

    p_ret = strstr(s1, "world");
```

```
    if (p_ret) {
        // 指针相减获取指针间间隔的元素个数。
        printf("找到了\"world\", 它在 s1 的: %ld 位置\n", p_ret - s1);
    } else {
        printf("没有找到\"world\"\n");
    }

    p_ret = strstr(s1, "laowei");
    printf("p_ret:%p\n", p_ret);

    return 0;
}
```

运行结果如下:

```
找到了"world", 它在 s1 的: 6 位置
p_ret:(nil)
```

练习:

- 写程序，输入任意长度的字符串，自己写算法来代替 strlen 函数求输入字符串的长度并打印。

4. 字符串和数字的互转

这节课我们来学习如何使用 C 语言的标准库函数实现字符串和数字的互转。

数字转字符串

数字转字符串可以使用 `C` 语言标准库函数 `sprintf` 来实现。 `sprintf` 是 `C89` 的标准库函数。

sprintf 函数的声明格式如下:

```
int sprintf(char *str, const char *format, ...);
```

`sprintf` 函数和 `printf` 函数是同系列的函数，同是在 `stdio.h` 文件中声明。只是 `sprintf` 是将格式化输出的内容放入一个字符型数组中并自动追加尾零 `'\0'`。 它的第二个参数 `format` 是格式化字符串，其中的 格式说明符（占位符）和 `printf` 函数的 格式说明符 完全一致。

以下是 sprintf 将数组转为字符串的示例

```
// filename: mysprintf.c
#include <stdio.h>

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

```
    int x = 999;
    float pi = 3.1415926;
    char buf_x[100];
    char buf_pi[100];

    sprintf(buf_x, "%d", x);  // buf_x 的内容为 "999"
    sprintf(buf_pi, "%.3f", pi);  // buf_pi 的内容为 " 3.142"

    printf("%s\n%s\n", buf_x, buf_pi);
    return 0;
}
```

运行结果如下:

```
weimingze@mzstudio:~$ gcc -o mysprintf mysprintf.c
weimingze@mzstudio:~$ ./mysprintf
999
3.142
```

字符串转数字

C 语言标准库中提供了如下四个常用于将 数字组成的字符串转为数字的函数

函数	说明	备注
`int atoi(const char *nptr)`	字符串 `nptr` 转为整型数。	C89 标准启用。
`long atol(const char *nptr)`	字符串 `nptr` 转为长整型数。	C89 标准启用。
`long long atoll(const char *nptr)`	字符串 `nptr` 转为 `long long` 类型整数。	C99 标准启用。
`double atof(const char *nptr)`	字符串 `nptr` 转为双精度小数。	C89 标准启用。

以上函数的在 `stdlib.h` 头文件中声明。

示例:

```
// filename: myatox.c
#include <stdio.h>
#include <stdlib.h>

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

```
    const char * pstr = "3.1415";
    int value;
    long int lvalue;
    long long int llvalue;
    double fvalue;

    value = atoi(pstr);
    lvalue = atol(pstr);
    llvalue = atoll(pstr);
    fvalue = atof(pstr);

    printf("value: %d\n", value);
    printf("lvalue: %ld\n", lvalue);
    printf("llvalue: %lld\n", llvalue);
    printf("fvalue: %f\n", fvalue);

    return 0;
}
```

运行结果如下:

```
weimingze@mzstudio:~$ gcc -o myatox myatox.c
weimingze@mzstudio:~$ ./myatox
value: 3
lvalue: 3
llvalue: 3
fvalue: 3.141500
```

练习:

写程序，输入一个由整数数字组成的字符串，不使用标准库函数，自己编写代码将字符串转化为整数，然后将这个整数 加 1 后打印结果。

5. 字符串数组

在 C 语言中，可以如何实现一个数组来保存多个字符串？这时我们会用到**字符串数组**。

字符串数组的实现方式有两种：

1. 使用字符型二维数组；
2. 使用字符型指针数组。

1、 字符型二维数组

使用字符型二维数组 可以组成一个字符串数组，例如：

```
char strs[][20] = {"first line!", "second line!"};
```

由于二维数组中的每个数据元素都是等长度的一维数组，如果有些字符串比较长，有些比较短，则会大量的浪费内存空间。

```
char strs[][20] = {"a", "second line!!!!!!!"};
```

第一行只有一个 "a" 其它的内存空间都是字符零（'\0'）。

示例：

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

int main(int argc, char *argv[]) {
    char strs[][20] = {"a", "second line!!!!!!!"};
    int i;

    for (i = 0; i < sizeof(strs)/sizeof(strs[0]); i++) {
        printf("strs[%d]: %s\n", i, strs[i]);
    }
    return 0;
}
```

运行结果如下：

```
strs[0]: a
strs[1]: second line!!!!!!!
```

使用二维数组存储字符型数组的优点是可以任意修改每一行数据的内容。缺点是可能浪费存储空间，不适合长短不一的字符串的存储。

2、字符型指针数组

在存储多个字符串时，如果字符串是字面值或者动态内存分配（后面会讲）的存储空间的字符串，则可以使用 **字符型指针数组** 来存储这些字符串。

字符型指针数组的声明方式如下：

```
const char* strs[] = {"first line!", "second line!"};
```

数组中每一个数据元素都是一个 `const char *` 类型的指针，分别指向不同字符串字面值的起始地址。这样我们就可以通过指针来访问每一个字符串了。

示例：

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

int main(int argc, char *argv[]) {
    const char* strs[] = {"first line!", "second line!"};
    int i;

    for (i = 0; i < sizeof(strs)/sizeof(strs[0]); i++) {
        printf("strs[%d]: %s\n", i, strs[i]);
    }
    return 0;
}
```

运行结果如下：

```
strs[0]: first line!
strs[1]: second line!
```

实验

使用二维数组存储多行字符串。要求从键盘读取多行字符串，保存在二维数组中。然后打印所有你输入的字符串。然后打印你输入的字符串的总长度。

第十四章、编译预处理

你还记得一个写好的 C 语言程序文件最终编译成可执行程序需要几步吗?

当一个编写好的 `.c` 文件通过编译器变为二进制可执行文件一共经过四个阶段。

这四个阶段分别是:

1. 预处理（Preprocessing）。
2. 编译（Compilation）。
3. 汇编（Assembly）。
4. 连接（Linking）。

这一章我们先学习 C 语言的编译的第一步，编译预处理的指令。

C 语言的预处理指令都是以英文的井号（`#`）开头，在编译器进行正式编译之前执行。

预处理指令有如下几种:

1. 文件包含指令：`#include`
2. 宏定义指令：`#define`。
3. 取消宏定义指令：`#undef`。
4. 条件编译指令，`#if`、`#endif`等。
5. 停止编译报错指令 `#error`。
6. 编译参数设定 `#pragma`。

这一章我们先学习前五种预处理指令。

1. 文件包含

文件包含指令 是以 `#include` 开头的指令，它的作用主要是在预处理阶段将其它文件的内容插入到当前文件。

使用 **文件包含** 的好处是当多个文件都需要使用一段共有代码的时候，每个文件都抄写一份这段代码，这增大了工作量，同时也不方便程序的修改。在这种情况下我们可以将这段共有代码放入到一个文件中，然后使用文件包含指令 `#include` 插入到需要使用这段代码的文件中即可。

在 C 语言中。一般函数的声明或者变量的声明等共有代码通常放入一个 `.h` 为后缀的文件中。我们把这个以 `.h` 结尾的文件称为头文件。因为这部分代码通常放在一个文件的最上端。

文件包含的语法有两种

```
// 方法1
#include <文件路径>
// 方法2
#include "文件路径"
```

说明：

1. `#include <文件路径>` 优先查找系统文件的路径，如果没有才查找本地路径。
2. `#include "文件路径"` 优先查找本地的路径，如果没有才查找系统文件路径。
3. `文件路径` 一般使用相对路径。

示例：

以下我们使用多个文件来实现打印一行 `hello world!`。

文件 `print.c` 内容如下：

```
printf("hello world!\n");
```

文件 `main.c` 内容如下：

```
#include <stdio.h>

int main(int argc, char *argv[]) {

#include "print.c"

    return 0;
}
```

使用 GCC 编译 main.c 运行结果如下:

```
weimingze@mzstudio:~$ gcc -o hello main.c
weimingze@mzstudio:~$ ./hello
hello world!
```

可见， `#include "print.c"` 将 `print.c` 中的内容 `printf("hello world!\n");` 插入到了 `main` 函数中。

实验：

完成上述示例代码的编写。然后使用自己的编译器查看预处理后的 `main.c` 文件的内容。如在 `GCC` 编译器下可以使用如下命令查看预处理后的结果。

```
gcc -E main.c
```

2. 宏定义

宏（Macro）是在预处理阶段，将一个字符串和一个标识符相关联，用于在预处理阶段进行字符串替换。

宏大致可以分为以下几种：

1. 不带参数的宏
2. 带有参数的宏
3. 编译器内预定义的宏

不带参数的宏定义的语法

```
#define 宏名   宏的内容
```

带有参数的宏定义的语法

```
#define 宏名(参数名1, 参数名2, ...)   宏的内容
```

说明：

1. 宏名必须是一个标识符，一般为了和变量名做区分，一般宏名都大写。
2. `#define` 和 `宏名` 以及 **宏的内容** 之间至少有一个空格或制表符分隔。
3. 宏的内容部分必须写在一行内，如果过长需要换行书写，则要在需要换行的行尾最后一个字符添加折行符（`\`）。
4. 宏的内容部分如果不需要可以为空。
5. 带有参数的宏的参数部分需要给出参数名，但不需要给出参数类型。
6. 在带有参数的宏的内容中，如果有参数名出现，则用实际传入参数代替内容部分的参数名。
7. 如果两次或以上使用 `#define` 指令定义同一个 `宏名` 则宏的定义方式必须完全相同，否则报错。

示例

1. 使用不带参数的宏 `PI` 代替圆周率的浮点数字面值 `3.1415926`。

2. 使用带有参数的宏 `AREA(rr)` 替代求圆的面积的表达式 `3.14*rr*rr`。其中 rr 是参数，在替换处会使用 `参数字符串` 替换。

```c
// filename: circle.c
#define PI         3.1415926

#define AREA(rr)   3.14*rr*rr

int main(int argc, char * argv[]) {
    double r = 10;  //圆的半径
    double length = PI * r * 2;  // 计算圆的周长。
    double area1 = PI * r * r;  // 计算面积
    double area2 = AREA(r);  // 计算面积
    printf("圆的半径: %f\n", r);
    printf("圆的周长: %f\n", length);
    printf("圆的面积1: %f\n", area1);
    printf("圆的面积2: %f\n", area1);

    return 0;
}
```

使用 `gcc -E circle.c` 命令预处理后的结果如下：

```c
weimingze@mzstudio:~$ gcc -E circle.c
# 0 "temp.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "temp.c"



int main(int argc, char * argv[]) {
    double r = 10;
    double length = 3.1415926 * r * 2;
    double area1 = 3.1415926 * r * r;
    double area2 = 3.14*r*r;
    printf("圆的半径: %f\n", r);
    printf("圆的周长: %f\n", length);
    printf("圆的面积1: %f\n", area1);
    printf("圆的面积2: %f\n", area1);

    return 0;
}
```

可见在实际使用时 `PI` 会替换成 `3.1415926`； `AREA(r)` 被替换成了 `3.14*r*r`。

需要注意的问题：

宏的替换只是源码级别的替换，不会做类型的判断。如我们在使用宏时写成`AREA(r+1)`则会被替换成为`3.14*r+1*r+1`，显然这不是我们想要的结果，因为乘法的运算符的优先级大于加法的优先级。

为了解决上述问题，我们可以尝试将宏定义`AREA(r)`的**宏的内容**部分将参数加上括号，如:`#define AREA(rr) 3.14*(rr)*(rr)`，这样`AREA(r+1)`将会替换成`3.14*(r+1)*(r+1)`，上述问的得以解决。

上述带有参数的宏中，如果再使用宏 AREA 是写成`AREA(++i)`，则会被替换成为`3.14*(++i)*(++i)`，显然使用者的原意是执行一次`++i`, 但替换后就会执行两次`++i`且结果也不正确，因此在使用宏的时候尽可能避免上述问题。

宏的优缺点

优点

1. 使用带有参数的宏定义可以象函数一样使用，但没有函数调用的开销，**执行效率高**。
2. 与类型无关，可以编写出于类型无关的通用代码。

缺点

1. 预处理阶段展开，难以调试。
2. 容易出错，带有参数的宏的参数可能会被多次求值。
3. 没有类型检查，在编辑阶段报错而非预处理阶段报错。

练习

编写带有参数的宏`MY_MAX(a,b)`返回参数`a`或`b`的最大值。

如:

```
printf("%d\n", MY_MAX(100, 200));  // 打印 200
printf("%d\n", MY_MAX(99, 88));  // 打印 99
```

3. 宏的内容中的特殊字符

我在使用带有参数的宏时，宏的内容中可以使用如下两种特殊字符来扩充宏的功能。

1. `#` 用于将参数两端加上双引号（"）成为字符串字面值。
2. `##` 用于参数代码级别的拼接

示例

下列示例中宏 `TO_STR(a)` 是将 参数a 转为字符串字面值。宏 `JOIN(a,b)` 将参数 a 和 b 合并在一起，中间加一个下划线。形成一个字符串内容的替换

```c
#include <stdio.h>

#define TO_STR(a)   #a
#define JOIN(a,b)   a##_##b

int main(int argc, char * argv[]) {
    int my_score = 100;

    printf("%s\n", TO_STR(XYZ));  // 替换为: printf("%s\n", "XYZ");
    printf("%d\n", JOIN(my, score));  // 替换为: printf("%s\n", my_score);

    return 0;
}
```

实验：

复制上述代码，使用编译器查看预处理后的结果，看上述定义的宏都替换成了什么？

4. 特殊预定义宏

在 C 语言的编译器中，已经预置好了特殊的一些预定义的宏。这个宏有利于帮助我们来为程序添加有用的信息。

下表列出了常用的特殊预定义宏

宏	说明	备注
`__DATE__`	当前源文件的编译日期（字符串），格式为："Mmm dd yyyy"，（如 "Nov 23 2025"）。	
`__TIME__`	当前源文件的编译时间（字符串），格式为："hh:mm:ss"（如 "14:30:00"）。	
`__FILE__`	当前源文件的文件名（字符串）。	
`__LINE__`	当前代码行的行号（整数）。	
`__func__`	当前所在的函数名（字符串）。	C99 启用

示例：

```c
// filename: special_macro.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    printf("当前编译日期：%s\n", __DATE__);
    printf("当前编译时间：%s\n", __TIME__);
    printf("当前编译文件：%s\n", __FILE__);
    printf("当前的行编号：%d\n", __LINE__);
    printf("当前函数名：%s\n", __func__);

    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ gcc -o special_macro special_macro.c
weimingze@mzstudio:~$ ./special_macro
当前编译日期：Nov 23 2025
当前编译时间：16:06:43
当前编译文件：special_macro.c
当前的行编号：8
当前函数名：main
```

5. 取消宏定义

在 C 语言中，使用 `#define` 宏定义指令定义的宏可以使用 `#undef` 指令取消宏定义。

语法：

```c
#undef 宏名
```

说明：

1. 如果宏名已经定义过，则取消此宏的定义。
2. 如果 宏名没有定义过，则 `#undef` 无效。

示例：

```c
// filename: undef.c
#include <stdio.h>

#define PI         3.14
```

```c
int main(int argc, char * argv[]) {
    printf("圆周率: %f\n", PI);
#undef PI  // 取消 PI 的宏定义
#define PI  3.1415926  // 重新定义宏 PI
    printf("圆周率: %f\n", PI);

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o undef undef.c
weimingze@mzstudio:~$ ./undef
圆周率: 3.140000
圆周率: 3.141593
```

可见重新定义后 PI 的值的精度增高了。

6. 条件编译

条件编译 是在 **预处理阶段** 让编译器根据给定的条件来选择性的将某些代码参与编译。它在 C 语言的**项目管理**和代码**跨平台移植** 中起到了至关重要的作用。

条件编译的指令如下

指令	说明
`#if 常量表达式`	**常量表达式**为真值则参加编译。
`#ifdef 宏名`	如果已有宏定义则参加编译。
`#ifndef 宏名`	如果没有宏定义则参加编译。
`#elif 常量表达式`	上述三种指令的子语句，用于再次判断 **常量表达式** 的值来决定是否参加编译。
`#else`	条件编译的子语句，上述条件都不成立则参加编译。
`#endif`	条件编译的结束标记
`defined(宏名)`	用于常量表达式中，如果宏名已经定义则返回 `1`，否则返回 `0`。

条件编译的语法只有三种：

语法1

```
#if 常量表达式1
  ...
#elif 常量表达式2
  ...
#else
  ...
#endif
```

语法2

```
#ifdef 宏名
  ...
#elif 常量表达式2
  ...
#else
  ...
#endif
```

语法3

```
#ifndef 宏名
  ...
#elif 常量表达式2
  ...
#else
  ...
#endif
```

说明:

1. 条件表达式必须以 `#if`、`#ifdef`、`#ifndef`其中的一个开始。
2. 条件表达式必须以 `#endif` 结束。
3. `#elif` 可以有 0 个、1 个或多个，可以用于再次判断。
4. `#else` 用于以上都不成立的情况参加编译。且只能放在最后。

示例1

根据宏定义 `ZH_CN`、`ZH_HK`、`ZH_MO`、`KO_KR` 分别定义软件不同语言的发行版。

```
// filename: cond_comp.c
#include <stdio.h>

#define ZH_CN
// #define ZH_HK
```

```
// #define ZH_MO
// #define KO_KR

int main(int argc, char * argv[]) {
#ifdef ZH_CN
    printf("欢迎来到北京!\n");
#elif defined(ZH_HK) || defined(ZH_MO)
    printf("歡迎來到北京!\n");
#elif defined(KO_KR)
    printf("베이징에 어서 오세요!\n");
#else
    printf("welcome to beijing!\n");
#endif
    return 0;
}
```

编译和运行结果如下

```
weimingze@mzstudio:~$ gcc -o cond_comp cond_comp.c
weimingze@mzstudio:~$ ./cond_comp
欢迎来到北京!
```

修改写上述代码，给 `#define ZH_CN` 添加注释，去掉 `#define ZH_HK` 的注释，如下：

```
// filename: cond_comp.c
#include <stdio.h>

// #define ZH_CN
#define ZH_HK
// #define ZH_MO
// #define KO_KR

int main(int argc, char * argv[]) {
#ifdef ZH_CN
    printf("欢迎来到北京!\n");
#elif defined(ZH_HK) || defined(ZH_MO)
    printf("歡迎來到北京!\n");
#elif defined(KO_KR)
    printf("베이징에 어서 오세요!\n");
#else
    printf("welcome to beijing!\n");
#endif
    return 0;
}
```

编译和运行结果如下

```
weimingze@mzstudio:~$ gcc -o cond_comp cond_comp.c
weimingze@mzstudio:~$ ./cond_comp
歡迎來到北京!
```

可见在使用不同的宏定义时，参与编译的 `printf` 函数也是不同的。

实验：

1. 尝试使用 `#if 常量表达式` 的方式进行条件编译，如 `#if 1` 或 `#if 100 > 50`等方式进行条件编译。

7. GCC 的 -D 编译选项

在使用 GCC 编译器对 C 语言进行编译时，可以使用 `-D` 选项临时为正在编译的文件的预处理阶段定义没有参数的宏。

> 几乎所有的 C 语言的编译器都有类似功能。请使用其它编译器的朋友自行解决各自编译器的问题。

`-D` 现象的用法如下:

定义不带参数且没有 值 的宏。

```
gcc -D 宏名 xxx.c
```

等价于在 `xxx.c` 文件的最上面添加如下一行

```
#define 宏名
```

如：`gcc -D ZH_CN xxx.c` 等同于 `#define ZH_CN`。

定义不带参数并且有 值 的宏。

```
gcc -D 宏名=值 xxx.c
```

等价于在 `xxx.c` 文件的最上面添加如下一行

```
#define 宏名  值
```

如：`gcc -D ZH_CN=100 xxx.c` 等同于 `#define ZH_CN 100`。

改写上一节条件编译的代码如下:


```c
// filename: cond_comp2.c
#include <stdio.h>

// 去掉了此处的宏定义部分。

int main(int argc, char * argv[]) {
#ifdef ZH_CN
    printf("欢迎来到北京!\n");
#elif defined(ZH_HK) || defined(ZH_MO)
    printf("歡迎來到北京!\n");
#elif defined(KO_KR)
    printf("베이징에 어서 오세요!\n");
#else
    printf("welcome to beijing!\n");
#endif
    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ gcc -o cond_comp2 cond_comp2.c
weimingze@mzstudio:~$ ./cond_comp2
welcome to beijing!
```

使用 `-D ZH_CN` 编译后运行结果如下：

```
weimingze@mzstudio:~$ gcc -D ZH_CN -o cond_comp2 cond_comp2.c
weimingze@mzstudio:~$ ./cond_comp2
欢迎来到北京!
```

使用 `-D KO_KR` 编译后运行结果如下：

```
weimingze@mzstudio:~$ gcc -D KO_KR -o cond_comp2 cond_comp2.c
weimingze@mzstudio:~$ ./cond_comp2
베이징에 어서 오세요!
```

使用 `-D ZH_HK=666` 编译后运行结果如下：

```
weimingze@mzstudio:~$ gcc -D ZH_HK=666 -o cond_comp2 cond_comp2.c
weimingze@mzstudio:~$ ./cond_comp2
歡迎來到北京!
```

实验：

- 练习使用 GCC 的 `-D` 选项进行条件编译。

8. 停止编译报错

C 语言中 预处理指令 `#error` 主要用于在编译时强制生成一个错误信息，并立即终止编译。使用 `#error` 预处理指令可以尽早的在编译阶段发现错误，而不至于在运行阶段出现致命错误。

语法

```
#error [错误信息]
```

说明

1. 错误信息通常为文字信息，用于描述错误原因。可以省略不写。

示例：

在程序用定义一个数组用于存储 100 ~ 10000 个学生的成绩，数组小于 100 不符合设计规范，数组大于 10000 则可能导致栈崩溃。结合上述需求我们编写代码如下:

```c
// filename: error.c
#include <stdio.h>

#if MAX_STUDENT_COUNT < 100
    #error MAX_STUDENT_COUNT is too small
#elif MAX_STUDENT_COUNT > 10000
    #error MAX_STUDENT_COUNT  is too large
#endif

int main(int argc, char * argv[]) {
    int i;
    int scores[MAX_STUDENT_COUNT];
    int total_count = 0;

    printf("数组的元素个数是: %d\n", MAX_STUDENT_COUNT);

    for (i = 0; i < MAX_STUDENT_COUNT; i++) {
        int scr = 0;
        scanf("%d", &scr);
        if (scr < 0) {
            break;
        }
        scores[i] = scr;
    }
    total_count = i;

    return 0;
}
```

编译运行结果如下:

```
weimingze@mzstudio:~$ gcc -D MAX_STUDENT_COUNT=1 -o error error.c
error.c:5:6: error: #error MAX_STUDENT_COUNT is too small
    5 |     #error MAX_STUDENT_COUNT is too small
      |      ^~~~~
weimingze@mzstudio:~$ gcc -D MAX_STUDENT_COUNT=9999999 -o error error.c
error.c:7:6: error: #error MAX_STUDENT_COUNT  is too large
    7 |     #error MAX_STUDENT_COUNT  is too large
      |      ^~~~~
weimingze@mzstudio:~$ gcc -D MAX_STUDENT_COUNT=1000 -o error error.c
weimingze@mzstudio:~$ ./error
数组的元素个数是: 1000
```

从上述运行结果可知，使用 gcc 的编译选项 `-D MAX_STUDENT_COUNT=1` 和 `-D MAX_STUDENT_COUNT=9999999` 都会在编译时报错，只用 `MAX_STUDENT_COUNT` 设定为合适的值，程序才能编译通过，程序才可以正常执行。

实验：

使用上述测试代码结合你编译器的预处理选项编译并执行上述程序。

第十五章、函数

函数（Function）是实现了某一功能的代码块。函数可以一次定义，多次使用。

前面我们学习使用过很多的函数，如：`printf`、`scanf`、`strlen`等。这些函数实现了输出、输入、求字符串长度功能。那这些函数从哪来的呢？我们自己能否写类似的函数呢？这一章我们来学习自己编写和使用函数。

函数是面向过程编程的重要组成部分。在编写大型软件的时候，我们通常以函数为单位来编写程序，用多个函数的相互调用来组合出复杂的软件功能。

函数 对应的英文的单词是 **Function**，个人感觉理解为**功能**更有利于学习。

使用函数机制的优点:

1. 使程序变得简短而清晰。
2. 有利于程序的维护。
3. 可以提高程序的开发效率。
4. 可以提高代码的重用性。
5. 有利于错误调试。

下面我们来学习函数的语法。

1. 函数定义和调用

在使用函数时，需要了解几个知识点：**函数定义**、**函数调用**、**函数声明**和**函数的传参**等。我们先来说一下函数的定义。

函数定义 是将一段代码块组合打包，成为一个整体。方便以后使用。

函数定义的语法

```
返回值的数据类型 函数名(数据类型1 形参变量1, 数据类型2 形参变量2, ...) {
    语句
}
```

说明：

1. 返回值的类型表示函数的返回值（函数的输出）的类型，如果函数没有返回值，可以定义为 `void` 类型。
2. 函数名必须是**标识符**
3. 函数名是一个常量，不要对其赋值。
4. 括号内的 `数据类型1 形参变量1, 数据类型2 形参变量2, ...` 被称为**形式参数列表**，它是一系列形参变量的声明列表，表示函数的输入。
5. 语句部分是函数的主体，此部分如同复合语句，它的内部可以有变量声明和语句部分。
6. 函数名的值是语句部分的代码块编译后的指令序列的起始地址。

示例

写一个函数 `myadd`，函数有两个整数输入 ，返回两个数的和。

```
int myadd(int x, int y) {
    int r; // 声明一个变量，用来保存临时结果。
    r = x + y;
    return r;
}
```

上述程序我们定义了一个名为 `myadd` 的函数，此函数的括号内`()`有两个变量声明 `int x, int y`，这个变量用来接收存入进来的数据（实际调用传递参数）。最前面的 `int` 是函数调用时调用表达式的返回类型。函数的大括号`{}`内是函数的主体部分。其中的 `return r;` 结束函数的执行，并将 `r` 的值返回给调用表达式。

下面我们再来看一下上述函数如何使用。

函数的调用

函数定义只是定义编码函数体编译一段代码

函数调用的语法

```
函数名(表达式1, 表达式2, ...)
```

说明：

1. 函数名必须是已经定义的函数。
2. `表达式1, 表达式2, ...` 被称为**实际调用传递参数列表**(也称为**实参列表**)。
3. **实参列表**的表达式的个数（实参个数）由函数的定义决定。调用时不能随意改写实参个数。

4. 实参列表内的表达式需要计算完毕后才一次性的传递给被调用函数的形式参数变量，然后再执行函数体内的语句块。

示例:

```
// filename: function_def.c
#include <stdio.h>

// 定义一个函数myadd事项两个整数相加并返回结果。
int myadd(int x, int y) {
    int r; // 声明一个变量，用来保存临时结果。
    r = x + y;
    return r;
}

int main(int argc, char *argv[]) {
    int a;
    int b;
    int result;

    printf("请输入第一个数: ");
    scanf("%d", &a);
    printf("请输入第二个数: ");
    scanf("%d", &b);

    // 第一次调用 myadd 函数，实参 a 赋值给形参变量 x, 实参 b 赋值给形参变量 y。
    result = myadd(a, b);  // 将myadd 函数调用结果保存在 result 变量中
    printf("result:%d\n", result);

    // 第二次调用，传入两个整数字面值常量并赋值给 x 和 y。
    result = myadd(1, 5);
    printf("result:%d\n", result);

    return 0;
}
```

执行结果如下:

```
weimingze@mzstudio:~$ gcc -o function_def function_def.c
weimingze@mzstudio:~$ ./function_def
请输入第一个数: 10
请输入第二个数: 20
result:30
result:6
```

练习:

- 写一个函数 mymax，此函数传入两个值，返回最大的一个值。最后由主函数来打印结果，完成如下函数的编写。

```c
#include <stdio.h>

int mymax(int a, int b) {
    // ... 完成此函数内的代码。
}

int main(int argc, char *argv[]) {

    printf("100 和 200 的最大值是: %d\n", mymax(100, 200));
    printf("9 和 1 的最大值是: %d\n", mymax(9, 1));

    return 0;
}
```

2. 函数声明

函数声明 是告诉编译器遇到的标识符的类型是什么。函数要先声明后使用。

我们先来看如下的一段代码，我们将函数的 `myadd` 的定义放到了 main 函数的下面，编译时就会报告警告。

```c
// filename: function_declaration.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("1 + 5 = %d\n", myadd(1, 5));

    return 0;
}

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

编辑信息如下：

```
weimingze@mzstudio:~$ gcc -o function_declaration function_declaration.c
function_declaration.c: In function ‘main’:
function_declaration.c:4:28: warning: implicit declaration of function ‘myadd’
[-Wimplicit-function-declaration]
    4 |     printf("1 + 5 = %d\n", myadd(1, 5));
      |                            ^~~~~
```

上述是 GCC 编译器给出的警告，原因是程序编译和我们阅读文章的顺序是一样的，程序编译时是自上而下，每一行自左向右进行分析和编译。当程序读取到标识符 `myadd` 时，程序不知道 `myadd` 是变量还是函数。因此就会给出警告或错误来提示程序编写者修改代码来解决上述问题。

上面的问题我们只需要在程序读取到 `myadd` 之前告诉编译器 `myadd` 是什么就可以了。那我们需要使用函数声明来告诉编译器 `myadd` 是含有两个整数形式参数和一个整数返回值的函数。下面我们来看一下函数声明的语法。

函数声明的语法

```
返回值的类型 函数名(数据类型1 [变量名1], 数据类型2 [变量名2], ...);
```

说明：

- 语法中 `[]` 表示其内部的形参变量名可以省略不写。
- 函数声明的标识符、形式参数的个数和类型、函数的返回值都必须和函数的定义完全一致。

示例

修改上述程序，加入函数声明如下

```c
#include <stdio.h>

// 声明 myadd 是函数，也可以写成 : int myadd(int, int);
int myadd(int x, int y);

int main(int argc, char *argv[]) {
    printf("1 + 5 = %d\n", myadd(1, 5));

    return 0;
}

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

编译和运行结果如下：

```
weimingze@mzstudio:~$ gcc -o function_declaration function_declaration.c
weimingze@mzstudio:~$ ./function_declaration
1 + 5 = 6
```

练习：

编写一个判断一个整数是否是素数的函数 `is_prime`。其功能是判断传入的参数是否是素数，如果传入的参数是素数则返回 `1`，否则返回 `0`。

素数（Prime number）也叫质数，是只能被 `1` 和自身整数的整数，如：`2`、`3`、`5`、`7`、`11`、`13`、`17`、...

要求函数的定义格式如下：

```
int is_prime(int x) {
    ...
}
```

算法描述：

1. 如果 `x` 小于 `2` 则一定不是素数。
2. 如果 `x` 大于等于 `2`，则循环使用 `2` 到 `x-1` 的整数 `y` 求余数，即 `x % y` 只要有一个余数为零，则说明 `x` 不是素数。如果余数都不为零， 说明 `x` 是素数。

> 其实还用更高效的方法请各位朋友自己去网上搜索并实现。

3. 局部变量和全局变量

3.1、局部变量

局部变量 是指函数内部声明（没有用 static修饰）的变量和函数的形参变量。

说明:

- 局部变量只有在函数运行期间被创建，函数运行结束，局部变量自动销毁，对应的内存空间也是释放待下个函数调用时使用。
- 局部变量存在于栈（Stack）上，在函数调用时，形参变量会压栈并使用，在函数调用结束后，此变量会弹栈释放。

示例:

```c
#include <stdio.h>

int myadd(int x, int y) {
    int r; // x, y, r 都是局部变量。
    r = x + y;
    return r;
}

int main(int argc, char *argv[]) {
    int result;  // argc, argv 和 result 也是局部变量。
    result = myadd(1, 5);  // 调用时在栈上创建 x、y、r变量
    // myadd 函数调用结束，myadd 内部的x、y、r 变量销毁

    printf("result:%d\n", result);
```

```
    return 0;
}
```

3.2、全局变量

全局变量 是指函数外部，`.c` 文件内部声明的变量。全局变量的特点是所有的函数可以不通过传参可以直接引用其变量。方便全局数据的存储。

说明:

- 全局变量会在程序启动时自动创建并赋初始值。
- 全局变量如果没有给出初始值，则初始值默认0值（具体要看编译器）。
- 全局变量存在程序的数据段，它会在整个程序执行过程中一直存在，在程序退出后才销毁。
  - 通常未初始化的全局变量放在程序数据段的 `BSS` 段（根据操作系统不同而不同）。
  - 通常已初始化的全局变量或静态局部变量（后面会讲）放在程序数据段的 `初始化数据段`（根据操作系统不同而不同）。
- 在程序内有同名的全局变量和局部变量，在函数内部优先访问局部变量而非全局变量（由 C 语言的作用域决定）。
- 全局变量在没有 `static` 关键字修饰的情况下，为整个程序内全局。即其他的 `.c` 文件也能够访问此变量，且不可重复声明。
- 任何一个函数都有可能改变全局变量的值，因此要慎重使用全局变量。

一个程序优先访问离自己最近的复合语句或函数内的局部变量，如果局部变量不存在才访问全局变量，如果全局变量也不存在则程序编译时报错。

示例:

```
#include <stdio.h>

// week 数组为全局变量的数组
const char * week[] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
int call_count = 0;  // call_count 为 int 型全局变量

int myadd(int x, int y) {
    call_count += 1;
    return x + y;
}

int main(int argc, char *argv[]) {
    myadd(1, 2);
    myadd(3, 4);

    printf("myadd 函数共调用 %d 次\n", call_count);
```

```
    printf("today is %s\n", week[1]);
    return 0;
}
```

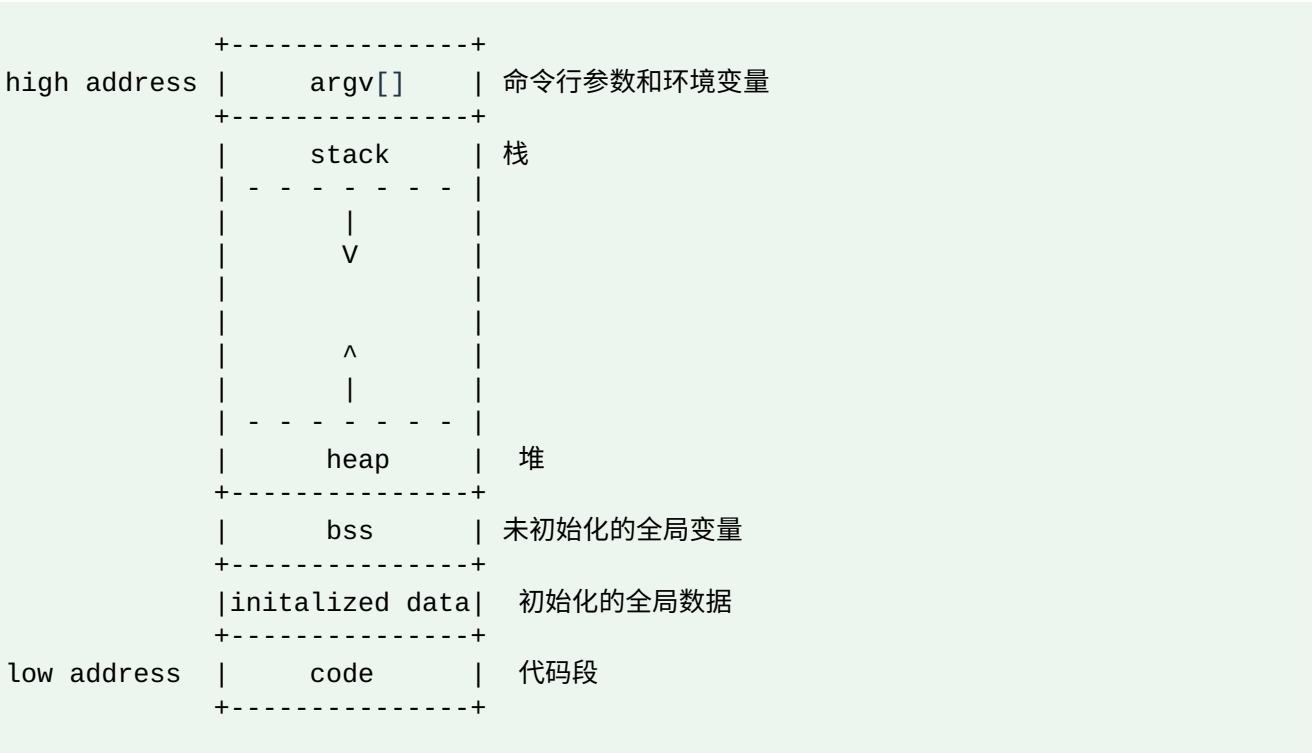
程序运行结果如下：

```
myadd 函数共调用 2 次
today is Mon
```

计算机程序在运行时，大致分为如下几部分：

1. 栈(Stack)
2. 堆(Heap)
3. 数据段(Data)
   - 初始化的数据
   - 未初始化的数据(BSS)段，其数据初始值全部置零。
4. 代码段(Code)

典型程序运行时的存储空间安排如下图所示：



其中，代码段用来存储函数编译后的二进制指令；`bbs` 和 `initalized data` 存放全局变量；`heap` 存放动态分配内存的数据；`stack` 用于存放函数内的局部变量。

练习：

写一个函数 `factorial` 求一个整数 `n` 的阶乘。

要求函数的定义格式如下:

```
int factorial(int n) {
    ...
}
```

在数学中 `n` 的阶乘用 `n!` 表示，即:

```
n!  等于 1 * 2 * 3 * ... * n-1 * n 的积。
```

如:

- `0!` 等于 `1`
- `1!` 等于 `1`
- `2!` 等于 `1*2` 等于 `2`
- `3!` 等于 `1*2*3` 等于 `6`
- 以此类推。

4. 函数的传参

这节课我们来学习函数传参，同时用变量交换的示例来说明**传值**和**传址**的用法。

先来做一道练习题

写一个程序，创建两个整数变量 x 和 y;，然后交换两个变量的值后打印结果。如下程序请填充代码:

```
#include <stdio.h>

int main(int argc, char * argv[]) {
    int x = 100, y = 200;

    // 此处交换两个变量的值。
    printf("x: %d, y: %d\n", x, y);  // 计划结果打印: x: 200, y: 100

    return 0;
}
```

问题说明:

现实世界中，如果你左手托着一个大西瓜，右手托着一个大哈密瓜，你想将左右手的两个水果交换一下，如果不允许抛起或放地上，怎么办呢？聪明的你请路人帮忙拿一下你手里的西瓜，然后你的左手就空出来了。你右手的哈密瓜倒到左手，右手接过路人手里的西瓜，此时你完成了左右手水果的交换。

对于 C 语言中交换两个变量的值一定要借助于第三个变量来临时保存一个某个变量的值，然后才能进行交换。

我们改写上述程序，使用 临时变量 `temp` 完成交换如下：

```c
// filename: myswap.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    int x = 100, y = 200;

    // 此处交换两个变量的值。
    int temp = x;
    x = y;
    y = temp;

    printf("x: %d, y: %d\n", x, y);  // 计划结果打印: x: 200, y: 100

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o myswap myswap.c
weimingze@mzstudio:~$ ./myswap
x: 200, y: 100
```

可见我们实现了 `x` 和 `y` 两个变量的交换。

为完成上述功能。我们在程序声明了一个临时变量 `temp`。这个变量在 声明后就会一直存在于程序中，它会占用空间，也可能对后续的变量命名产生冲突，我们再改进一下这个程序，我们将 临时变量 `temp` 放入一个复合语句中。因为复合语句有自己的**作用域**。在复合语句的作用域中声明的变量是复合语句内部的局部变量，只能在此作用域内有效，并优先访问最近的作用域内的变量。当复合语句执行结束后，此作用域中会释放，其中的所有声明的变量也会释放。基于这个原理。我们改写如下：

```c
// filename: myswap.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    int x = 100, y = 200;
```



```
    // 此处交换两个变量的值。
    {
        int temp = x;
        x = y;
        y = temp;
    }

    printf("x: %d, y: %d\n", x, y);  // 计划结果打印: x: 200, y: 100

    // 注意此时变量 temp 已经不存在了，不能再使用 temp 变量
    return 0;
}
```

程序的运行结果相同。

下面我们将上述程序中的交换部分的代码

```
    {
        int temp = x;
        x = y;
        y = temp;
    }
```

修改成为一个函数 `myswap` 程序改写如下:

```
// filename: myswap.c
#include <stdio.h>

void myswap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}

int main(int argc, char * argv[]) {
    int x = 100, y = 200;

    // 此处交换两个变量的值。
    myswap(x, y);

    printf("x: %d, y: %d\n", x, y);  // 计划结果打印: x: 200, y: 100

    return 0;
}
```

运行结果如下:

```
weimingze@mzstudio:~$ gcc -o myswap myswap.c
weimingze@mzstudio:~$ ./myswap
x: 100, y: 200
```

可见上述程序中并没有实现 `main` 函数内两个变量 `x` 和 `y` 的交换。为什么呢?

原因是函数 `myswap` 函数内的两个变量 `x`、`y` 并不是 `main` 函数内的 `x`、`y`，函数调用会有自己的运行空间。当 `main` 主函数调用 `myswap` 时，实参 `x`、`y`的值会赋值给 `myswap` 函数的形参变量 `x`、`y`，而这两个形参变量是 `myswap` 内的局部变量。在函数内是交换了这两个形参变量的值，但当函数退出后，这两个变量也就销毁了。而主函数中的 `x`、`y` 不会改变。

结论:

函数实参传递给形参的实质是赋值，我们也把这种传递方式称为**传值**。

那我们要如何实现在 `myswap` 函数内实现对 `main` 主函数的 `x`、`y` 进行交换呢？答案就是将 `x`、`y` 的地址传递给 `myswap` 函数，`myswap` 函数的形式参数用指针指向 `main` 函数内的 `x`、`y`。这样我们就可以在 `myswap` 函数内修改 `main` 函数内的 `x`、`y` 的值了。我们把这种传递方式称之为**传址**。

再次改写上述示例

```c
// filename: myswap.c
#include <stdio.h>

void myswap(int *px, int *py)  // 形参用指针保存实参地址
{
    int temp = *px;
    *px = *py;
    *py = temp;
}

int main(int argc, char * argv[]) {
    int x = 100, y = 200;

    // 此处交换两个变量的值。
    myswap(&x, &y);  // 传址方式调用

    printf("x: %d, y: %d\n", x, y);  // 计划结果打印: x: 200, y: 100

    return 0;
}
```

运行结果如下:

```
weimingze@mzstudio:~$ gcc -o myswap myswap.c
weimingze@mzstudio:~$ ./myswap
x: 200, y: 100
```

可见，使用 **传址** 的方式传递参数，函数 `void myswap(int *px, int *py)` 内的形参 `px`、`py` 指向了 main 函数内的 `x`、`y`，只用 `*px`、`*py` 指针解引用后就可以操作 `x`、`y`了。

传值 和 **传址** 总结：

1. **传值**和**传址**实质都是传值，只是**传址**时传递的是内存地址的值。
2. **传址**方式传参时，被调用函数可以通过形参的指针来修改上层函数内变量的值。

练习：

写一个函数 `void mydiv(int x, int y, int *quotient, int *remainder);` 实现 x ➗ y 的结果，一次性的获取商和余数，参数 `quotient` 用于保存商，`remainder` 用于保存余数。

函数的调用示例如下:

```c
int main(int argc, char * argv[]) {

    int shang = 0;  // 用于保存除法的商
    int yushu = 0;  // 用于保存除法的余数

    mydiv(14, 3, &shang, &yushu);

    printf("14 除以 3 等于 %d，余 %d\n", shang, yushu);

    return 0;
}
```

运行结果如下:

```
14 除以 3 等于 4，余 2
```

5. 数组作为函数的参数

函数的传参实质是传值，即函数的实参会赋值给函数的形参。

这节课我们来学习函数如何传递一个数组给调用函数。数组是不能够直接赋值给另一个数组，因此数组是不能够通过传值的方式传递给另一个函数。

那么如何来传递一个数组呢？答案就是实参传递数组的起始地址，形参用指针接收数组地址。数组名称的返回值就是数组的起始地址，类型为数据元素的类型的指针。

数组的返回类型

1. 一维数组的数组名返回类型为 `数据类型*`。
2. 二维数组的数组名返回类型为 `数据类型(*)[数组内一维数组元素个数]`。
3. 三维数组同二维数组，其它多维数组以此类推。

示例

```
// 一维数组
int arr1[4];  // 表达式 arr1 的返回类型为 int*

// 二维数组
int arr2[3][4];  // 表达式 arr2 的返回类型为 int(*)[4]

// 三维数组
int arr3[2][3][4];  // 表达式 arr3 的返回类型为 int(*)[3][4]
```

如果定义函数时需要在形式参数中定义形参变量来接收数组数据。则可以按着上述类型的指针来指向实参数组，在 C 语言中，也可以用类似于数组定义的方式来定义形参变量。如：

- 接收一维数组 arr1 的形参变量可以写成 `int * parr1`、`int parr1[]` 或 `int parr1[3]`。
- 接收二维数组 arr2 的形参变量可以写成 `int (* parr2)[4]`、`int parr2[][4]` 或 `int parr2[3][4]`。
- 接收三维数组 arr3 的形参变量可以写成 `int (* parr3)[3][4]`、`int parr3[][3][4]` 或 `int parr3[2][3][4]`。

说明:

- 数组的形式参数写成数组形式时，最高维度的元素个数可以省略不写。
- 数组传递的实质的传址，在函数内部可以对原数组进行修改，如果不需要修改可以为形参变量的类型前加入 `const` 关键字修饰。
- 接收数组的形参无论如何写法本质都是指针，不是数组，因此 sizeof 运算符得到的都是一个指针的字节数。

一维数组示例

在函数内求一维数组的所有数据元素的和。

```c
// filename: array1d_as_arg.c
#include <stdio.h>

// 参数使用指针写法
void arr1d_sum1(int * parr1d, int arr_size)
{
    int i, sum = 0;
    printf("sizeof(parr1): %ld\n", sizeof(parr1d));
    for (i = 0; i < arr_size; i++) {
        sum += parr1d[i];
    }
    printf("数据内数据元素的和是: %d\n", sum);
}

// 参数使用类似于数组写法，不给定数据个数
void arr1d_sum2(int parr1d[], int arr_size)
{
    int i, sum = 0;
    printf("sizeof(parr1): %ld\n", sizeof(parr1d));
    for (i = 0; i < arr_size; i++) {
        sum += parr1d[i];
    }
    printf("数据内数据元素的和是: %d\n", sum);
}

// 参数使用类似于数组写法，给定数据个数
void arr1d_sum3(int parr1d[4], int arr_size)
{
    int i, sum = 0;
    printf("sizeof(parr1): %ld\n", sizeof(parr1d));
    for (i = 0; i < arr_size; i++) {
        sum += parr1d[i];
    }
    printf("数据内数据元素的和是: %d\n", sum);
}

int main(int argc, char * argv[]) {
    int arr1[4] = {1, 2, 3, 4};

    arr1d_sum1(arr1, 4);
    arr1d_sum2(arr1, 4);
    arr1d_sum3(arr1, 4);

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o array1d_as_arg array1d_as_arg.c
array1d_as_arg.c: In function ‘arr1d_sum2’:
array1d_as_arg.c:19:42: warning: ‘sizeof’ on array function parameter ‘parr1d’ w
ill return size of ‘int *’ [-Wsizeof-array-argument]
   19 |     printf("sizeof(parr1): %ld\n", sizeof(parr1d));
      |                                          ^
```

```
array1d_as_arg.c:16:21: note: declared here
   16 | void arr1d_sum2(int parr1d[], int arr_size)
      |                 ~~~~^~~~~~~~
array1d_as_arg.c: In function ‘arr1d_sum3’:
array1d_as_arg.c:30:42: warning: ‘sizeof’ on array function parameter ‘parr1d’ w
ill return size of ‘int *’ [-Wsizeof-array-argument]
   30 |     printf("sizeof(parr1): %ld\n", sizeof(parr1d));
      |                                          ^
array1d_as_arg.c:27:21: note: declared here
   27 | void arr1d_sum3(int parr1d[4], int arr_size)
      |                 ~~~~^~~~~~~~~
weimingze@mzstudio:~$ ./array1d_as_arg
sizeof(parr1): 8
数据内数据元素的和是: 10
sizeof(parr1): 8
数据内数据元素的和是: 10
sizeof(parr1): 8
数据内数据元素的和是: 10
```

在使用 gcc 编译时，在 `arr1d_sum2` 和 `arr1d_sum3` 函数中，形参给出了数组的写法，但时间 `sizeof` 运算会给出警告，原因是它们不是数组，且 `sizeof` 返回的数据长度都是 8。

上述三个函数的形式参数的写法看似不同，实质确是完全一样的，三个函数的功能也完全一样。

二维数组示例

在函数内求二维数组的所有数据元素的和。

```c
// filename: array2d_as_arg.c
#include <stdio.h>

// 参数使用指针写法
void arr2d_sum1(int (*parr2d)[4], int row, int col)
{
    int r, c, sum = 0;

    printf("sizeof(parr1): %ld\n", sizeof(parr2d));
    for (r = 0; r < row; r++)
        for (c = 0; c < col; c++)
            sum += parr2d[r][c];

    printf("数据内数据元素的和是: %d\n", sum);
}

// 参数使用类似于数组写法，不给定数据个数
void arr2d_sum2(int parr2d[][4], int row, int col)
{
    int r, c, sum = 0;

    printf("sizeof(parr1): %ld\n", sizeof(parr2d));
    for (r = 0; r < row; r++)
        for (c = 0; c < col; c++)
            sum += parr2d[r][c];
```

```
    printf("数据内数据元素的和是: %d\n", sum);
}

// 参数使用类似于数组写法，给定数据个数
void arr2d_sum3(int parr2d[3][4], int row, int col)
{
    int r, c, sum = 0;

    printf("sizeof(parr1): %ld\n", sizeof(parr2d));
    for (r = 0; r < row; r++)
        for (c = 0; c < col; c++)
            sum += parr2d[r][c];

    printf("数据内数据元素的和是: %d\n", sum);
}

int main(int argc, char * argv[]) {
    int arr2[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};

    arr2d_sum1(arr2, 3, 4);
    arr2d_sum2(arr2, 3, 4);
    arr2d_sum3(arr2, 3, 4);

    return 0;
}
```

运行结果如下:

```
weimingze@mzstudio:~$ gcc -o array2d_as_arg array2d_as_arg.c
... 此处依旧会报告警告，请自行查看!
weimingze@mzstudio:~$ ./array2d_as_arg
sizeof(parr1): 8
数据内数据元素的和是: 78
sizeof(parr1): 8
数据内数据元素的和是: 78
sizeof(parr1): 8
数据内数据元素的和是: 78
```

上述三个函数 `arr2d_sum1`、`arr2d_sum2`、`arr2d_sum3` 只是形参列表不同，三个函数的功能完全一样。

实验:

1. 在上述一维数组中的示例中，在函数内部修改数组中的数据元素的值。然后在主函数内查看数据是否修改。
2. 尝试使用三维数组完成上述示例中的实验。

6. main 函数的参数列表

在 MacOS 和 Linux 系统中，常用的 main 函数的标准写法如下：

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

前面我们学过，main 函数的返回值表示该程序作为命令的返回值。这节课我们来学习 main 函数的形参列表。

它的形参列表的含义如下：

- argc 的值是 Shell 命令行参数的个数。
- argv 是指向 Shell 命令行参数的数组的指针，该指针指向字符型指针数组的起始位置。

假设我们写的程序 `test_argv.c` 编译后为 `test_argv`，在运行时传入如下的命令行参数。

```
./test_argv abc 123 ABC
```

则此时的 `argc` 的值为 `4`。此时形参变量 `argv` 为指向一维数组的指针。此一维数组的数据元素的个数是 `argc+1` 个，数组内各个数据元素的类型为字符型指针，值分别指向各个命令行参数，最后一个指针为空指针。如下图所示:

```
              +------+
argv ---> [0] |      | ---> "./test_argv"
              +------+
          [1] |      | ---> "abc"
              +------+
          [2] |      | ---> "123"
              +------+
          [3] |      | ---> "ABC"
              +------+
          [4] | NULL |
              +------+
```

示例：

运行时打印命令行参数

```
// filename: test_argv.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    int i;

    printf("argc: %d\n", argc);
```

```
    for (i = 0; i < argc; i++) {
        printf("argv[%d]: %s\n", i, argv[i]);
    }
    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o test_argv test_argv.c
weimingze@mzstudio:~$ ./test_argv
argc: 1
argv[0]: ./test_argv
weimingze@mzstudio:~$ ./test_argv  abc 123 ABC
argc: 4
argv[0]: ./test_argv
argv[1]: abc
argv[2]: 123
argv[3]: ABC
```

可见 执行命令 `./test_argv` 和 `./test_argv abc 123 ABC` 时，`main` 函数的形参的 `argc` 的值不同，`argv` 指向的数组的内容也不相同。

练习:

写一个程序 `myadd.c` 并生成可执行程序 `myadd`，传入的命令行参数必须为 3 个，第一个为命令，第二个和第三个参数为数字。如果参数个数不是 3 个则提示如下信息。

```
weimingze@mzstudio:~$ ./myadd
USAGE:
    ./myadd number1 number2
```

如果参数个数位正好为 3 个，则计算 第二个和第三个参数的和并打印结果，如:

```
weimingze@mzstudio:~$ ./myadd 123 456
579
```

上述程序计算两个参数 `123 + 456` 的和是 `579`。

7. 多文件编译

在使用 C 语言编写大型项目的时候，我们通常以函数为单位编写很多个函数，通常这些函数并不会放入在一个 `.c` 文件内，而是拆分成多个 `.c` 文件，最后在链接阶段才组成一个大型程序。这种由多个文件组成一个程序的编译方式我们称之为**多文件编译**。

现在假设我们有一个大型的数学计算的程序，由 多个函数组成的。其中 `myadd` 和 `mymul` 函数由张三来编写，其中的 `main` 函数由李四来编写，且会在 `main` 函数内调用张三写的 `myadd` 和 `mymul` 函数。

原本希望得到的程序结构如下：

```c
#include <stdio.h>

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

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

int main(int argc, char * argv[]) {
    printf("%d\n", myadd(100, 200));  // 300
    printf("%d\n", mymul(300, 400));  // 120000
    return 0;
}
```

但实际张三写的 `mymath.c` 文件内容如下:

```c
// filename: mymath.c
int myadd(int x, int y) {
    return x + y;
}

int mymul(int x, int y) {
    return x * y;
}
```

实际李四写的 `main.c` 文件内容如下:

```c
// filename: main.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    printf("%d\n", myadd(100, 200));  // 300
    printf("%d\n", mymul(300, 400));  // 120000
    return 0;
}
```

上述做法虽然将一个大文件拆成了两个文件，但在编译 `main.c` 文件时会报告警告或错误，原因是 `main.c` 文件中并不知道 `myadd` 和 `mymul` 这两个标识符是什么？张三为此有编写了一个 用于声明这两个标识符的文件 `mymath.h`。

`mymath.h` 文件内容如下:


```
// filename: mymath.h
#ifndef __MYMATH_H
#define __MYMATH_H

int myadd(int x, int y);
int mymul(int x, int y);

#endif
```

上述文件中写了 `myadd` 和 `mymul` 这两个函数的声明。其中 `#ifndef __MYMATH_H`、`#define __MYMATH_H` 和 `#endif` 是为了防止函数重复包含，即两次 使用 `#include "mymath.h"` 进行头文件包含时可以避免重复声明这两个函数。宏的名称通常使用文件名大写的方式，有时候前后也会加几个下划线以避免和系统定义的宏产生冲突。

张三完成了上述 `mymath.h` 头文件的书写。张三和李四再次改写这两个 `.c` 文件如下。

张三写的 `mymath.c` 文件内容加入头文件包含如下:

```
// filename: mymath.c
#include "mymath.h"

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

int mymul(int x, int y) {
    return x * y;
}
```

李四写的 `main.c` 文件内容加入头文件包含如下:

```
// filename: main.c
#include <stdio.h>
#include "mymath.h"

int main(int argc, char * argv[]) {
    printf("%d\n", myadd(100, 200));  // 300
    printf("%d\n", mymul(300, 400));  // 120000
    return 0;
}
```

这样我们就完成了三个文件：`mymath.h`、`mymath.c`、`main.c`的编写。

下面我们来将上述文件编译成一个程序 `math_proj`。假设我们三个文件都放在同一个文件夹 `myproject` 下。在 GCC 编译器下。通常我们将 .c 文件汇编成为目标文件（`.o` 文件），在由目标文件链接成为可执行程序 `math_proj`。

编译、链接和运行步骤和结果如下：

```
weimingze@mzstudio:~/myproject$ ls
main.c  mymath.c  mymath.h
weimingze@mzstudio:~/myproject$ gcc -c -o mymath.o mymath.c -I.
weimingze@mzstudio:~/myproject$ gcc -c -o main.o main.c -I.
weimingze@mzstudio:~/myproject$ gcc -o math_proj mymath.o main.o
weimingze@mzstudio:~/myproject$ ./math_proj
300
120000
```

`gcc` 编译选项说明:

- `-c` 选项是汇编成为一个目标文件。
- `-o 文件名` 是将一个结果输出到一个文件中
- `-I` 选项是将当前文件夹（`.`） 成为包含路径。能让 `#include` 找到此文件夹。
- `gcc -o math_proj mymath.o main.o` 是将两个目标文件链接成一个可执行程序 `math_proj`。

可见使用多个文件也可以将上述三个文件编译成一个可执行程序。

实验

将上述示例的源码复制到对应的文件。使用你自己的编译器完成上述程序的编译和运行。

你也可以点击此处直接下载上述示例的源码。

8. extern 关键字

extern 关键字主要用于多文件编译中声明变量名或函数名不是在当前模块（`.c`文件）内定义，而是在其它模块（`.c`文件）中定义。

extern 关键字的作用 是声明这些标识符（变量名或函数名）在汇编阶段让编译器先预留内存位置，不进行定位，而在链接阶段再根据外部的其它目标文件中这些标识符的定义来定位这些标识符存在的实际位置。

extern 关键字用于两种声明中：

1. 全局变量声明
2. 函数声明

先来说以一下**定义（Definition）**和**声明（Declaration）**的区别。

定义是为变量或函数分配内存空间，在运行时变量或函数一定存在与内存中。

声明时在编译阶段告诉编译器，一个标识符的类型是什么，编译器将根据声明的类型来决定如何处理这个标识符。

变量声明

我在模块文件 `a.c` 中的函数外写入 `int x;` 来定义一个全局变量 `x`， 这时编译器一定会在数据段为变量 x 预留 4 个字节的内存空间以便后续保存数据。当我们在另外一个模块文件 `b.c` 中 使用 `a.c` 中的全局变量 `x` 我们应当怎么办呢？因为 `a.c` 和 `b.c` 只有在链接阶段才能相遇。因此在编译和汇编 `b.c` 文件时我们先要告诉编译器 `b.c` 内的 `x` 实际是外部模块文件内的整型变量 `x`，这时我们要 `b.c` 文件内添加变量声明 `extern int x;`。这样编译才能准确的识别标识符 `x` 并为其重新定位地址做好准备。

全局变量定义的语法:

```
数据类型 变量名1[=初始值1],变量名2[=初始值2],...;
```

全局变量声明的语法:

```
extern 数据类型 变量名1,变量名2,...;
```

说明:

1. 全局变量定义前一定不能添加 `extern` 关键字。
2. 全局变量声明必须在前面添加 `extern` 关键字。
3. 局部变量不能用 `extern` 进行声明，因为局部变量只能存在于函数调用时。
4. 全局变量声明不能给初始值。

函数声明

在标准 C 语言中函数都是全局的函数，没有局部函数的写法。

函数定义的语法:

```
数据类型 函数名(数据类型1 形参变量1, 数据类型2 形参变量2, ...) { ... }
```

函数声明的语法:

```
[extern] 数据类型 函数名(数据类型1[ 形参变量1], 数据类型2[ 形参变量2], ...);
```

说明:

1. 函数定义的形参列表的右小括号后一定要跟一个大括号（`{}`），内部用来写函数的实现。
2. 函数定义后面不用分号（`;`）结尾。
3. 函数声明的形参列表的右小括号后一定要跟一个分号（`;`）。
4. 函数声明前面的 `extern` 关键字可以省略不写，也就是说函数声明自带 `extern` 属性。
5. 函数声明的形式参数列表中的形参变量名可以省略不写。
6. 函数声明的形式参数列表中如果没有形式参数，建议写入 `void` 类型（C99标准）以便编译器做出准确判断。

示例:

改写上节课的 `math_proj` 的示例，我们在 `mymath.c` 内加入整数全局变量 `total_call_times` 用来记录 `myadd` 和 `mymul` 两个函数调用的总次数。在 main 函数结束前我们来打印这两个函数的调用次数。

`mymath.c` 文件内容修改如下:

```c
// filename: mymath.c
#include "mymath.h"

// 定义全局变量 total_call_times 用来记录 以下两个函数调用的次数
int total_call_times = 0;

int myadd(int x, int y) {
    total_call_times++;  // 每次调用后全局变量个数加1
    return x + y;
}

int mymul(int x, int y) {
    total_call_times++;  // 每次调用后全局变量个数加1
    return x * y;
}
```

`mymath.h` 文件内容修改如下:

```c
// filename: mymath.h
#ifndef __MYMATH_H
#define __MYMATH_H

// 变量声明，使用者只需要包含头文件即可，不同在重复声明
extern int total_call_times;

// 函数声明，这次我们加了 extern 关键字
extern int myadd(int x, int y);
extern int mymul(int x, int y);
```

```
#endif
```

`main.c` 文件内容修改如下:

```c
// filename: main.c
#include <stdio.h>
#include "mymath.h"

int main(int argc, char * argv[]) {
    printf("%d\n", myadd(100, 200));  // 300
    printf("%d\n", mymul(300, 400));  // 120000

    printf("您共调用了 %d 次 mymath.c 内的函数\n", total_call_times);
    return 0;
}
```

程序的编译和运行结果如下:

```
weimingze@mzstudio:~/myproject2$ gcc -c -o mymath.o mymath.c -I.
weimingze@mzstudio:~/myproject2$ gcc -c -o main.o main.c -I.
weimingze@mzstudio:~/myproject2$ gcc -o math_proj mymath.o main.o
weimingze@mzstudio:~/myproject2$ ./math_proj
300
120000
您共调用了 2 次 mymath.c 内的函数
```

实验:

将你自己写过的程序，以功能为单位拆分成多个 `.c` 文件，然后进行编译和运行。

9. static 关键字

static 关键字的作用是在函数定义时修饰并限定一个标识符（函数名或变量名）它的作用域是模块内（静态全局变量或函数）或函数内（静态局部变量）的静态全局变量，而非程序内全局或局部变量。

static 关键字修饰的标识符的类型:

1. 静态全局变量和函数。
2. 静态局部变量。

1、静态全局变量和函数

要理解 static 关键字。我们先从静态全局变量和函数讲起。在多文件编译时，所有的全局变量和函数都是程序内全局，而非模块内全局。就是说如果再同一个程序中的不同模块内定义两个相同名称的全局标识符，可能会引起冲突问题。下面我们用示例来讲解。

我们现在写一个程序 static_demo ，此程序由三个文件：`a.c`、`b.c` 和 `main.c`三个文件组成。其内容如下：

文件：`a.c` 的内容如下：

```
// filename: a.c
#include <stdio.h>

int gx = 100;

int fx(void)
{
    printf("fx(void) 函数被调用!\n");
}
```

文件：`b.c` 的内容如下：

```
// filename: b.c
#include <stdio.h>

int gx = 666;

int fx(int x)
{
    printf("fx(int x) 函数被调用!\n");
}
```

文件：`main.c` 的内容如下：

```
// filename: main.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    extern int gx;
    extern int fx(void);

    printf("gx: %d\n", gx);

    fx();
    return 0;
}
```

从上述程序中，我们发现文件 `a.c` 和文件 `b.c` 内部有同名的标识符 `gx` 和 `fx`，那在 main 函数内调用的 `gx` 变量和函数 `fx` 是哪一个呢？让我们编译一下来看看。

```
weimingze@mzstudio:~/static_demo$ gcc -c -o main.o main.c -I.
weimingze@mzstudio:~/static_demo$ gcc -c -o a.o a.c -I.
weimingze@mzstudio:~/static_demo$ gcc -c -o b.o b.c -I.
weimingze@mzstudio:~/static_demo$ gcc -o static_demo main.o a.o b.o
/usr/bin/ld: b.o:(.data+0x0): multiple definition of `gx'; a.o:(.data+0x0):
first defined here
/usr/bin/ld: b.o: in function `fx':
b.c:(.text+0x0): multiple definition of `fx'; a.o:a.c:(.text+0x0): first
defined here
collect2: error: ld returned 1 exit status
```

从上述 GCC 编译过程可知，在汇编生成目标文件 `a.o`、`b.o`、`main.o` 的过程都是没有问题的。但在链接阶段要生成 `static_demo` 时报告重复定义 `gx` 和 `fx`，最终编译失败。

错误的原因是程序由三个模块`main.c`、`a.c` 和 `b.c` 组成。而 `a.c` 这三个模块中有两个 `gx` 和 `fx` 全局标识符，致使 `main` 函数无法决定调用哪一个 `gx` 变量和 `fx` 函数而报错，最终将由开发人员改写代码手动解决。

假设 `a.c` 内的 `gx` 变量只供 `a.c` 文件内的函数使用，那我们可以限定 `a.c` 内的 `gx` 变量在生成为目标文件 `a.o` 时不导出其标识符 `gx`，那么在定义这个全局变量时可以在类型前加上 `static` 关键字修饰，如: `static int gx = 100;`。`static` 修饰的标识符仅模块内可见，因此 `main` 函数将只能**看到** `b.c` 内的 `gx`，从而解决了 `gx` 重复定义的问题。

同理，我们假设 `b.c` 文件内的函数 `int fx(int x)` 也仅供 `b.c` 文件内的其它函数调用，那我们也将此函数加上 `static` 关键字修饰。这样就只有 `a.c` 文件内的 `int fx(void)` 函数全局可见了。

我们改写上述程序中的 `a.c` 和 `b.c` 如下：

文件：`a.c` 的内容修改如下：

```c
// filename: a.c
#include <stdio.h>

static int gx = 100;  // 声明为模块内全局变量

int fx(void)
{
    printf("fx(void) 函数被调用!\n");
}
```

文件：`b.c` 的内容修改如下：

```c
// filename: b.c
#include <stdio.h>

int gx = 666;
```

```
static int fx(int x)  // 声明为模块内函数。
{
    printf("fx(int x) 函数被调用!\n");
}
```

再次重新编译运行如下:

```
weimingze@mzstudio:~/static_demo$ gcc -c -o main.o main.c -I.
weimingze@mzstudio:~/static_demo$ gcc -c -o a.o a.c -I.
weimingze@mzstudio:~/static_demo$ gcc -c -o b.o b.c -I.
weimingze@mzstudio:~/static_demo$ gcc -o static_demo main.o a.o b.o
weimingze@mzstudio:~/static_demo$ ./static_demo
gx: 666
fx(void) 函数被调用!
```

可见编译正确，并能够正确运行。

注意：

需要注意的是 使用 `static` 修饰的变量或函数，不能在使用 `extern` 关键字进行声明为外部函数。

2、静态局部变量

在 C 语言中，可以使用 static 关键字来修饰局部变量。被 static 修饰的局部变量称为**静态局部变量**。

说明:

1. **静态局部变量**在定义时必须被初始化。
2. **静态局部变量** 是在数据段定义，而非在栈上定义。即可以理解成是全局变量，但只能在定义函数内部可见。

示例：

定义一个函数 `int myadd(int x, int y)` 每次调用，将返回 x + y 的和，并打印这是第几次调用。

```
// filename: static_local_var.c
#include <stdio.h>

int myadd(int x, int y)
{
    static int call_times = 0;
    call_times++;
    printf("这是第 %d 此调用 myadd函数!\n", call_times);
    return x + y;
}
```

```c
int main(int argc, char * argv[]) {
    printf("%d\n", myadd(100, 200));
    printf("%d\n", myadd(300, 400));

    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ gcc -o static_local_var static_local_var.c
weimingze@mzstudio:~$ ./static_local_var
这是第 1 此调用 myadd函数!
300
这是第 2 此调用 myadd函数!
700
```

从运行结果可见，变量 `call_times` 并不会在 `myadd` 函数调用后销毁和重新初始化。你把静态局部变量理解成 仅在定义函数内可以使用的静态全局变量就对了。

实验：

独立写代码来完成 **静态全局变量和函数** 和 **静态局部变量** 的上述示例，并通过改写来验证上述讲解是否正确。

10. 递归

递归 是指在函数内部直接或间接调用自身的函数。

递归不是 C 语言的语法，而是函数的一种特殊用法。使用递归可以很方便的解决很多复杂问题。

递归通过将问题分解为更小的同类子问题来解决问题，直到达到一个终止条件，从而避免循环和当前数据压栈、弹栈等操作。

递归的作用 是简化特殊场合的编程难度，如：遍历二叉树、快速排序等场景。

在理解递归之前我们先复习前面讲过的函数一些概念。

函数在每一次调用时会在栈上开辟一段内存空间用来保存此次函数调用时的局部变量、返回地址等信息。此局部变量直到函数返回后才弹栈销毁。当此函数没有退出时又再次调用进入下一个函数调用时，也同样会为下一个函数调用在栈上开辟新的空间来保存这次调用的局部变量，即便是当前函数在调用同名的函数自身时也是如此。

递归解决问题是分为两个阶段：

1. 递进：是指函数逐次调用，一层层进入的过程。
2. 回归：是指深层的函数逐次返回到上一次调用的地方的过程。

下面我们用递归求阶乘来说明递归的运行原理。

在数学中，`n` 的阶乘表示为 `n!`，含义是 `1 * 2 * 3 * ... * n` 的乘积的结果。如 `3!` 等于 `1 * 2 * 3` 等于 `6`。

在写递归函数之前假设我们已经完成了这个递归求阶乘的函数为 `int factorial(int n)`，这个函数传入 `n` 返回 `n!`。

求5! 递归示意

```
5! = 5 * 4!     表示为 factorial(5) = 5 * factorial(5-1);
4! = 4 * 3!     表示为 factorial(4) = 4 * factorial(4-1);
3! = 3 * 2!     表示为 factorial(3) = 3 * factorial(3-1);
2! = 2 * 1!     表示为 factorial(2) = 2 * factorial(2-1);
1! = 1          表示为 factorial(1) = 1;
```

可见求 `1!` 时，结果就一定是 `1`, 但在求 `2!` 时我们不知道 `2!` 是多少，但我们知道 `2!` 等于 `2 * 1!`，从而也可以求出 `2!`，同理 `3!` 等于 `3 * 2!` 那我也可以求出 `3!`。这样我们就可以求出 `5!` 是 `5 * 4!` 以此类推。

根据以上分析。我们来用递归的方法实现 `factorial` 函数如下:

```
int factorial(int n) {
    if (n == 1)
        return 1;
    return n * factorial(n-1);
}
```

在上述函数中，调用时如果 `n` 为 `1` 则直接返回 `1`，如果 `n` 不为 `1` 则 先 递进调用 `factorial(n-1)`，待 `factorial(n-1)` 返回后再乘以 `n` 再返回，至此达到了求阶乘的目的。至于 `factorial` 函数会递进多少次，这要看参数 `n` 的值是多少了。这种方法就是递归。

完成测试程序如下：

```
// filename: factorial.c
#include <stdio.h>

int factorial(int n) {
    if (n == 1)
        return 1;
    return n * factorial(n-1);
```



```
}

int main(int argc, char * argv[]) {
    printf("5! 等于 %d\n", factorial(5));
    printf("4! 等于 %d\n", factorial(4));
    printf("1! 等于 %d\n", factorial(1));

    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ ./factorial
5! 等于 120
4! 等于 24
1! 等于 1
```

可见，上述函数 `factorial` 实现了求阶乘的算法。如果您不了解程序调用的过程，你可以尝试在 `factorial` 函数内部添加 `printf` 打印 `n` 的值，在研究调用次序就可以明白了。

递归的优缺点

优点

- 代码简洁：适合解决分治问题（如快速排序、树的遍历等操作）。
- 更符合数学定义：如阶乘、斐波那契数列的数学描述直接对应递归代码。

缺点

- 栈溢出风险：深层递归可能导致调用栈溢出。
- 性能问题：可能存在重复计算导致性能下降。

练习：

1. 使用循环写一个函数 `int sum1_to_n(int n)` 求 `1 + 2 + 3 + ... + n` 的和。 如：`sum1_to_n(4)` 的返回值是 `10`, `sum1_to_n(10)` 的返回值是 `55`。
2. 将上述函数改为递归实现。

11. 函数指针

函数指针是指向函数的指针。通过函数指针也可以调用它指向的函数。函数指针是 C 语言实现 C++ 中多态的基础。

通过 C 语言编译器编译后的函数会成为 CPU 能够识别的指令集，这些指令集在程序时也会保存在内存的一段连续的地址空间中，这段地址空间通常是代码段中，且大多数系统为了安全起见会让代码段只读（不可修改）。

经过编译器编译后，函数名其实最终会变成代码段中某段代码的起始地址，这个地址通常是常量，不可改变。

在 C 语言中可以定义一个指针，指向代码段中的某个函数的起始地址，然后通过这个指针来调用它指向的函数。这种做法在 Linux 内核和驱动程序中非常常见。

在 C 语言中，每个函数也有自己的类型，如函数 `int fx(char c, double d) {return 0;}` 的类型为 `int(char, double)`，指向这种类型函数的指针类型为 `int(*)(char, double)`。

那么如何来声明一个 指向函数的指针呢?

函数指针的定义语法:

```
返回类型 (*指针名)(形参类型1, 形参类型2, ...) = 初始值;
```

说明:

1. 指针名必须是标识符。
2. 函数如果没有返回类型需要写成 `void` 类型
3. `(形参类型1, 形参类型2, ...)` 为形参类型，如果没有形参，则可以为 `()` 或写成 `(void)`。
4. 指针如果不赋值初始值，则 `= 初始值` 可以省略不写。
5. `= 初始值` 是为指针赋初始值，初始值必须是同类型的指针，或者为 `NULL`。

那么如何为指针赋值呢？接下来我们说一下函数名的返回值和函数名取地址运算符（`&`）。

在 C 语言中，函数名作为表达式时，它的返回值一个函数指针。对函数名取地址的表达式的返回值也同样是一个函数指针。如函数 `void fy(double a, double b){}` 的函数名 `fy` 的返回值是函数的起始地址，返回类型是 `void(*)(double, double)`。对 `fy` 取地址 `&fy` 和 fy 取值是一样的，即返回值是函数的起始地址，返回类型同样是 `void(*)(double, double)`。

使用函数指针调用函数的语法如下:

```
函数指针(实际调用参数1, 实际调用参数1, ... )
// 或
(*函数指针)(实际调用参数1, 实际调用参数1, ... )
```

上述直接使用**指针调用函数**和**指针解引用后调用函数**的写法调用结果是一样的。且函数调用同样是一个表达式。

示例：

定义一个函数指针，分别指向不同的两个函数，然后使用函数指针调用这两个函数。

```c
// filename: function_pointer.c
#include <stdio.h>

int myadd(int x, int y) {
    printf("%d + %d = %d\n", x, y, x + y);
    return x + y;
}

int mymul(int x, int y) {
    printf("%d * %d = %d\n", x, y, x * y);
    return x * y;
}

int main(int argc, char * argv[]) {
    int value;
    // 声明一个指向 int(int, int) 类型函数的函数指针 pfun
    int (*pfun)(int, int) = NULL;  // 初始值为零值

    pfun = myadd;  // 让 pfun 指向 myadd  函数
    pfun(1, 2);  // 使用指针调用 myadd 函数
    (*pfun)(3, 4);  // 使用指针解引用后调用 myadd 函数

    pfun = mymul;
    pfun(5, 6);
    (*pfun)(7, 8);

    pfun = &myadd;  // 函数取地址也同样是函数的地址。
    value = pfun(9, 10);  // 将函数调用的返回值赋值给 value
    printf("pfun(9, 10) 的返回值是 %d\n", value);

    return 0;
}
```

运行结果如下:

```
weimingze@mzstudio:~$ gcc -o function_pointer function_pointer.c
weimingze@mzstudio:~$ ./function_pointer
1 + 2 = 3
3 + 4 = 7
5 * 6 = 30
7 * 8 = 56
9 + 10 = 19
pfun(9, 10) 的返回值是 19
```

实验：

尝试定义一个函数指针的数组，数组内含有三个函数指针。让这 3 个指针指向相同参数和返回值的三个不同的函数。然后使用循环遍历这个数组中的指针，然后用指针调用不同的函数。

提示：

函数指针类型的数组的定义语法如下：

```
返回类型 (*数组名[元素个数])(形参类型1, 形参类型2, ...) = {初始值1, 初始值2, ...};
```

12. 回调函数

回调函数（Callback Function）是指通过函数指针保存函数的地址，在某个条件满足的时候再由其它函数调用的函数。

回调函数经常用于操作系统驱动，信号中断处理等调用函数者和函数实现者不同的场合。使用回调函数，可以在运行时改变处理的方式（动态调用不同的回调函数）。

示例：

比如有如下的函数 process_number_with，此函数传入一个数字和一个函数。我们要求此函数调用传入的函数 pfun 来处理此数字。

```
void process_number_with(int number, void(*pfun)(int)) {
    pfun(number);  // 调用回调函数来处理此数字。
}
```

下面我们给出三种处理数字的方法。

方法1:

使用**英文**提示打印此数字到控制台终端。

```
void print_number_with_english_hint(int n){
    printf("this number is %d.\n", n);
}
```

方法2:

使用**中文**提示打印此数字到控制台终端。

```
void print_number_with_chinese_hint(int n){
    printf("这个数是 %d。\n", n);
}
```

方法3:

将此数字保存通过网络发送给远程服务器

```
void send_number_to_server(int n){
    printf("正在发送数字 %d。\n", n);
    // ... 此处发送数据的代码省略。
}
```

我们在调用 `process_number_with` 函数时，函数的第二个参数传入上述三个函数之一，process_number_with 就会根据传入的函数调用相应的函数。我们称上述三个函数为 process_number_with 的回调函数。

如:

```
process_number_with(100, print_number_with_english_hint);
process_number_with(200, print_number_with_chinese_hint);
process_number_with(300, send_number_to_server);
```

示例代码:

```
// filename: callback_func.c
#include <stdio.h>

// 使用**英文**提示打印此数字到控制台终端。
void print_number_with_english_hint(int n){
    printf("this number is %d.\n", n);
}

// 使用**中文**提示打印此数字到控制台终端。

void print_number_with_chinese_hint(int n){
    printf("这个数是 %d。\n", n);
}

// 将此数字保存通过网络发送给远程服务器
void send_number_to_server(int n){
    printf("正在发送数字 %d。\n", n);
    // ... 此处发送数据的代码省略。
}
```

```c
void process_number_with(int number, void(*pfun)(int)) {
    pfun(number);  // 调用回调函数来处理此数字。
}

int main(int argc, char * argv[]) {
    process_number_with(100, print_number_with_english_hint);
    process_number_with(200, print_number_with_chinese_hint);
    process_number_with(300, send_number_to_server);

    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ gcc -o callback_func callback_func.c
weimingze@mzstudio:~$ ./callback_func
this number is 100.
这个数是 200。
正在发送数字 300。
```

使用回调函数的优点：

1. 可以先定义函数的主要功能和框架，后使用回调函数来动态的完善或修改数据的处理行为。
2. 可以实现调用者和功能实现者（回调函数部分）的解耦。
3. 可以添加新的回调函数来扩展软件新的功能，实现可程序的可扩展性。

实验：

1. 完成上述示例，为 `process_number_with` 再多添加几个回调函数然后来尝试调用。
2. 思考 `process_number_with` 的回调函数的参数个数和类型是否可以不同?

第十六章、结构体

结构体（Structure）是用于将不同的类型的数据通过结构体声明在逻辑上组合成一个整体的复合数据类型。它将一系列不同数据类型的数据放在一起，将其看成一个整体。这样更有利于逻辑分类。

结构体的作用 是封装相关联的数据成为一个整体，有利于数据管理。提高编程的效率。

使用结构体的函数是使代码模块化，增强可读性和易用性。

1. 结构体类型

在使用结构体之前我们要先使用结构体声明，将一些列数据类型整合成一个结构体类型，然后再使用这个新创建的结构体类型来声明变量和保存数据。

结构体类型声明的语法

```
struct [结构体名] {
    数据类型1  成员变量名1, 成员变量名2, 成员变量名3, ...;
    数据类型2  成员变量名4;
    // ... 结构体内其它成员变量
}[变量名1[={初始化列表1}]][,变量名2[={初始化列表2}], ...];
```

说明：

- `struct` 是关键字。
- 语法中，中括号（`[]`）括起来的部分代表可以省略不写。
- 结构体名必须是标识符。
- 前面 `struct [结构体名] { ... }` 部分是类型声明。后面 `[变量名1][,变量名2, ...]`是变量定义。即类型声明后可以直接定义该结构体类型的变量。
- 以上声明的结构体的类型是 `struct 结构体名`。
- 结构体名和可以省略不写。如果不给出**结构体名**则后续无法为该类型的结构体创建变量，此时则必须在声明结构体时创建变量。
- `{...}`部分是结构体内成员变量的声明内容。
- 成员变量的数据类型可以是基础的数据类型、指针、结构体、联合体、枚举等任意数据类型。
- 成员变量名的声明方法同普通变量的声明方法相同，但不允许在声明时初始化该成员变量。
- 结构体可以在声明后立即定义结构体类型的变量并初始化。

- 结构体声明的后面要跟一个分号（`;`）表示声明结束。

示例:

声明一个学生（`student`）类型的结构体，用于存放每一个学生的姓名（name），年龄（age）和身高（height）信息。

```
struct student {
   char name[32];  // 姓名
   int age;  // 年龄
   float height; // 身高
};
```

以上我们声明了一个新的类型 `struct student`，这个类型的变量的内部有三个成员，分别是`name` 占 32 个字节，`age` 占 4 个字节，`height` 占 4 个字节。也就是说每一个 `struct student` 类型的变量至少占用 40 字节（`32+4+4`）。这里说至少占 40 个字节，那就是说还有可能会多于 40 个字节，这是因为结构体对齐问题引起的。我们后面会讲。

使用结构体类型声明变量的语法和使用内建数据类型声明变量的语法相同，但需要在结构体类型名前加入关键字 `struct`。

使用结构体声明变量的语法：

1、在声明结构体时直接声明结构体变量。

```
struct [结构体名] {
    // ... 结构体内成员变量
} 变量名[ = {成员变量1初始值, 成员变量1初始值, ... }];
```

2、先声明结构体，然后再使用结构体类型创建结构体变量。

```
// 语法1：按成员声明顺序初始化成员变量
struct 结构体名 变量名 = {成员变量1初始值, 成员变量1初始值, ... };

// 语法2：按成员变量名初始化成员变量
struct 结构体名 变量名 = {.成员变量1 = 初始值1, .成员变量2 = 初始值2, ... };
```

说明:

1. 语法中的中括号（`[]`）代表其中的内容可以省略。
2. 结构体名前面必须添加关键字 `struct`

3. 等号和后面的初始化列表可以省略不写，如果不写入初始化列表，该结构体各个成员变量的值可能是任意值。
4. 如果声明结构体变量时给出的初始化列表，则没有指定初始值的成员变量的初始值为**零值**。

我们使用`struct student`类型声明变量，用于保存学生张三、李四、王五和100个学生信息的数组 class1。

参考代码如下：

```
struct student {
   char name[32];  // 姓名
   int age;  // 年龄
   float height; // 身高
} zhang3 = {"zhang san", 18, 1.73};

// 初始化列表的顺序必须和结构体成员变量名声明顺序相同。
struct student li4 = {"li si", 19, 1.80};

// 成员变量初始化顺序可以按成员变量名给出。
struct student wang5 = {.age = 16, .height = 1.71, .name = "wang wu"};

// 定义100个学生类型的数组 class1;
struct student class1[100];
```

成员访问运算符

我们定义的结构体类型的变量，那我们要如何访问结构体变量内部的数据呢？接下来我们学习 C 语言中专门用来访问结构体（或联合体，后面会讲）成员的**成员访问运算符**（`.`）。

成员访问运算符（`.`）的作用是引用结构体（或联合体）内部的成员变量。

语法:

```
结构体变量.成员变量
```

使用`.`运算符我们就可以访问（取值）或修改（赋值）结构体成员的变量了。

综合示例如下：

```
// filename: struct_declaration.c
#include <stdio.h>

struct student {
   char name[32];  // 姓名
```

```
    int age;  // 年龄
    float height; // 身高
} zhang3 = {"zhang san", 18, 1.73};

int main(int argc, char * argv[]) {
    // 初始化列表的顺序必须和结构体成员变量名声明顺序相同。
    struct student li4 = {"li si", 19, 1.80};

    // 成员变量初始化顺序可以按成员变量名给出。
    struct student wang5 = {.age = 16, .height = 1.71, .name = "wang wu"};

    printf("姓名:%s,年龄:%d,身高:%.2f\n", zhang3.name, zhang3.age, zhang3.height);
    printf("姓名:%s,年龄:%d,身高:%.2f\n", li4.name, li4.age, li4.height);
    printf("姓名:%s,年龄:%d,身高:%.2f\n", wang5.name, wang5.age, wang5.height);

    // 打印结构体变量占用的内存空间的大小。
    printf("sizeof(zhang3): %ld\n", sizeof(zhang3));
    printf("sizeof(struct student): %ld\n", sizeof(struct student));

    return 0;
}
```

运行结果如下:

```
weimingze@mzstudio:~$ gcc -o struct_declaration struct_declaration.c
weimingze@mzstudio:~$ ./struct_declaration
姓名:zhang san,年龄:18,身高:1.73
姓名:li si,年龄:19,身高:1.80
姓名:wang wu,年龄:16,身高:1.71
sizeof(zhang3): 40
sizeof(struct student): 40
```

练习:

循环输入几个学生的姓名，年龄和身高，当输入年龄输入负数时结束输入，打印你输入的所有学生的姓名，年龄和身高。

2. 结构体指针

这节课我们来学习指向结构体的指针。

前面学过指针。指针的值实际就是一段内存地址的起始地址。指针的类型是使用指针的关键，它决定了编译器对指针解引用后的地址中的数据如何操作。

结构体指针 就是指向结构体的指针。我们可以通过结构体指针来将一段内存看成是一个结构体。然后对其进行操作。

结构体指针同样是 C 语言中非常常用且重要的使用场景。

结构体指针的定义语法

```
struct 结构体名 *结构体指针名1, *结构体指针名2, ...;
```

从上面的语法可以看出，结构体指针的定义语法和其它指针的定义语法没有区别。

示例

定义一个指针 pstr 分别指向 zhang3 和 li4 连个结构体类型的变量，然后使用指针访问其结构体内部的成员。

```
// filename: struct_pointer.c
#include <stdio.h>

struct student {
   char name[32];  // 姓名
   int age;  // 年龄
   float height; // 身高
};

int main(int argc, char * argv[]) {
    struct student zhang3 = {"zhang san", 18, 1.73};
    struct student li4 = {.age = 19, .height = 1.80, .name = "li si"};
    struct student * pstu = NULL;

    pstu = &zhang3;  // 将指针指向结构体变量 zhang3
    printf("姓名:%s,年龄:%d,身高:%.2f\n",(*pstu).name,(*pstu).age,(*pstu).height);

    pstu = &li4;  // 将指针指向结构体变量 li4
    printf("姓名:%s,年龄:%d,身高:%.2f\n",(*pstu).name,(*pstu).age,(*pstu).height);

    return 0;
}
```

运行结果如下

```
weimingze@mzstudio:~$ gcc -o struct_pointer struct_pointer.c
weimingze@mzstudio:~$ ./struct_pointer
姓名:zhang san,年龄:18,身高:1.73
姓名:li si,年龄:19,身高:1.80
```

可见我们使用指针 `pstu` 依旧可以访问结构体 `zhang3` 和 `li4` 内的成员变量。

上述示例中我们要先将**指针解引用** `*pstr` 然后再使用**成员访问运算符** `.` 来访问内部的成员。在上述示例中我们使用的 `*pstu` 解引用时要将解引用后加一个括号然后再用 `.` 运算符，这是因为 **成员访问运算符** `.` 的优先级比**指针解引用运算符** `*` 的优先级高。因此我们在使用结构体指针访问结构体成员变量时需要写成 `(*pstu).name` 而不是 `*pstu.name` 就是这个原因。

指针成员访问运算符 `->`

基于上述结构体指针访问结构体成员变量的应用场景，C 语言提供了**指针成员访问运算符** `->` 来将**指针解引用** `*` 后再用 成员访问运算符 `.` 的两步运算整合成一个运算符的 **指针成员访问运算符** `->`。

语法如下:

```
结构体指针->结构体成员变量名
```

即上述示例代码

```
printf("姓名:%s,年龄:%d,身高:%.2f\n",(*pstu).name,(*pstu).age,(*pstu).height);
```

可以改写成

```
printf("姓名:%s,年龄:%d,身高:%.2f\n",pstu->name,pstu->age,pstu->height);
```

效果完全一样。

改写后的示例如下:

```c
// filename: struct_pointer.c
#include <stdio.h>

struct student {
   char name[32];  // 姓名
   int age;  // 年龄
   float height; // 身高
};

int main(int argc, char * argv[]) {
    struct student zhang3 = {"zhang san", 18, 1.73};
    struct student li4 = {.age = 19, .height = 1.80, .name = "li si"};
    struct student * pstu = NULL;

    pstu = &zhang3;  // 将指针指向结构体变量 zhang3
    printf("姓名:%s,年龄:%d,身高:%.2f\n", pstu->name, pstu->age, pstu->height);
```

```
    pstu = &li4;  // 将指针指向结构体变量 li4
    printf("姓名:%s,年龄:%d,身高:%.2f\n", pstu->name, pstu->age, pstu->height);

    return 0;
}
```

其运行结果与上述示例完全相同。

实验

1. 改写上述示例程序，尝试使用指针对结构体进行访问。

3. 结构体赋值

这节课我们来学习 C 语言的结构体赋值。

前面我们学过两个相同数据类型的数组是不能够直接使用赋值语句修改数据内容的。但 C 语言的两个相同类型的结构体是可以直接赋值来修改数据的。

结构体赋值的方式通常有如下两种：

1. 逐个成员变量赋值
2. 结构体整体赋值

下面我们来举例说明上述两种赋值的方法。

现在我们定义两个 `struct student` 类型的结构体如下:

```
struct student {
   char name[32];  // 姓名
   int age;  // 年龄
   float height; // 身高
};
```

接下来我们创建两个结构体类型的变量。

```
struct student s1 = {"zhang san", 18, 1.73};
struct student s2;
```

1、 使用**逐个成员赋值**的方法来复制 `s1` 的值到 `s2` 中则需要如下代码实现。

```
strcpy(s2.name, s1.name);
s2.age = s1.age;
s2.height = s1.height;
```

2、 使用**结构体整体赋值**赋值的方法来复制 `s1` 的值到 `s2` 中 则需要如下代码实现。

```
s2 = s1;
```

可见结构体赋值整体赋值相对比较简单。

当使用 **结构体整体赋值** 时，由于大部分的处理器没有对应的指令能够实现整体赋值，因此整体赋值实际是整体内存拷贝，即 `s2 = s1;` 实际等同于 `memcpy(&s2, &s1, sizeof(s1));`。

函数的结构体传参

在函数调用的过程中，实际参数的值传递给形式参数时，实际就是赋值。因此在函数调用传递结构体时，通常使用**传址**的方式来代替**传值**，这样可以大大提高软件的执行效率。因为指针赋值的开销远远小于结构体赋值的开销。

在函数中使用传址的方式传递结构体数据时，如果函数内不改动结构体的数据，我们经常在形参的结构体指针中加入一个 `const` 修饰符。以避免在函数中错误的修改了结构体的数据。

示例:

```c
// filename: struct_assigment.c
#include <stdio.h>

struct student {
   char name[32];  // 姓名
   int age;   // 年龄
   float height; // 身高
};

// 使用传址的方式传递结构体，参数的复制的开销比较大
void print_stu_info1(struct student s) {
    printf("姓名: %s, 年龄: %d, 身高: %.2f\n", s.name, s.age, s.height);
}

// 使用传址的方式传递结构体，并添加 const 修饰符
void print_stu_info2(const struct student *ps) {
    printf("姓名: %s, 年龄: %d, 身高: %.2f\n", ps->name, ps->age, ps->height);
}

int main(int argc, char * argv[]) {
    struct student s1 = {"zhang san", 18, 1.73};

    print_stu_info1(s1);
    print_stu_info2(&s1);
```

```
    return 0;
}
```

运行结果如下：

```
weimingze@mzstudio:~$ gcc -o struct_assigment struct_assigment.c
weimingze@mzstudio:~$ ./struct_assigment
姓名: zhang san, 年龄: 18, 身高: 1.73
姓名: zhang san, 年龄: 18, 身高: 1.73
```

可见上述两个函数 `print_stu_info1` 和 `print_stu_info2` 都能实现打印结构体数据的功能，但 `print_stu_info2` 的传参效率会优于 `print_stu_info1`。这也是我们在查看很多的 C 语言源代码时发现很多人写的函数调用在传递结构体时多数都是使用**传址**代替**传值**的主要原因。

复合字面值（Compound literal）

在 C99 的 C 语言标准中，我们可以通过 **复合字面值** 来构建一个结构体类型的字面值，通常这个字面值用于整体赋值给某个结构体或函数的传参。

复合字面值的语法

```
(结构体类型){初始化列表}
```

复合字面值是一个表达式，它会返回一个结构体类型的临时对象。

示例：

改写上述程序，我们使用 **复合字面值** 为结构体变量赋值和传参。

```
// filename: struct_assigment.c
#include <stdio.h>

struct student {
   char name[32];  // 姓名
   int age;   // 年龄
   float height; // 身高
};

void print_stu_info1(struct student s) {
    printf("姓名: %s, 年龄: %d, 身高: %.2f\n", s.name, s.age, s.height);
}

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

```
    struct student s1;

    s1 = (struct student){"wang5", 12, 1.41};
    print_stu_info1(s1);

    s1 = (struct student){.name="zhao6", .height=1.80, .age=19};
    print_stu_info1(s1);

    print_stu_info1((struct student){"qian7", 20, 1.69});

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o struct_assigment struct_assigment.c
weimingze@mzstudio:~$ ./struct_assigment
姓名: wang5, 年龄: 12, 身高: 1.41
姓名: zhao6, 年龄: 19, 身高: 1.80
姓名: qian7, 年龄: 20, 身高: 1.69
```

可见，使用复合字面值也可以临时的创建一个 结构体类型的对象。

实验

- 使用你自己的编译器，实现上述代码的编写，看你的编译器是否支持复合字面值的语法。
- 思考：数组不能复制，为什么结构体内的数组就能够通过结构体整体赋值而进行复制?

4. 结构体数组

结构体类型是使用 `struct` 关键字声明的自定义数据类型，这种数据类型可以和 C 语言预置的数据类型一样来构成一维数组和多维数组。我们通常使用结构体数组来存储较为复杂的信息。

一维结构体数组定义个语法如下:

```
struct 结构体名 数组名[整数表达式n] = {{...}, {...}, ...};
```

说明:

1. 在数组的初始化列表中，最外层大括号的内部使用 `{...}` 的初始化列表类初始化数组中结构体的数据。

示例:

下面我们用一维结构体数组来存储 100 个学生信息。

```
// filename: struct_array.c
#include <stdio.h>

struct student {
   char name[32];  // 姓名
   int age;   // 年龄
   float height; // 身高
};
int main(int argc, char * argv[]) {
    struct student class1[100] = {
            {"zhang san", 18, 1.73},
            {.age = 19, .height = 1.80, .name = "li si"}
        };
    class1[2] = (struct student){"wang wu", 12, 1.41};

    for (int i = 0; i < 3; i++) {
        printf("姓名: %s, 年龄: %d, 身高: %.2f\n",
            class1[i].name, class1[i].age, class1[i].height);
    }

    return 0;
}
```

编译和运行结果如下

```
weimingze@mzstudio:~$ gcc -o struct_array struct_array.c
weimingze@mzstudio:~$ ./struct_array
姓名: zhang san, 年龄: 18, 身高: 1.73
姓名: li si, 年龄: 19, 身高: 1.80
姓名: wang wu, 年龄: 12, 身高: 1.41
```

可见，一维的结构体数组和一维内建数据类型的数组没有区别，同样我们也可以使用结构体创建二维及以上维度的数组。

练习：

使用上述结构体数组，读取 5 个学生信息，计算并打印出这 5 位学生的平均年龄。以及最高身高的学生信息。

5. 结构体字节对齐

这节课我们来学习 C 语言中的**结构体字节对齐**。

结构体字节对齐 是 C 语言编译器为了适用计算机硬件系统而做出的一种优化策略，目的是在有可能浪费部分内存空间的情况下来换取程序尽可能快的运行速度。

字节对齐方式通常是 1字节对齐、2字节对齐、4字节对齐、8字节对齐。对齐方式都是 2 的 n 次方的方式。

结构体字节对齐 根据编译器和硬件平台不同而不同。

先来说一下什么是字节对齐。我们先看下面这个结构体占用几个字节?

```
struct mydata {
    char c;
    short int s;
};
```

我来写代码测试一下:

```
// filename: struct_align.c
#include <stdio.h>

struct mydata {
    char c;
    short int s;
};

int main(int argc, char * argv[]) {
    struct mydata a;

    printf("sizeof(a.c): %ld\n", sizeof(a.c));
    printf("sizeof(a.s): %ld\n", sizeof(a.s));
    printf("sizeof(struct mydata): %ld\n", sizeof(struct mydata));
    printf("sizeof(a): %ld\n", sizeof(a));

    printf("&a: %p\n", &a);
    printf("&a.c: %p\n", &a.c);
    printf("&a.s: %p\n", &a.s);

    return 0;
}
```

运行结果如下:

```
sizeof(a.c): 1
sizeof(a.s): 2
sizeof(struct mydata): 4
sizeof(a): 4
&a: 0x7ffe8c2d93a4
```

```
&a.c: 0x7ffe8c2d93a4
&a.s: 0x7ffe8c2d93a6
```

根据打印结果我们绘制其结构体的内存结构如下：

```
struct mydata a;

    这个字节空着
         |
         V
+-----+-----+-----+-----+
| a.c |     |    a.s    |
+-----+-----+-----+-----+
^           ^
|           |
&a.c         &a.s
&a           0x7ffe8c2d93a6
0x7ffe8c2d93a4
```

可见结构体 `struct mydata` 一共有两个成员，其中 `c` 占 1 个字节，`s` 占 2 个字节，两个成员组成一个结构体后的变量 `a` 却是占用 4 个字节。其中有一个字节没有使用。这就是由编译器字节对齐引起的结果。

字节对齐

要理解字节对齐这个问题，我们要从计算机内存的结构说起。在早期，由于制造工艺的限制。我们在制作内存时规定一个字节都是由 8 个位组成的。构成内存的存储单元都是用一个字节（8 个位）构成。这时我们一个 short int 类型变量要放在两个存储单元中，每次读取这个 `short int` 类型的数据需要读取两次内存单元，然后组合在一起形成 16 位的 `short int` 数据。随着计算机技术的发展，内存的每个存储单元使用 8个 位就太小了。于是后来出现了 16 位内存、 32 位内存、甚至现在有些内存是 64 位内存。就是说在计算中，每一个存储单元用 64 个位构成，一个机器周期就可以一次性的读取这 64 个位的数据。将内存存储单元做大可以节省制造成本，又可以一次访问更多的数据位数，提高了效率。现代计算机很多都是 64 位 CPU，就是说它一次可以处理 64 位的内存数据。

下面用示意图的方式来说明内存位数和地址的关系：

1、8 位内存每个存储单元存储一个字节。存储单元的地址从 0 开始，每个地址 递增 1

```
+-----+-----+-----+-----+-----+-----+-----+-----+----
|     |     |     |     |     |     |     |     | ...
+-----+-----+-----+-----+-----+-----+-----+-----+----
0     1     2     3     4     5     6     7     8
```

2、16 位内存每个存储单元存储2个字节。存储单元的地址从 0 开始，每个地址 递增 2。

16 位内存每次存取的最小单位是 16 位，每次存取的地址都是 2 的 1 次方的整数倍

```
+-----+-----+-----+-----+-----+-----+-----+-----+----
|           |           |           |           | ...
+-----+-----+-----+-----+-----+-----+-----+-----+----
0           2           4           6           8
```

3、同理，32 位内存每个存储单元存储4个字节。存储单元的地址从 0 开始，每个地址 递增 4。

32 位内存每次存取的最小单位是 32 位，每次存取的地址都是 2 的 2 次方的整数倍

```
+-----+-----+-----+-----+-----+-----+-----+-----+----
|                       |                       | ...
+-----+-----+-----+-----+-----+-----+-----+-----+----
0                       4                       8
```

如何理解内存的储存单元呢？我们用建造车库做比喻。你是一个建筑设计师。计划建造能够停放 1000 辆小轿车的车库。于是你从村东头向村西头一共建造了 1000 间小房子，每个房间存放一辆小汽车。这样建造房子会有一个问题，如果我们车库内存放车辆长度比较长的小货车，它要占用两间房，这时候我们就要将小货车的车头和货箱分开后分别放入两间房子中，等取车时在组合在一起。虽然可行，但效率较低。有了这个经验。下一个项目还是要计划建造能够停放 1000 辆小轿车的车库，同样也可能存放小货车。聪明的你这一次建造了 500 间原来两倍大的房间，每个房间可以存放两辆小汽车或一辆小货车。房间号由原来的编号`0、1、2、3、...`改成了`0、2、4、6、...`。这样你的建造房间的数量少了，成本降低且达到了同样的效果。那有一个问题出现了。我们在第一个房间停放了一辆小汽车，这个房间还能再停放另外一辆小汽车，但此时我要停放一辆小货车。聪明的你不再拆车，而是让或小货车停在隔壁车库里。你让当前这个房间的一个车位空着就行了。

这就是字节对齐的原理。字节对齐可以提高数据存储的效率。

字节对齐的规则

每个变量的起始地址都是这个变量自身长度的整数倍，我们把这个数值称为该类型的**对齐字节数**。

- 如：`char` 为 1，`short` 为 2，`int` 为 4，`double` 为 8。

结构体整体对齐规则

1. **编译器默认的对齐字节数** 通常是 4 或 8。这个规则不同的编译器是不一样的。
2. **结构体对齐**先保证结构体内所有成员变量都对齐。

3. 整个结构体的对齐字节数必须是 **最大成员对齐字节数** 和 **编译器默认对齐字节数** 取最小值后的整数倍。
   - 编译器可能在结构体的末尾添加填充字节来满足此要求。
4. 结构体的起始地址也必须是此**结构体字节对齐字节数**的整数倍。

改写上述结构体如下：

```
struct mydata {
    char c;
    short int s;
    int i;
};
```

我们使用 `sizeof(struct myadata)` 来计算此结构体占用的字节数为 `8`，其对齐方式为 4 字节对齐。

我们打印各个变量的地址，然后来分析它的内存结构如下：

```
// filename: struct_align.c
#include <stdio.h>

struct mydata {
    char c;
    short int s;
    int i;
};

int main(int argc, char * argv[]) {
    struct mydata d;

    printf("&d: %p\n", &d);
    printf("&d.c: %p\n", &d.c);
    printf("&d.s: %p\n", &d.s);
    printf("&d.i: %p\n", &d.i);

    printf("sizeof(struct mydata): %ld\n", sizeof(struct mydata));

    return 0;
}
```

运行结果如下：

```
&d: 0x7ffdeaf78a00
&d.c: 0x7ffdeaf78a00
&d.s: 0x7ffdeaf78a02
&d.i: 0x7ffdeaf78a04
sizeof(struct mydata): 8
```

分析内存结构如下:


```
    struct mydata da;

      这个字节空着
           |
           V
  +-----+-----+-----+-----+-----+-----+-----+-----+
  | d.c |     |    d.s    |          d.i          |
  +-----+-----+-----+-----+-----+-----+-----+-----+
  ^           ^           ^
  |           |           |
 &d.c         &d.s        &d.i
 &d                       0x7ffdeaf78a04
0x7ffdeaf78a00
```

6. _Alignof 运算符

_Alignof 运算符 用于在编译阶段返回一个数据类型的对齐字节数（整数常量）。

在 C11 标准之前我们只能靠打印地址的方式来根据经验猜测编译器的对齐方式，在 C11 标准的编译器中提供了 `_Alignof` 运算符可以返回某种类型数据的对齐字节数。根据这个对齐字节数，我们可以优化程序和结构体字节对齐。

语法:

```
_Alignof(类型名)
// 或
_Alignof(表达式)
```

示例:

```
// filename: alignof.c
#include <stdio.h>

struct aaa {
    char c;
    double d;
    int i;
};

struct bbb {
    char c;
    int i;
};

int main(int argc, char * argv[]) {
    char c1;
    short int s1;
```

```c
    int i1;
    double d1;

    struct aaa a;
    struct bbb b;

    printf("_Alignof(c1): %ld\n", _Alignof(c1));
    printf("_Alignof(s1): %ld\n", _Alignof(s1));
    printf("_Alignof(i1): %ld\n", _Alignof(i1));
    printf("_Alignof(d1): %ld\n", _Alignof(d1));

    printf("_Alignof(char): %ld\n", _Alignof(char));
    printf("_Alignof(short int): %ld\n", _Alignof(short int));
    printf("_Alignof(int): %ld\n", _Alignof(int));
    printf("_Alignof(double): %ld\n", _Alignof(double));

    printf("_Alignof(struct aaa): %ld\n", _Alignof(struct aaa));
    printf("_Alignof(struct bbb): %ld\n", _Alignof(struct bbb));
    printf("_Alignof(a): %ld\n", _Alignof(a));
    printf("_Alignof(b): %ld\n", _Alignof(b));

    printf("sizeof(a): %ld\n", sizeof(a));  // 24，末尾添加填充字节

    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ gcc -o alignof alignof.c -std=c11
weimingze@mzstudio:~$ ./alignof
_Alignof(c1): 1
_Alignof(s1): 2
_Alignof(i1): 4
_Alignof(d1): 8
_Alignof(char): 1
_Alignof(short int): 2
_Alignof(int): 4
_Alignof(double): 8
_Alignof(struct aaa): 8
_Alignof(struct bbb): 4
_Alignof(a): 8
_Alignof(b): 4
sizeof(a): 24
```

说明：

- GCC 的编译选项 `-std=c11` 是使用 C11 标准编译。

实验：

分析如下结构体的内存结构。

```
struct mydata {
    char c;
    double d;
    short int s;
};
```

7. 字节对齐控制

上节课我们学习了结构体字节对齐，编译器为了优化程序，让程序更快的运行，它在编辑阶段可能在结构体内添加填充字节以让结构体内所有的成员变量都字节对齐。但这些填充字节有时会给我们带来不小的麻烦。比如我们将一个结构体内的数据以二进制方式发送到远程计算机或保存到文件中时，这些填充字节也会随之发送。我们浪费了流量和存储空间，同时为系统安全留下了隐患。

在 C 语言中 我们可以使用 `#pragma pack` 预处理指令来对编译器默认的字节对齐字节数进行修改。

`#pragma pack` 预处理指令

`#pragma pack` 预处理指令的作用是指示编译器以下代码以哪种对齐字节数进行对齐。

语法:

```
// 设置指定的对齐字节数
#pragma pack(n)

// 恢复编译器默认的对齐字节数
#pragma pack()
```

说明

1. 当使用 `#pragma pack(n)` 时，编译器默认对齐字节数将被替换为 `n`。
2. n 的值必须是 2 的 x 次方的形式，通常是：1、2、4、8 等。
3. 当使用 `#pragma pack()` 时，将恢复对齐字节数为编译器默认对齐字节数。

示例:

将以下结构体设置为 1 字节对齐

```
struct mydata {
    char c;
```



```
    short int s;
    int i;
};
```

示例代码如下:

```
// filename: pragma_pack.c
#include <stdio.h>

#pragma pack(1)  // 设置为 1 字节对齐
struct mydata {
    char c;
    short int s;
    int i;
};
#pragma pack()  //  恢复为默认对齐方式

int main(int argc, char * argv[]) {
    struct mydata d;

    printf("sizeof(d): %ld\n", sizeof(d));

    printf("&d.c: %p\n", &d.c);
    printf("&d.s: %p\n", &d.s);
    printf("&d.i: %p\n", &d.i);

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o pragma_pack pragma_pack.c
weimingze@mzstudio:~$ ./pragma_pack
sizeof(d): 7
&d.c: 0x7ffdd39b4421
&d.s: 0x7ffdd39b4422
&d.i: 0x7ffdd39b4424
weimingze@mzstudio:~$
```

根据上述结果分析结构体的内存结构如下:

```
    _____________ 共占用 7 字节 _____________
   /                                         \
  +-----+-----+-----+-----+-----+-----+-----+
  | d.c |    d.s    |          d.i          |
  +-----+-----+-----+-----+-----+-----+-----+
  ^     ^           ^
  |     |           |
 &d.c   &d.s        &d.i
 &d
```

GCC 编译器的 `__attribute__((packed))` 特性

`__attribute__((packed))` 特性 的作用同 `#pragma pack(1)` 一致，它用于结构体声明中，它可使当前声明的结构体取消字节对齐（即设置编译器对齐字节数是1）。

语法:

```
struct 结构体名 {
    数据类型 1成员变量名1;
    ...
} __attribute__((packed));
```

说明:

1. `__attribute__((packed))` 特性仅能用于 GCC 编译器中。
2. `__attribute__((packed))` 特性仅对当前声明的结构体有效。

示例：

上述结构体中

```
#pragma pack(8)  // 设置为 1 字节对齐
struct mydata {
    char c;
    short int s;
    int i;
};
#pragma pack()  //  恢复为默认对齐方式
```

在 GCC 编译器中可以替换成如下代码，但运行结果是一样的。

```
struct mydata {
    char c;
    short int s;
    int i;
}__attribute__((packed));
```

如果你安装的是 GCC 编译器，请自行尝试运行上述代码。

实验

已知结构体如下：

```
struct pack_demo {
    char a;
```

```
    int b;
    short c;
};
```

其尝试使用 1 字节对齐、2字节对齐、4字节对齐、8字节对齐和编译器默认的对齐方式来测试其结构体的内存结构。

8. _AlignAs 关键字

`_AlignAs` **关键字** 是 C11 标准引入的关键字，它的作用是显式指定一个变量或结构体成员变量在内存中的对齐方式。

`_AlignAs` **关键字** 可以强制一个变量、数组、或结构体以及结构体成员的起始地址以某个数值（通常是 2 的 x 次方的值，如:1, 2, 4, 8）的整数倍进行对齐。这种对齐通常是为了满足某种硬件需求或提高 CPU 的运行速度。

前面我们学过 `#pragma pack(n)` 这个预处理指令，这个预处理指令会在预处理阶段来压缩一个结构体内部的成员变量，使其按给定的对齐字节数进行排列，但这个指令只压缩结构体的对齐字节数，并不会扩大对齐字节数。

`_AlignAs` **关键字** 和 `#pragma pack(n)` 预处理指令不同，它是编译阶段的关键字。它不会压缩结构体成员的字节对齐数，它只会扩大某个变量字节对齐数（但不会缩小）。如默认 double 类型的变量 d 在 64 位系统中的起始地址一定是 8 的整数倍。我们通过 `_AlignAs` 关键字 可以让其变为 16 的整数倍、32 的整数倍，但不能让其起始地址变为 4 的整数倍或 2 的整数倍。

语法:

```
// 根据 数据类型名 的对齐数作为最小对齐字节数
_Alignas(数据类型名)
// 给定 对齐字节数
_Alignas(整数常量表达式)
```

说明

1. `_Alignas` 对齐属性不得用于声明说明，如：typedef、位域、函数、形式参数、以及使用 register 声明的变量或对象。
2. **整数常量表达式** 的计算值的应为有效的对齐字节数（即 2 的 x 次方的值，如:1, 2, 4, 8等）。
3. **整数常量表达式** 的计算值不得小于修饰变量的最小**对齐字节数**。

4. `_Alignas(数据类型名)` 等同于 `_Alignas(_Alignof(数据类型名))`。即根据数据类型的来计算对齐字节数。
5. 如果**整数常量表达式**的值为 0，即 `_Alignas(0)`，则此 `_Alignas` 关键字则不再有效（等同于没有对齐要求）。

示例：

```
// filename: alignas.c
#include <stdio.h>

struct mystruct {
    char c;
    _Alignas(4) short int s;
    int i;
} t;

int main(int argc, char * argv[]) {
    char c;  // 默认起始地址 1 字节对齐。
    short s;  // 默认起始地址 2 字节对齐。
    int i;  // 默认起始地址 4 字节对齐。
    double d;  // 默认起始地址 8 字节对齐。
    _Alignas(16) double a16d; // 起始地址修改为 16 的整数倍。
    _Alignas(256) char a256c; // 起始地址修改为 256 的整数倍。

    printf("&c: %p\n", &c);
    printf("&s: %p\n", &s);
    printf("&i: %p\n", &i);
    printf("&d: %p\n", &d);
    printf("&a16d: %p\n", &a16d);  // 地址的后 1 位一定是0
    printf("&a256c: %p\n", &a256c); // 地址的后 2 位一定是00


    printf("sizeof(t): %ld\n", sizeof(t));
    printf("_Alignof(t): %ld\n", _Alignof(t));
    printf("&t: %p\n", &t);
    printf("&t.c: %p\n", &t.c);
    printf("&t.s: %p\n", &t.s);
    printf("&t.i: %p\n", &t.i);

    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ gcc -o alignas alignas.c
weimingze@mzstudio:~$ ./alignas
&c: 0x7ffc86e71cd9
&s: 0x7ffc86e71cda
&i: 0x7ffc86e71cdc
&d: 0x7ffc86e71cf0
&a16d: 0x7ffc86e71ce0
```

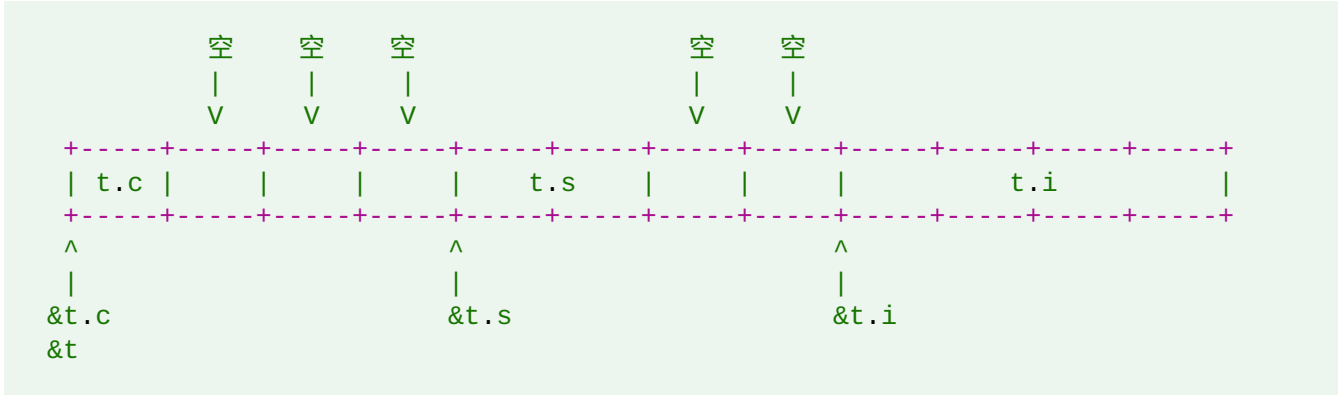
```
&a256c: 0x7ffc86e71c00
sizeof(t): 12
_Alignof(t): 4
&t: 0x60f5972f4018
&t.c: 0x60f5972f4018
&t.s: 0x60f5972f401c
&t.i: 0x60f5972f4020
```

从上述运行结果可知， 变量 `a16d` 的地址是 `0x7ffc86e71ce0` 是 16 的整数倍。变量 `a256c` 的地址是 `0x7ffc86e71c00` 是 256 的整数倍。

上述定义的结构体

```
struct mystruct {
    char c;
    _Alignas(4) short int s;
    int i;
} t;
```

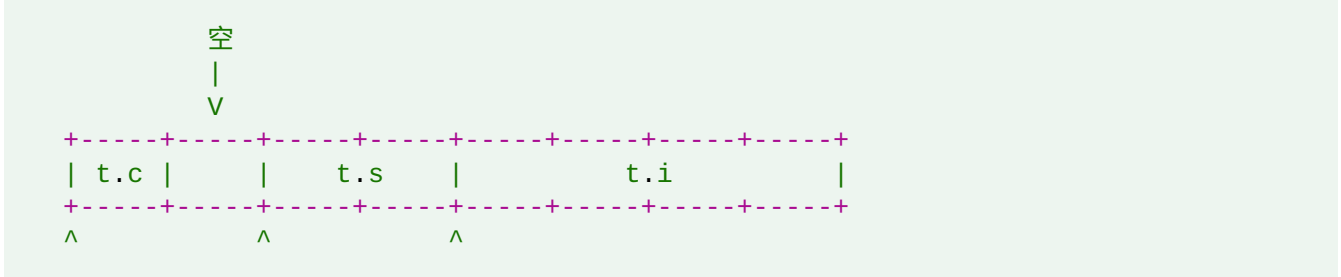
则 t 的内存结构是 如下 12 字节的内存结构。



如果 `strcut mystruct` 的声明改写如下

```
struct mystruct {
    char c;
    _Alignas(0) short int s;  // 等同于 short int s;
    int i;
} t;
```

则 t 的内存结构会变为 如下的 8 个字节的内存结构。




```
  |           |           |
 &t.c        &t.s        &t.i
 &t
```

实验

写程序，声明如下结构体变量的 ss, 通过打印地址的方式分析 ss 结构体在内存中的结构和占用字节数。

```
struct mystruct {
    char c;
    _Alignas(256) short int s;
    int i;
} ss;
```

9. 结构体嵌套声明

结构体嵌套声明是指结构体声明的内部可以再次声明结构体。其内部的结构体创建的变量可以作为外部结构体的成员变量。

下面我们声明一个结构体 `struct scr` 用来保存学生的语文成绩（chinese）、数学成绩（math）。

`struct score` 结构体声明如下：

```
struct scr {
    int chinese;
    int math;
};
```

下面我们有定义一个结构体 `struct stu` 用来保存学生的姓名（name）、年龄（age）和成绩（score）信息。

`struct stu` 结构体声明如下：

```
struct stu {
    char name[32];
    int age;
    struct scr score;
};
```


这样结构体 `struct stu` 就形成了一个内部嵌套结构体的结构体。我们把这种结构体的声明方式叫做**结构体嵌套声明**。

上述结构体声明我们还可以将 `struct scr` 的声明放到 `struct stu` 的内部。如下所示：

```
struct stu {
    char name[32];
    int age;
    struct scr {
        int chinese;
        int math;
    } score;
};
```

这样我们就将两个结构体类型声明整理成了一个结构体类型声明。

那如何使用结构体 `struct stu` 来定义变量呢?

使用嵌套声明的结构体定义变量和初始化的语法和没有嵌套声明的结构体是一致的。

示例：

使用嵌套上述嵌套的结构体来创建变量并初始化。

```
// filename: struct_nesting_decl.c
#include <stdio.h>

struct stu {
    char name[32];
    int age;
    struct scr {
        int chinese;
        int math;
    } score;
};

int main(int argc, char * argv[]) {
    struct stu s1 = {"zhang san", 18, {100, 90}};
    struct stu s2 = {"li si", 16, {.math=30, .chinese=70}};

    // 李四重新考试取得了新的数学成绩为 60
    s2.score.math = 60;
    printf("%d 岁的 %s 语文成绩: %d, 数学成绩: %d\n",
        s1.age, s1.name, s1.score.chinese, s1.score.math);
    printf("%d 岁的 %s 语文成绩: %d, 数学成绩: %d\n",
        s2.age, s2.name, s2.score.chinese, s2.score.math);

    return 0;
}
```

编译和运行结果如下:


```
weimingze@mzstudio:~$ gcc -o struct_nesting_decl struct_nesting_decl.c
weimingze@mzstudio:~$ ./struct_nesting_decl
18 岁的 zhang san 语文成绩: 100, 数学成绩: 90
16 岁的 li si 语文成绩: 70, 数学成绩: 60
```

可见嵌套声明的结构体和普通结构体没有区别。只是有了包含关系。

实验:

将上述结构体嵌套定义的代码

```
struct stu {
    char name[32];
    int age;
    struct scr {
        int chinese;
        int math;
    } score;
};
```

改写成两个结构体分别声明，如下所示。

```
struct scr {
    int chinese;
    int math;
};

struct stu {
    char name[32];
    int age;
    struct scr  score;
};
```

然后使用上面的示例运行，看运行结果是否会有变化。

分析上述两种写法的区别是什么?

10. 位域

位域（bit-fields） 也叫**位段**，是在一个结构体内部，使用 位(bit) 来划分边界，将划分出来的位重新定义成为结构体成员变量来代替位操作一种方式。位域是减少内存消耗，有效利用内存空间，提高开发效率的方法。

位域能够解决的问题:

在 C 语言中，如果使用一个 char 类型的变量来表示男（用1表示）和女（用0表示），则只用到了一个字节的最低位，而高位 7 位都是零。这样比较浪费内存空间。实际 一个 char 类型表示一个字节，由 8 个位组成。我们可以用 一个 char 类型来表示 8 个布尔值。或用一个 char 类型中的 1 个位表示男女，1个位表示是否已婚，1个位保留不用，2个位来表示是体重等级（过轻、正常、超重、严重超重），3个位表示信用等级。这样可以充分利用这段内存。

位域的语法

```
struct [结构体名] {
    数据类型 [成员变量名1] : 占用位宽1, [成员变量名2] : 占用位宽2, ...;
    // ... 结构体内其它成员或位域
} [变量名1][,变量名2, ...];
```

说明

1. **结构体名**必须是标识符。
2. **占用位宽**必须是整数常量表达式，表示成员变量占用的位数。
3. **成员变量名**是占用位数的别名，可以通过成员变量访问该成员变量占用的位。
4. **成员变量名**可以省略不写，省略不写时占用的位不可使用。
5. **数据类型** 是 该成员变量占用的位在引用时作为的数据类型，把这几位当成什么类型来处理。
   - 当数据类型是有符号类型时，占用位宽的高位会视为符号位（需要注意）。
6. 结构体可以在声明直接定义变量。也可以在声明后再定义变量。

示例：

```c
// filename: bit_field.c
#include <stdio.h>

struct human {
    unsigned char gender : 1, married: 1;
    unsigned char : 1;  // 该1位保留不用
    unsigned char  weight: 2, credit: 3;
};

int main(int argc, char * argv[]) {
    struct human h = {
            .gender = 1,  // 性别
            .married = 0,  // 是否已婚
            .weight = 2,  // 体重指数
            .credit = 5  // 信用评级
        };

    printf("sizeof(struct human): %ld\n", sizeof(struct human));
```

```
    printf("gender: %d\n", h.gender);
    printf("married: %d\n", h.married);
    printf("weight: %d\n", h.weight);
    printf("credit: %d\n", h.credit);

    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ gcc -o bit_field bit_field.c
weimingze@mzstudio:~$ ./bit_field
sizeof(struct human): 1
gender: 1
married: 0
weight: 2
credit: 5
```

可见上述结构体创建的变量只有一个字节，但可以当成 四个变量来用，每个成员变量使用这个字节中的某一位或某几位。

注意事项：

- 在大多是操作系统中位域操作还是会编译成处理器的位操作。因此不具备原子性（多线程操作可能出错）。

实验：

尝试使用位域和普通的结构体成员变量混合使用。然后测试你编译器是如何字节对齐的，并画出内存结构图。

第十七章、联合体

上一章我们学习了结构体。结构体是由多个成员变量按先后顺序在内存中依次向后摆放的数据类型。结构体内的成员变量默认按各自类型进行字节对齐。结构体变量本身也会按结构体的对齐字节数整体对齐。

这一章我们学习**联合体**数据类型，它是同结构体类型类似的数据类型，也可以认为它是一种特殊的结构体。

1. 联合体

联合体（Union），也叫共用体，它的允许你在同一块内存位置存储不同的数据类型。

联合体是一种和结构体一样的复合数据类型。它的内部同样可以定义多个成员变量，但这些成员变量的起始地址都是联合体变量的起始地址，即这些成员变量重叠的摆放在以联合体开始的同一内存地址中，共用同一内存的起始地址。

联合体的声明的语法和结构体声明的语法基本完全相同，只是联合体声明使用 `union` 关键字。

语法:

```
union [联合体名] {
    数据类型1  成员变量名1[: 占用位宽1], 成员变量名2[: 占用位宽2], ...;
    数据类型2  成员变量名4;
    // ... 联合体内其它成员变量
}[变量名1[={初始化列表1}]][,变量名2[={初始化列表2}], ...];
```

说明:

1. 语法中的中括号（`[]`）表示其中的内容可以省略。
2. **联合体名**必须是标识符。
3. `union` 是联合体声明的关键字，声明的联合体类型名为 `union 联合体名`。
4. **联合体名**可是省略不写，如果省略不写则必须在声明此联合体是定义变量。否则后续则无法为该联合体创建变量。
5. **成员变量名**必须是标识符，可以使用位域的形式。
6. 联合体的成员变量（包含位域）共用联合体变量的起始地址。
7. 前面 `union [联合体名] {...}` 部分是类型声明，是数据类型说明部分。

8. 后面 `[变量名1[={初始化列表1}]]` 是联合体变量声明，此部分可以省略不写。后续可以使用联合体类型来声明变量。
9. 联合体声明必须以分号（;）结束。

示例：

我们定义一个联合体 `union myunion` 并同时声明一个变量 `u`。

```
union myunion {
    char c;
    short int s;
    int i;
} u;
```

此时这个联合体变量 `u` 占用的内存长度是 4 个字节。其中成员变量 `c` 引用最低位的 1 个字节；成员变量 `s` 引用最低位的 2 个字节；成员变量 `i` 引用结构体的全部 4 个字节。 由于 成员变量 `c`、`s`、`i` 共用同一内存空间，当我们修改 成员变量 `c` 时，成员变量 `s` 和 `i` 的最低位的一个字节也会被同时修改。当我们修改成员变量 `s` 时，成员变量 `c` 也会被修改，成员变量 `i` 的最低位的 2 个字节也会被同时修改。如果成员变量 `i` 被修改，则成员变量 `c` 和 `s` 也会随之改变。因为它们共用同一内存空间。

上述联合体变量 u 的内存结构如下图所示：

```
  union myunion u;
  // u 占用的 4 字节内存如下
  +-----+-----+-----+-----+
  |     |     |     |     |
  +-----+-----+-----+-----+
  // u.c 在内存中的位置
  +-----+
  |     |
  +-----+
  // u.s 在内存中的位置
  +-----+-----+
  |     |     |
  +-----+-----+
  // u.i 在内存中的位置
  +-----+-----+-----+-----+
  |     |     |     |     |
  +-----+-----+-----+-----+
  ^
  |
  &u.c
  &u.s
  &u.i
  &u
  // 此四个字节的起始位置也是三个成员变量的起始位置
```

可见其中的第一个字节是三个成员都共用的字节。

我们在来看一下结构体的内存结构。

我们定义如下结构体。

```
struct mystruct {
    char c;
    short int s;
    int i;
} s;
```

则结构体成员变量 `c`、`s`、`i` 会按各自成员的对齐方式依次向后排列，不会叠加。

上述结构体变量 s 的内存结构如下图所示：

```
  struct mystruct s;
  // s 占用的 8 字节内存如下
  +-----+-----+-----+-----+-----+-----+-----+-----+
  |     |     |     |     |     |     |     |     |
  +-----+-----+-----+-----+-----+-----+-----+-----+
  // s.c 在内存中的位置
  +-----+
  |     |
  +-----+
  ^           // s.s 在内存中的位置
  |           +-----+-----+
  &s.c        |     |     |
  &s          +-----+-----+
              ^           // s.i 在内存中的位置
              |           +-----+-----+-----+-----+
              &s.s        |     |     |     |     |
                          +-----+-----+-----+-----+
                          ^
                          |
                          &s.i
```

可见，结构体和联合体的区别就是结构体的成员变量不会叠加，而且为提高内存读/写效率还会字节对齐。联合体的各个成员变量在内存中叠加（共用内存），且各个成员变量按联合体的起始地址对齐，而联合体变量按自己的对齐字节数整体对齐。

示例

```
// filename: union_demo.c
#include <stdio.h>

union myunion {
    char c;
    short int s;
```

```
    int i;
} u;

struct mystruct {
    char c;
    short int s;
    int i;
} s;

int main(int argc, char * argv[]) {
    printf("sizeof(u): %ld\n", sizeof(u));
    printf("sizeof(s): %ld\n", sizeof(s));
    printf("  &u: %p\n", &u);
    printf("&u.c: %p\n", &u.c);
    printf("&u.s: %p\n", &u.s);
    printf("&u.i: %p\n", &u.i);

    printf("  &s: %p\n", &s);
    printf("&s.c: %p\n", &s.c);
    printf("&s.s: %p\n", &s.s);
    printf("&s.i: %p\n", &s.i);

    u.i = 0x04030201;
    printf("u.c: 0x%08x\n", u.c);
    printf("u.s: 0x%08x\n", u.s);
    printf("u.i: 0x%08x\n", u.i);
    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ gcc -o union_demo union_demo.c
weimingze@mzstudio:~$ ./union_demo
sizeof(u): 4
sizeof(s): 8
  &u: 0x5f14a9032018
&u.c: 0x5f14a9032018
&u.s: 0x5f14a9032018
&u.i: 0x5f14a9032018
  &s: 0x5f14a9032020
&s.c: 0x5f14a9032020
&s.s: 0x5f14a9032022
&s.i: 0x5f14a9032024
u.c: 0x00000001
u.s: 0x00000201
u.i: 0x04030201
```

从运行结果可以看出，有相同个数和类型的成员变量的结构体和联合体占用的内存字节数是不一样的。联合体各个成员变量的内存起始地址是一样的，结构体中各个成员变量的内存起始地址则依次向后排列。

当我们修改 联合体中的一个成员变量。则内存改变。其它成员变量的值也会受到影响。

使用联合体的场景

在如下的两种情况我们通常会使用联合体：

1. 节省内存：在多个变量不同时使用的情况下，可以定义在一个联合体内，共用同一空间可以节省内存。
2. 以多种方式解析同一内存中的数据：在网络传输的字节串数据中，可以方便的解析出每个字节或某些字节对应的含义。

实验

使用联合体分析无符号整数、有符号整数、单精度浮点数、双精度浮点数的内存结构。我们定义如下联合体。

```
union {
    unsigned char b[8];
    unsigned int ui;
    int i;
    float f;
    double d;
} m;
```

我们使用赋值语句 `m.i = 100;`，然后打印 `b[0]`、`b[1]`、`b[2]`、`b[3]` 值就可以知道 `m.i` 在内存中的每一个位对应则值了。同理我们可以分别对 `m.ui`、`m.f`、`m.d` 赋值，也可看到内存中每一个字节的值对应的值。用此方法我们可以分析出各种数据类型在内存中的存储结构。

2. 字节序

字节序是指计算存储多字节数据（如：`short int`、`int`、`float`、`double`等）时，各个字节在内存中的排列顺序。

字节序分为两种：

1. 大端字节序（Big Endian）。
   - 低位字节存储在高地址，高位字节存储在低地址。
2. 小端字节序（Little Endian）。
   - 低位字节存储在低地址，高位字节存储在高地址。

通常字节序是 CPU 的运行模式决定的。现代计算机系统中，一般台式电脑，服务器等都使用小端字节序，而一些网络设备的路由器，交换机等的处理器通常使用大端字节序。

字节序在数据存储和数据通信领域尤为重要。一般网络通信协议中要规定数值的字节序（大端或小端二选一），否则可能会引起数据出错。

下面以四个字节的无符号整数为例来说明整数在不同字节序下的数据存放顺序。

如无符号整型变量 x 的定义如下：

```
unsigned int x = 0x04030201;
```

小端字节序存放方式：

```
+------+------+------+------+
| 0x01 | 0x02 | 0x03 | 0x04 |
+------+------+------+------+
低地址                    高地址
```

大端字节序存放方式：

```
+------+------+------+------+
| 0x04 | 0x03 | 0x02 | 0x01 |
+------+------+------+------+
低地址                    高地址
```

字节序检测方法

我们可以使用联合体对运行程序的字节序进行检测。

如上图所示的存放顺序，我们定义一个联合体，如下：

```
union byte_order {
    unsigned char buf[4];
    unsigned int x;
} bo;
```

我们将 整数 `0x04030201` 赋值给 `bo.x`，然后判断最低位字节 `bo.buf[0]`，如果 `bo.buf[0]` 的值是 `0x01` 则说明你的系统是小端字节序。如果是 `0x04` 则是大端字节序。

示例：

```
// filename: byte_order.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    union {
        unsigned char buf[4];
        unsigned int x;
```

```
    } bo;

    bo.x = 0x04030201;

    if (bo.buf[0] == 0x01) {
        printf("此电脑是小端字节序。\n");
    } else if (bo.buf[0] == 0x04) {
        printf("此电脑是小端字节序。\n");
    } else {
        printf("这是不可能出现的情况! \n");
    }

    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ gcc -o byte_order byte_order.c
weimingze@mzstudio:~$ ./byte_order
此电脑是小端字节序。
```

实验：

测试你的电脑是大端字节序还是小端字节序。

第十八章、枚举

无论任何的数据，在计算机内都要用数字来表示。比如在表示性别时，我们用 1 表示男性，用 0 表示女性；表示成功还是失败时我们用 0 表示成功，-1 表示失败等。当我们打开之前写的程序时我们可能会发现我们写过如下的代码：

```
result = 0;
```

请问这个 0 是表示成功还是失败呢？当初我们是怎么定义这个 0 的含义呢？

为了提高代码的可读性，提高编写代码的效率。我们通常用标识符来表示数值，常用的方法有两种：

1. 使用宏定义代替值，如: `#define SUCCESS (1)` 、`#define ERROR (0)`。
2. 使用枚举类型，如： `enum {ERROR = 0, SUCCESS = 1};`

当我们使用宏定义时。我们上述代码可以写成：

```
#define SUCCESS  (1)
#define ERROR    (0)

result = ERROR;
```

这样程序的可读性就好很多。

当让我们可以使用枚举类型，上面的代码依旧可以写成

```
enum {ERROR = 0, SUCCESS = 1};

result = ERROR;
```

以上两种写法都增强了代码的可读性。其中使用宏的方法是预处理阶段进行源码级提换，使用枚举则是在编译阶段进行数据处理。

这一章我们来学习 枚举类型。

1. 枚举

枚举（Enumeration） 是 C 语言中的一种基本数据类型，它主要用于定义一组标识符来代表整数常量。让代码更具可读性和可维护性。

语法:

```
enum [枚举类型名] {
    枚举常量名1[ = 整数常量表达式1],
    枚举常量名2[ = 整数常量表达式2],
    ...
} [变量名 [= 整数表达式]];
```

说明:

1. 枚举有自己的类型，类型名为 `enum 枚举类型名`。
2. **枚举类型名** 必须是标识符，可以省略不写，省略不写则只能在声明时直接定义变量，后续不能使用类型名定义该类型的变量。
3. **枚举常量** 必须是标识符，它代表一个整数常量值，只能取值不能赋值。
4. **枚举常量** 如果没有给出初始值，则它的值是上一个 **枚举常量** 的值加 1。
5. 第一个 **枚举常量** 如果没有给出初始值，则它的值是 0。
6. `enum [枚举类型名] {...}` 部分是类型声明，后面可以定义该类型的变量，也可以直接用分号（;）结束声明。

例如:

```
// 声明一个枚举类型 WeekDay 表示一周中的星期几。
enum WeekDay {
    Mon = 1,  // Mon 值为1
    Tue,  // Tue 值为 2
    Wed,  // Wed 值为 3
    Tru,  // Tru 值为 4
    Fri,  // Fri 值为 5
    Sat,  // Sat 值为 6
    Sun  // Sun 值为 7
};

// 声明一个枚举类型 ReturnState，同时定义一个变量 result。
enum ReturnState{
    DISABLE,  // ERROR 值为 0
    ENABLE = 2,  // ENABLE 值为 2
    OTHER_RESULT = -1  // OTHER_RESULT 值为 -1
} result;

// 声明两个枚举类型的常量值 ERROE 和 SUCCESS。
enum {
    ERROR,  // ERROR 值为 0
    SUCCESS = !ERROR,  // SUCCESS 值为 1
};

// 声明两个枚举类型的常量值 RESET 和 SET，同时定义一个变量 x, y;
```

```
enum {
    RESET = -1,  // RESET 值为 -1
    SET  // SET 值为 -1
} x, y;
```

使用枚举类型声明变量的语法:

```
enum 枚举类型名 变量名1 [= 整数表达式1][, 变量名1 [= 整数表达式1]]];
```

说明:

如果一个枚举类型在声明时没有给出类型名。则可以定义整数类型变量来保存其值。

示例

```
// filename: enumeration.c
#include <stdio.h>

// 声明一个枚举类型 WeekDay 表示一周中的星期几。
enum WeekDay {
    Mon = 1,  // Mon 值为1
    Tue,  // Tue 值为 2
    Wed,  // Wed 值为 3
    Tru,  // Tru 值为 4
    Fri,  // Fri 值为 5
    Sat,  // Sat 值为 6
    Sun  // Sun 值为 7
};

// 声明一个枚举类型 ReturnState，同时定义一个变量 result。
enum ReturnState{
    DISABLE,  // ERROR 值为 0
    ENABLE = 2,  // ENABLE 值为 2
    OTHER_RESULT = -1  // OTHER_RESULT 值为 -1
} result;

// 声明两个枚举类型的常量值 ERROE 和 SUCCESS。
enum {
    ERROR,  // ERROR 值为 0
    SUCCESS = !ERROR,  // SUCCESS 值为 1
};


// 声明两个枚举类型的常量值 RESET 和 SET，同时定义一个变量 x, y;
enum {
    RESET = 0,
    SET = !RESET
} x, y;
```

```c
int main(int argc, char * argv[]) {
    enum WeekDay theday;
    int ret = ERROR;
    if (ret != SUCCESS) {
        printf("result: %d\n", result);
    }
    x = SET;
    printf("x: %d\n", SET);
    printf("sizeof(x): %ld\n", sizeof(x));
    printf("sizeof(SET): %ld\n", sizeof(SET));

    for (theday = Mon; theday <= Sun; theday++) {
        printf("theday: %d\n", theday);
    }

    return 0;
}
```

编译和运行结果如下：

```
result: 0
x: 1
sizeof(x): 4
sizeof(SET): 4
theday: 1
theday: 2
theday: 3
theday: 4
theday: 5
theday: 6
theday: 7
```

从容上述运行结果可知，枚举类型的**枚举常量**可以当做整数常量来使用。

枚举类型的变量 和 枚举常量 的类型在不同的编译器下使用的内存长度是不一致的。大多数编译器将枚举类型看做是整数（四个字节），但也会有编译器将枚举类型的变量用一个字节保存。具体要看编译器和枚举常量的值来决定。

练习：

声明枚举类型 `Season` 来表示季节中的 `春`、`夏`、`秋`、`冬` 四个常量值（用标识符表示）。然后定义变量保存其中的某一个值。

第十九章、C 语言高级语法

本章我们来学习 C 语言中为了提高软件开发效率和运行效率等给出的一些语法规范。

1. 指针的类型总结

本节我们总结一下 C 语言中常用数据类型以及指向它们的指针的类型以及书写方法。

详见见如下表格

变量或函数	指向变量或函数的指针	说明
`char c;`	`char *pc = &c;`	`pc` 指向字符型数据
`short s;`	`short *ps = &s;`	`ps` 指向短整型数据
`int i;`	`int *pi = &i;`	`pi` 指向整型数据
`long int l;`	`long int *pl = &l;`	`pl` 指向长整型数据
`long long int ll;`	`long long int *pll = &ll;`	`pll` 指向长长整型数据
`float f;`	`float *pf = &f;`	`pf` 指向浮点型数据
`double d;`	`double *pd = &d;`	`pd` 指向双精度浮点型数据
`struct stu stu`	`struct stu * pstu`	`pstu` 是指向 结构体的指针
`union pkg g`	`union pkg *g`	`pg` 是指向联合体的指针
`enum color c`	`enum color *pc`	`pc` 是指向枚举的指针
`int *p;`	`int **pp = &p;`	`pp` 是指向整型指针的二级指针
`int a[100];`	`int(*pa)[100] = &a;`	`pa` 是指向指向整型一维数组的指针
`int ar[3][4];`	`int(*par)[3][4] = &ar;`	`par` 是指向指向整型二维数组的指针
`int* ap[100];`	`int*(*pap)[100] = &ap;`	`pap`是指向一维整型指针数组的指针
`void fn(void);`	`void(*pfn)(void) = &fn;`	`pfn` 是指向 `void(void)` 型函数的函数指针，表达式 &fn 等同于 fn
`int fn(int x, int y);`	`int(*pfn)(int, int) = &fn;`	`pfn` 是指向 `int(int,int)` 型函数的函数指针
`void fn(int x, int(*fx)(float));`	`void(*pfn)(int, int(*)(float)) = &fn;`	函数的参数是函数指针

`int *fn(int x);`	`int *(*pfn)(int);`	`pfn` 是指向返回值是指针的函数指针
`void (*fn(int x))(int);`	`void (*(*pfn)(int))(int) = &fn;`	返回值是函数指针函数的函数指针

> 最后一条看似很复杂，我们后面再解释。

上述我们分别声明了变量、数组和函数，也分别声明了指向变量、数组和函数的指针。此时你是否能否发现规律呢?

1. 声明指向变量的指针在变量名处前加一个 `*` 然后把变量名改成指针名。
2. 声明指向数组的指针在是将数组名用括号括起来，然后在数组名前加一个 `*` 然后把数组名改成指针名。
3. 声明指向函数的指针在是将函数名用括号括起来，然后在函数名前加一个 `*` 然后把函数名改成指针名（函数的形参名可以去掉，也可以保持不变）。

上述指针在解引用后得到的就是指向的数据对象。类型也和指向的对象类型相同。

指针的运算

基本数组类型的指针、指针类型的指针和指向数组的指针可以进行运算加、减运算。函数指针通常不进行加、减运算。

1. 基本数据类型的指针加 n 或减 n 则先前或向后移动一个n 个指向数据类型的长度。
2. 指向数组的指针也是如此，如： 指针 `int(*pa)[100]`，则 `pa+1` 则向后移动 400 个字节；指针 `int(*par)[3][4]`，则 `par+1` 则向后移动 48 个字节。

函数指针的使用

函数指针可以`(*函数指针)(实际调用参数)`的方式解引用后调用，也可以使用 `函数指针(实际调用参数)` 的方式直接调用。

返回函数指针的函数

我们来看 `void (*fn(int x))(int);` 这个声明。这个是声明一个函数 `fn(int x)`, 它有一个 形参变量 x，该函数返回值是一个 `void(*)(int)` 类型的函数指针。即该函数 fn 能够返回一个函数的地址。

如果想让返回函数指针的函数 写起来更容易理解，我们需要使用 C 语言的 typedef 关键字来为复杂类型起别名。

实验：

如下是 Linux/UNIX 操作系统信号设置函数的声明，分析该函数是怎样一个函数，该函数如何调用，实参是什么类型？返回值是什么类型？

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

2. typedef 类型别名

typedef 关键字用于为已有的数据类型取一个别名。

typedef 的主要作用是用简单的标识符来表示复杂的数据类型，方便程序编写和阅读。

使用 typedef 只是用原有类型映射出一个新的类型别名，但并不是创建一个新的类型。也就是说使用 typedef 声明的类型别名和原类型相同，没有区别，可以替换使用。

语法:

```
// 为基本数据取别名
typedef 旧类型 新类型别名;

// 在定义结构体、联合体、枚举时为结构体、联合体、枚举取别名
typedef [struct|union|enum] {
    // ... 成员变量列表
} 新类型别名;

// 在定义结构体、联合体、枚举后再取别名
typedef struct|union|enum 结构体|联合体名|枚举名 新类型别名;

// 定义一维数组类型别名
typedef 数组类型 新类型别名[整数表达式];

// 定义一维数组指针类型别名
typedef 数据类型(*新类型别名)[整数表达式];

// 定义二维数组指针类型别名
typedef 数据类型(*新类型别名)[整数表达式][整数表达式];

// 定义函数指针类型别名
typedef 返回值数据类型 (*新类型别名)(形参1类型, 形参1类型,...);
```

说明

- 中括号（[]）表示其中其中的内容可以省略。

- 竖线（|）表示所有选项必选其一。

示例

```c
// filename: typedef_demo.c
#include <stdio.h>

// 声明各种类型的别名， 标识符后加 _t 表示这个标识符是类型。
typedef unsigned char uint8_t;
typedef unsigned short int uint16_t;
typedef unsigned int uint32_t;
typedef unsigned long int uint64_t;

// 声明字符串型指针类型的别名
typedef const char * cstring_t;

// 定义结构体别名
typedef struct student {
    char name[32];
    int age;
} student_t;
typedef struct student stu_t;

// 定义枚举类型别名
typedef enum weekday {
    mon,
    tue,
} weekday_t;

// 定义 含有 10 个数据元素的整型一维数组
typedef int arr10_t[4] ;

// 定义 含有 10 个数据元素的整型一维数组
typedef int (*parr10_t)[4] ;

// 定义 void(int) 类型的函数的函数指针的类型别名
typedef void (*sighandler_t)(int);

void print_x(int x) {
    printf("the number is %d\n", x);
}

int main(int argc, char * argv[]) {
    // 上述类型别名的使用方法如下
    uint32_t x = 100;
    cstring_t name = "weimingze";
    student_t stu1 = {"zhang3", 18};
    stu_t stu2 = stu1;
    weekday_t today = mon;
    arr10_t scores = {70, 80, 90};
    parr10_t pscores = &scores;
    sighandler_t pfun = &print_x;

    // 使用上述数据
```

```
    printf("x: %d\n", x);
    printf("name: %s\n", name);
    printf("stu1.name: %s\n", stu1.name);
    printf("stu2.age: %d\n", stu2.age);
    printf("today: %d\n", today);
    for (x = 0; x < sizeof(scores)/sizeof(scores[0]); x++)
        printf("scores[%d]: %d\n", x, scores[x]);
    for (x = 0; x < sizeof(scores)/sizeof(scores[0]); x++)
        printf("(*pscores)[%d]: %d\n", x, (*pscores)[x]);

    pfun(200);
    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o typedef_demo typedef_demo.c
weimingze@mzstudio:~$ ./typedef_demo
x: 100
name: weimingze
stu1.name: zhang3
stu2.age: 18
today: 0
scores[0]: 70
scores[1]: 80
scores[2]: 90
scores[3]: 0
(*pscores)[0]: 70
(*pscores)[1]: 80
(*pscores)[2]: 90
(*pscores)[3]: 0
the number is 200
```

通过上述运行结果可以看出，使用 `typedef` 关键字仅使用一个标识符就可以代替复杂的类型名，使用该类型声明变量也可以写成 `类型别名 变量名` 的方式，用起来十分方便。

练习

函数声明如下:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

请问 函数 `signal` 的形参参数变量和返回值的类型都是什么？如果不使用 `typedef void (*sighandler_t)(int);` 声明函数的类型别名，这个函数 `signal` 函数的声明应当如何来写？尝试写出此函数 `signal` 的函数声明。

3. volatile 关键字

volatile 关键字的作用是指示编译器 volatile 修饰的变量可能会被程序之外的设备所修改，不允许优化掉。

volatile 关键字在 嵌入式系统中常用。

volatile 的语法

```
volatile 数据类型 变量名;
```

说明

1. volatile 关键字和 const 关键字都是类型修饰符，其用法和 const 也基本一致。

示例

我们来分析如下程序的运行结果。

```c
#include <stdio.h>

int main(int argc, char * argv[]) {
    const int x = 100;
    int y = x;
    int z = y;
    printf("z:%d\n", z);
    return 0;
}
```

在编译器优化后可能如下:

```c
#include <stdio.h>

int main(int argc, char * argv[]) {
    // ... 此处的代码都被优化掉了
    printf("z:%d\n", 100);
    return 0;
}
```

如果不希望将变量 y 进行优化，可以写成

```c
#include <stdio.h>

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

```
    const int x = 100;
    volatile int y = x; // 禁止编译器优化y这个变量。
    int z = y;
    printf("z:%d\n", z);
    return 0;
}
```

这样至少变量 y 不会被优化，至于变量 x 和 z 是否被优化，这个将由编译器和优化级别来决定。

使用 volatile 时，如下的用法是合理的。

```
// 以下两种写法效果相同
volatile int x;
int volatile x;

// 指针不可优化，指针指向的内容也不可优化
volatile int * volatile px;
```

实验

尝试使用 `volatile` 修饰变量，看你的编译器是否支持 `volatile` 关键字。

4. inline 内联函数

函数是 C 语言中重要的编写功能的单位。

在函数调用的过程中，函数的实际调用参数会赋值给函数的形式参数，函数的返回地址也会压栈到栈内存中，函数的返回值也会传递会调用处。这无疑是一笔运行时开销。尤其是函数调用以值传递的方式传递结构体或联合体时，这个开销就会更大，尤其是这个函数的调用频率特别高，比如每秒成百上千次时，这个开销可能难以承受。这时为了减小函数的调用开销。我们通常将函数内部的代码直接写到调用该函数的地方，这样就避免了调用的传参开销。

C 语言的 C99 标准引入了**内联函数**的概念，它是将一个比较短小的函数，在调用阶段，将函数体部分插入到调用的地方，来避免函数调用的开销。

内联函数是指有可能被优化掉，被放入到调用表达式处的函数。

C 99 的标准中可以使用 `inline` 关键字来声明内联函数。

inline 关键字 用于函数标识符的声明，它是建议编译器，使用 `inline` 关键字声明的函数尽可能使用内联编译。

语法


```
inline 返回类型 函数名(形式参数变量列表){ 函数体内的语句 };
```

说明:

- inline 关键字需要写在函数定义处。
- 内联函数要尽可能的短小，否则编译器可能会放弃内联编译。

示例

比如有如下内联函数 myadd。

```c
#include <stdio.h>

// 声明 myadd 函数是内联函数
inline int myadd(int x, int y) {
    return x + y;
}

int main(int argc, char * argv[]) {
    int result;

    result = myadd(100, 200);
    printf("result: %d\n", result);
    return 0;
}
```

其编译后的结果可能是如下程序。

```c
#include <stdio.h>

int main(int argc, char * argv[]) {
    int result;

    result = 100 + 200;
    printf("result: %d\n", result);
    return 0;
}
```

再次经过编译器优化后可能是如下程序。

```c
#include <stdio.h>

int main(int argc, char * argv[]) {
    printf("result: %d\n", 300);
    return 0;
}
```

内联函数的优缺点

优点

- 减少函数调用的开销，对小函数调用尤为有效。

缺点

- 如果多次调用可能会引起代码膨胀，比如一个内联函数内有 10 行代码，你在程序中有 100 次调用，那这 10 行代码会复制一百次到调用的地方，引起编译后的可执行程序变大，运行时代码段也会变大。

实验：

尝试完成上述示例的代码编写，看你的编译器是否支持 `inline` 关键字。

5. _Noreturn 关键字

`_Noreturn` 关键字是 C11 标准引入的一个函数说明符，它的作用是告诉编译器这个函数永远不会返回到它的调用表达式的地方。也就是说，使用此函数一旦被调用，就会以某种方式终止程序的执行，而不会返回到调用的地方继续执行。

`_Noreturn` 关键字 是函数说明符，用于函数标识符的声明。

语法

```
_Noreturn 返回类型 函数名(形式参数变量列表){ 函数体内的语句 };
```

说明:

1. **`_Noreturn` 关键字** 只能用于 `C11` 标准的编译器中。
2. C99 及之前的编译器对不返回的函数编译可能让会报告警告，但不会运行时出错。

示例：

写一个函数 input_numbers，输入一些列整数，此函数输入 0 时退出程序，但不会返回到调用此函数的地方。

```
// filename: line_func.c
#include <stdio.h>
#include <stdlib.h>
```



```c
_Noreturn void input_numbers(void) {
    int number;
    while(1) {
        printf("请输入一个整数，输入0结束输入：");
        scanf("%d", &number);
        printf("您输入的是：%d\n", number);
        if (0 == number) {
            printf("程序退出!\n");
            exit(0);  // 退出当前进程。
        }
    }
}

int main(int argc, char * argv[]) {
    input_numbers();
    printf("main 函数结束!\n"); // 此语句永远不会执行
    return 0;
}
```

实验：

用你的编译器尝试运行上述代码。

6. register 关键字

register 关键字是声明一个局部变量或一个形式参数变量为**寄存器变量**，这个变量在编译时可能会独占一个 CPU 的通用寄存器，以提高计算机的运行速读。

语法：

```c
register 数据类型 变量名1, 变量名2, ...;
```

说明：

1. 只有局部变量和形式参数变量可以声明为寄存器变量，全局变量和和静态局部变量不能声明为寄存器变量。
2. 由于 CPU 内的通用寄存器数量有限，因此在函数内不能声明超过 CPU 通用寄存器的个数的寄存器变量。
3. 声明过多的寄存器变量，可能会因为可用的通用寄存器过少致使程序运行缓慢。
4. 只有使用频率特别高的变量才有必要声明为寄存器变量。

示例：


```c
// filename: register.c
#include <stdio.h>

// 求 n 的阶乘 n!
int factorial(register int n) {
    register int result = 1;
    if (n < 1)
        return 0;
    while(n > 1) {
        result *= n;
        n--;
    }
    return result;
}

int main(int argc, char * argv[]) {
    printf("3!: %d\n", factorial(3));
    printf("5!: %d\n", factorial(5));

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o register register.c
weimingze@mzstudio:~$ ./register
3!: 6
5!: 120
```

实验

尝试使用你的编译器运行上述示例程序。

7. auto 关键字

auto 关键字 用于显示的声明一个函数内或复合语句内的变量为自动存储期的变量。

这里有一个**存储期**的概念。

存储期是指一个变量在内存中的生命周期。在 C99 标准和之前标准有三种存储期：

1. 静态存储期：是指全局变量或静态局部变量，它是从程序运行时开始，程序运行后销毁。
2. 自动存储期：是指函数内非静态局部变量、复合语句内部声明的非静态变量，这些变量在进入作用域内创建，退出作用域后自动销毁。

3. 动态存储期：由调用系统动态分配内存函数 `malloc` 等从操作系统中申请的内存。这的内存可以用 `free` 函数动态释放。

在 C11 的标准中又规定了线程存储期。

- 线程存储期：这些变量在线程启动时开始，线程结束后销毁。

默认一个局部变量如果没有声明为静态变量，则该变量默认就是自动存储期的变量，即自带 auto 属性。

语法

```
auto 数据类型 变量名1, 变量名2
```

说明：

1. 不能使用 auto 关键字修饰全局变量。
2. 局部变量如果使用 `static` 声明为静态局部变量，则不能再使用 `auto` 修饰为自动变量。

示例：

```c
// filename: auto.c
#include <stdio.h>

// 求 n 的阶乘 n!
int factorial(int n) {
    auto int result = 1;  // 显式声明自动变量
    if (n < 1)
        return 0;
    while(n > 1) {
        result *= n;
        n--;
    }
    return result;
}

int main(int argc, char * argv[]) {
    printf("3!: %d\n", factorial(3));
    printf("5!: %d\n", factorial(5));

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o register register.c
weimingze@mzstudio:~$ ./register
```

```
3!: 6
5!: 120
```

需要注意的是，在 C++ 的 C++11 标准中， auto 关键字的含义已经重新定义， auto 不再是 自动变量的修饰符。而是自动类型推断的关键字。

实验：

1. 将全局变量声明为自动类型变量，然后编译时查看编译器的信息提示，然后看提示是什么意思?
2. 将局部变量同时使用 `static` 和 `auto` 修饰，然后编译时查看编译器的错误提示信息，然后看提示是什么意思?

8. restrict 关键字

restrict 关键字 是 C 语言的 C99 标准中引入的一个关键字，它只能修饰指针的关键字。它用于向编译器提供承诺，此指针只能由这一个指针来操作这段内存，没有其它指针来指向这段内存。以便编译器使用更有力的优化策略对这段代码进行优化。

语法

```
数据类型 * restrict 指针变量名;
```

示例：

例如在 Linux系统下的 C 语言库函数中的 strcat 函数的定义如下：

```
char *strcat(char *restrict dst, const char *restrict src);
```

我们编写如下程序:

```
// filename: restrict.c
#include <stdio.h>
#include <string.h>

int main(int argc, char * argv[]) {
    char str1[100] = "hello ";
    char str2[100] = "world!";

      // 不会有restrict 警告
    strcat(str1, str2);
    printf("str1: %s\n", str1);
```



```
    // 以下 GCC 编译器会有restrict 警告 结果也会不确定。
    strcat(str1, str1);   // strcat 的形式参数的两个指针指向和同一个地址。
    printf("str1: %s\n", str1);

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o restrict restrict.c -Wall
restrict.c: In function ‘main’:
restrict.c:14:12: warning: passing argument 1 to ‘restrict’-qualified parameter
aliases with argument 2 [-Wrestrict]
   14 |     strcat(str1, str1);   // strcat 的形式参数的两个指针指向和同一个地址。
      |            ^~~~  ~~~~
weimingze@mzstudio:~$ ./restrict
str1: hello world!
str1: hello world!hello world!
```

重上面示例可以看出。由于 strcat 进行了优化。我们使用 `strcat(str1, str2);` 进行操作相对是比较安全的，因为两块内存是不同的区域。当我们使用 `strcat(str1, str1);` 对同一个地址的数据进行复制时，编译器会给出警告。

注意事项

1. restrict 是程序编写人员给编译器的承诺。编译器并不会检查此指针是否有别名。
2. 违反 restrict 的承诺，程序执行结果可能是未定义行为。

第二十章、动态内存管理

动态内存管理是在程序运行时（而非编译时）根据程序的运行情况手动申请和使用所需要的内存空间来存储数据，在使用完毕后有可以释放这段内存的机制。

动态内存管理是在堆地址空间内分配内存，在有操作系统的程序中，堆地址空间的内存通常由操作系统统一管理供各个进程使用。在没有操作系统的嵌入式系统中，通常需要自己写代码来管理堆地址空间的内存。

我们先来总结一下 C 语言数据存储的方法和声明周期。

1. 全局变量和静态局部变量，它们通常存储在数据段，它们的声明周期是在进程启动后一直存在，在进程退出后销毁。它们属于静态存储期。
2. 在函数内部定义的局部变量，它们通常在栈上自动分配。它们在函数调用时才分配内存空间，在函数结束后自动销毁。它们属于自动存储期。
3. 根据实际需要，使用 malloc 系列函数在堆地址空间内分配的内存，这个空间需要使用 `free` 函数手动释放。它们属于动态存储期。

有些程序在运行时才能确定具体需要的内存数量，并且这个内存数量往往比较大，不适合于在栈和数据段来分配内存，此时则可以使用堆上分配的内存。

说明：

栈的空间通常在进程启动后就已经确定，通常在 Linux 操作系统中，默认的栈空间是 8 兆字节。在很多的嵌入式系统中，栈空间可能只有几百字节到几千字节不等（实际可以软件控制）。因此使用栈空间来存储较大数据可能会带来系统的不确定性。通常在运行时需要较大的内存，我们常在堆上来动态分配内存。动态分配的内存需要使用指针来进行管理、使用和释放。

1. 动态内存分配和释放

C 语言的标准库提供了三个常用的动态内存分配的函数 `malloc`、`calloc` 和 `realloc` 函数和一个用于释放动态分配的内存的函数 `free`。

这些个函数如下表所示：

函数	说明
`void *malloc(size_t size);`	申请 `size` 个字节的连续的内存。
`void *calloc(size_t nmemb, size_t size);`	申请 `nmemb` 个 `size` 个字节的连续的内存(共`nmemb` ✖ `size`个字节)，并将内存中每个字节初始化为 0。
`void *realloc(void * ptr, size_t size);`	重新分配 `ptr` 指向的内存，将其改为 `size` 个字节，如果 `size` 小于之前的内存空间，则新内存的内容不变，如果 `size` 大于之前内存空间，则新增的内存空间不会被初始化。如果 `ptr` 为 `NULL` 则等同于 `malloc(size)`。
`void free(void * ptr);`	释放 `ptr` 指向的内存，还回给系统。

说明：

1. 使用这些函数是需要包含头文件 `stdlib.h`。
2. `malloc`、`calloc` 和 `realloc` 函数时，如果内存分配成功则返回内存的起始地址，失败返回 `NULL`。
3. 内存分配失败的原因通常是没有足够的连续的内存空间可供分配。
4. 上述三个函数返回内存的起始地址，类型是 `void*` 类型。在使用时需要强制类型转换到其它类型才能方便使用。
5. 上述 `malloc`、`calloc` 和 `realloc` 分配的内存空间必须使用指针记录，并在使用完毕后必须由 free 函数释放，否则会引起内存丢失等问题，严重会引起宕机。
6. free 函数的参数 `ptr` 必须是使用上述三个函数返回的内存地址，且每个地址只能释放一次。

示例：

动态在堆地址空间内分配 一段内存来存储 n 个学生的信息。然后读取 n 个学生的信息并打印。然后释放这段内存空间。

```c
// filename: dynamic_alloc.c
#include <stdio.h>
#include <stdlib.h>

struct student {
    char name[32];
    int age;
};

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

```c
    int n = 0;  // 用来记录学生个数。
    int i; // 循环变量
    struct student * all_stu = NULL;  // 用于记录学生数据的指针
    printf("请输入学生个数: ");
    scanf("%d", &n);

    all_stu = malloc(n * sizeof(struct student));
    // 上述语句也可以写成
    // all_stu = cmalloc(n, sizeof(struct student));
    if  (NULL == all_stu) {
        printf("内存空间不足，分配内存失败!");
        return 1;
    }
    // 分配内存成功
    for (i = 0; i < n; i++) {
        printf("请输入学生姓名: ");
        scanf("%s", all_stu[i].name);
        printf("请输入学生年龄: ");
        scanf("%d", &all_stu[i].age);
    }
    // 打印上述输入的信息
    for (i = 0; i < n; i++) {
        printf("%s 今年 %d 岁。\n", all_stu[i].name, all_stu[i].age);
    }
    // 释放内存
    free(all_stu);

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o dynamic_alloc dynamic_alloc.c
weimingze@mzstudio:~$ ./dynamic_alloc
请输入学生个数: 2
请输入学生姓名: zhangsan
请输入学生年龄: 18
请输入学生姓名: lisi
请输入学生年龄: 19
zhangsan 今年 18 岁。
lisi 今年 19 岁。
```

实验:

做一个具有破坏性的程序。在做此实验前请先保存你的所有文档。建议使用不用电脑或虚拟机进行实验。

在实验前，请提前打开**任务管理器**（Windows系统）、**活动监视器**（Mac OS 系统）或**系统监视器**（Ubuntu Linux 系统）等能够看到内存使用情况的工具软件。

写一个死循环，每次分为一定的内存空间，然后写入一些内容。当然每次内存并不释放。直至计算机内存耗尽而死机。

当然你也可以有限次数的循环来代替死循环来控制内存分配的最大值。

如:

```
while(1) {
    volatile int *p = NULL;
    p = (int*)malloc(sizeof(int));
    *p = 1;
}
```

2. 内存碎片问题

内存碎片（Memory fragmentation）是指在使用动态内存分配和释放时内存空间被分割成大量不连续的小块，导致虽然总空闲内存足够，但无法分配连续大块内存的现象。

如下图所示

现在有 8 个字节连续的空间，可以存储 8个 1 字节数据和 4 个两个字节的数据。现在我们存储 6 个 1 字节数据 和 1 个 2 字节数据如下:

```
   1     2     3     4        5        6     7
+-----+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |           |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

现在程序运行，释放了编号为 `2`、`4`、`6` 的内存，结果如下:

```
   1           3              5              7
+-----+     +-----+     +-----+-----+     +-----+
|     |     |     |     |           |     |     |
+-----+     +-----+     +-----+-----+     +-----+
```

现在内存空余 3 个字节，但因为不连续，无法形成一个 2 字节的连续内存。因此分配 2 字节的内存空间会失败。

要解决这一问题需要软件开发人员进行规避。比如无论需要多大的内存都按最大的内存块进行分配空间。这样虽然浪费内存空间，但可以有效解决内存碎片问题。

当然还有很对软件的解决方法，请各位朋友自行搜索研究。

第二十一章、文件操作

1. 文件概述

文件是将计算机运行时的数据进行归档的一种方式，它将计算机内运行的数据以字节为单位有序的存储于文件系统中，供后续使用。这个文件系统可以存在于硬盘、优盘甚至是内存中。

文件都是以字节为单位，多个字节顺序存储的。通常一组数据保存为一个文件并且有一个名称，称为**文件名**。

文件通常存储在文件系统中的某个文件夹中。从一个文件系统的起点，到达这个文件所经历的一系列文件夹名称为路径，用字符串表示。

路径

在 Windows 操作系统中，文件路径通常都是以 **C:**、**D:** 这样的磁盘的盘符开始。在 Moc OS 或 Linux 操作系统中，文件路径通常是以反斜杠 **/**（根文件夹）开始的。

如:

Windows 下的文件路径示例:

```
C:\Windows\regedit.exe
```

Mac OS、Linux 下的文件路径示例:

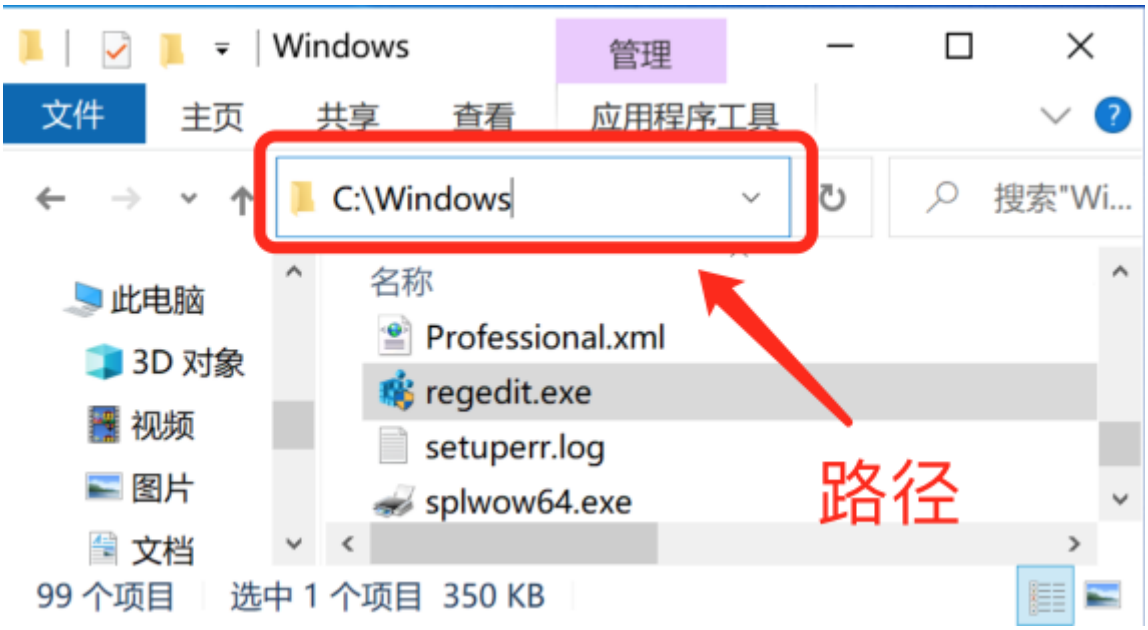
```
/etc/passwd
```

说明:

在路径表示的字符串中，文件夹和文件夹之间用一个符号来分隔。我们把这个叫做**路径分隔符**。

1. Windows 下路径的路径分隔符是**反斜杠**（`\`）。
2. Mac OS、Linux 下路径的路径分隔符是**斜杠**（`/`）。

如在 Windows 文件浏览器中的路径显示位置和路径表示方式如下图所示



绝对路径和相对路径

我们把从文件系统的起点开始的路径称为**绝对路径**。绝对路径是访问文件所必须的路径。

在现实编程开发的过程中，有些应用程序安装的位置不同，比如在 Windows 上安装应用程序时，有些人喜欢安装在 C 盘。有些喜欢安装在 D 盘。这就给程序访问所需要的文件带来了麻烦。因此我们常使用**相对路径**。

相对路径通常是一个不从文件系统起点的开始的路径。这个路径可以根据当前程序的工作路径动态的算出来绝对路径。这样的路径称为**相对路径**。如上图中，相对路径可以写成 `regedit.exe` 或 `.\regedit.exe` 它的当前工作路径是 `C:\Windows`，因此拼接以后 `C:\Windows\regedit.exe` 就是 `regedit.exe` 这个文件的绝对路径了。

在使用 C 语言对文件进行操作的方式有两种：

1. 读：从文件中获取数据到内存中。
2. 写：将内存中的数据保存到文件中，我们把这个过程称之为**序列化**。

在将计算机内存中的数据写入到文件中时，我们要人为规定文件中哪个字节或哪几个字节存放什么数据，这个规定称为**文件格式**。规定文件格式是保证数据能够顺利读写的基础。

对任何文件的完整操作都要经过如下三步。

文件操作的步骤

1. 打开文件

2. 读、写文件
3. 关闭文件

打开文件是设备上电、操作系统为该文件分配缓存的过程。关闭文件则是清空缓存、释放缓存和设备断电的过程。关闭文件可以释放此文件占用的系统资源。因此一个完整的文件操作，打开文件和关闭文件要成对出现。否则可能会影响系统的稳定性。读文件是将文件中的内容复制到内存并解释的过程，写文件是将内存中的数据放入到文件中的过程。

文件都是以字节为单位进行存储的。为了开发人员操作方便，C 语言的标准库函数提供了两套应用程序接口（Application Programming Interface，简称 API）供开发人员使用。这两套接口分别是**二进制文件操作**和**文本文件操作**。

二进制文件操作就是将计算机的内存中的某段数据和文件中的某段位置的数据直接进行复制，不进行解析。比如一个整数 `123456` 在内存中占用 4 个字节，直接放入文件中也是 4 个字节，且字节序也完全一致。

文本文件操作就是将计算机内存中的数据解析成字符串，然后将字符串对应的内存中的内容整体保存到文件中。比如一个整数 `123456` ，将其转为字符串则为 "123456"，在内存中实际占用 6 个字节然后把这 6 个字节放入文件中。使用文本文件操作的好处是使用文本文件编辑器（如 Windows 上的记事本和各种文本文件编辑器）可以直接打开阅读和编辑。缺点是在程序读写数据时需要将文件内容中的字符串和实际内存中的数据格式进行转换。有效率上的损失。

文本文件操作 操作通常以行为单位进行操作，一般行的结束符是 `"\n"`（Linux/UNIX常用）或 `"\r\n"`（Windows 常用）。

文本文件操作 时，每一行数据中表示的内容通常需要规定其分隔符，否则可能无法划分数据的边界，导致读取错误。常用的分隔符有空格(" ")、制表符("\t")、逗号(",")等。

实验：

找到你电脑上的任意一个记事本文件，查看其占用字节数。然后使用记事本等编辑器重新编辑，然后查看其文件大小。

1. 打开和关闭文件

打开文件

在使用文件前必须成功打开相应的文件才能够进行后续的读写操作。C 语言标准库中提供了 `fopen` 函数用于打开文件。

使用 C 语言标准库对文件进行操作需要包含头文件 `stdio.h`。

fopen 函数

```
FILE *fopen(const char *pathname, const char *mode);
```

说明:

1. `pathname` 参数是需要打开文件的路径（相对路径或决定路径）。
2. `mode` 参数用于控制文件的打开方式和操作权限。
3. 返回值是 FILE 类型的指针，用于对文件进行I/O 操作，我经常成之为文件流指针（stream）。成功则返回结构体地址，失败返回 `NULL`。
4. fopen函数发生错误时会设置全变变量 `errno` 的值。可以通过 `perror` 函数打印该值对应的错误信息。

`FILE` 是结构体类型别名，大致定义如下:

```
typedef struct {
    // ... 结构体内的成员变量
} FILE;
```

在使用标准库函数对文件进行操作时，我们无需关心 `FILE` 结构体的成员变量有哪些。

mode 参数详解

模式	说明
`"r"` 或 `"rt"`	从文件头位置读文本文件。
`"r+"` 或 `"rt+"`	从文件头位置读/写文本文件。
`"w"` 或 `"wt"`	从文件头位置写文本文件，没有文件则创建文件，文件存在则清空内容。
`"w+"` 或 `"wt+"`	从文件头位置读/写文本文件，没有文件则创建文件，文件存在则清空内容。
`"a"` 或 `"at"`	从文件头位置末尾写文本文件，没有文件则创建文件，文件存在则内容不变，在后面追加新内容。
`"a+"` 或 `"at+"`	从文件头位置末尾读/写文本文件，没有文件则创建文件，文件存在则内容不变，在后面追加新内容。

上述 `mode` 参数的值都是文本文件模式对文件进行操作。文本文件的读写方式默认是 `t`（text），可以省略不写。当以二进制方式对文件进行操作时，mode 参数的值中需要添加字符 `b`(binary)。

如下表所示

文本模式	二进制模式	说明
`"r"` 或 `"rt"`	`"rb"`	从文件头位置读文件。
`"r+"` 或 `"rt+"`	`"rb+"`	从文件头位置读/写文件。
`"w"` 或 `"wt"`	`"wb"`	从文件头位置写文件。
`"w+"` 或 `"wt+"`	`"wb+"`	从文件头位置读/写文件。
`"a"` 或 `"at"`	`"ab"`	从文件头位置末尾写文件（追加）。
`"a+"` 或 `"at+"`	`"ab+"`	从文件头位置末尾读/写文件（追加）。

关闭文件

在文件使用完毕后必须进行关闭，否则可能浪费资源和数据丢失。使用标准库函数 `fclose` 可以关闭 使用 `fopen` 成功打开的文件。

fclose 函数

```
int fclose(FILE *stream);
```

说明:

1. stream 参数是 fopen 成功打开文件后返回的文件结构体指针。
2. 成功关闭则返回 0，失败返回 EOF（-1），同时设置错误号 `errno`。

示例:

尝试以文本文件只写的方式打开一个文件 `myfile.txt`，如果文件不存在则创建该文件并打开。

```
// filename: fopen_for_write.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    FILE * pfile = NULL;  // 用于保存已经打开的文件

    // 打开文件 myfile.txt
    pfile = fopen("myfile.txt", "w");
    if (NULL == pfile) {
        // 根据全局变量 errno 错误号打印错误原因
        perror("打开文件 myfile.txt");
        return -1;
    }
    // ... 写入文件的内容（略）

    // 关闭文件
    fclose(pfile);

    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ ls
fopen_for_write.c
weimingze@mzstudio:~$ gcc -o fopen_for_write fopen_for_write.c
weimingze@mzstudio:~$ ./fopen_for_write
weimingze@mzstudio:~$ ls
fopen_for_write  fopen_for_write.c  myfile.txt
weimingze@mzstudio:~$ ls -l myfile.txt
-rw-r--r-- 1 weimingze weimingze 0 12月  5 14:35 myfile.txt
```

从运行结果可知，开始文件夹内只有 `fopen_for_write.c` 编译后多了一个 `fopen_for_write`文件，程序运行后又创建了 `myfile.txt`，`myfile.txt` 文件的长度是 0 字节。

示例：

尝试以文本文件只读的方式打开文件 `myfile.txt`，如果失败则报错。

```c
// filename: fopen_for_read.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    FILE * pfile = NULL;  // 用于保存已经打开的文件

    // 以只读方式打开文件 myfile.txt
    pfile = fopen("myfile.txt", "r");   //<-- 注意此处是 "r"
    if (NULL == pfile) {
        // 根据全局变量 errno 错误号打印错误原因
        perror("打开文件 myfile.txt");
        return -1;
    }
    // ... 读取文件的内容（略）

    // 关闭文件
    fclose(pfile);

    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ ls
fopen_for_read.c  myfile.txt
weimingze@mzstudio:~$ gcc -o fopen_for_read fopen_for_read.c
weimingze@mzstudio:~$ ls
fopen_for_read  fopen_for_read.c  myfile.txt
weimingze@mzstudio:~$ ./fopen_for_read
weimingze@mzstudio:~$ rm myfile.txt # 删除文件 myfile.txt
weimingze@mzstudio:~$ ls
fopen_for_read  fopen_for_read.c
weimingze@mzstudio:~$ ./fopen_for_read
打开文件 myfile.txt: No such file or directory
```

从运行结果可见，开始时存在文件 `myfile.txt`，程序运行不会有任何提示。当删除文件 `myfile.txt` 后再运行程序，则程序提示：`No such file or directory`。

实验：

写程序，使用循环创建 `file1.txt`、`file2.txt`、`file3.txt`、…`file20.txt` 等一共 20 个空的文件。

2. 文本文件的读写操作

当 `fopen` 函数打开文件时，第二个参数 `mode`（打开模式）中含有 `t` 标志或 没有 `b` 标志时则为以文本方式打开文件。当以文本文件打开文件时，建议使用文本文件读写相关的函数进行操作。

文本文件的写操作

文本文件写入操作常用的函数有 `fputc`、`fputs`、`fprintf`。

文本文件写入相关的函数

函数	说明
`int fputc(int c, FILE *stream);`	将一个字符串 `c` 转为无符号`unsigned char` 后写入文件 `stream`。
`int fputs(const char *str, FILE *stream);`	一个字符串 `str` 写入文件 `stream`，不写入字符串的尾零 `\0`。
`int fprintf(FILE *stream, const char *format, ...);`	使用 `printf` 函数的方式将格式化后的数据写入文件 `stream`。

示例：

```c
// filename: text_write.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    FILE * pf = NULL;  // 用于保存已经打开的文件

    pf = fopen("myinfo.csv", "w");
    if (NULL == pf) {
        perror("打开文件 myinfo.csv");
        return -1;
    }

    // 以下写入一行 "1,weimingze,35\r\n"。
    fputc('1', pf);  //  写入一个 "1"
    fputc(',', pf);  // 写入一个 逗号
    fputs("weimingze,35\r\n", pf);  // 写入 14 个字节
    // 以下写入一行 "2,laowei,18\r\n"。
    fprintf(pf, "%d,%s,%d\r\n", 2, "laowei", 18);

    // 关闭文件
    fclose(pf);
```

```
    return 0;
}
```

编译和运行结果如下:

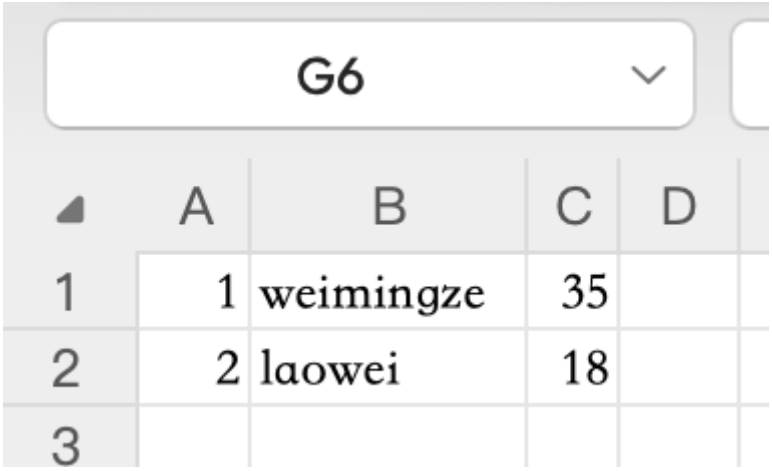
```
weimingze@mzstudio:~$ ls
text_write.c
weimingze@mzstudio:~$ gcc -o text_write text_write.c
weimingze@mzstudio:~$ ./text_write
weimingze@mzstudio:~$ ls
myinfo.csv  text_write  text_write.c
weimingze@mzstudio:~$ cat myinfo.csv
1,weimingze,35
2,laowei,18
```

说明:

可见上述程序生成的 `myinfo.csv` 文件的内容是

```
1,weimingze,35
2,laowei,18
```

Linux/Mac OS 终端中使用 `cat myinfo.csv` 命令是显示文件 `myinfo.csv` 文件的内容。此时的 `mycsv` 文件是标准的 csv 格式的文件，这个文件可以使用电子表格软件（如：wps）打开。打开后的效果如下：



文本文件的读操作

文本文件读取操作常用的函数有 `fgetc`、`fgets`、`fscanf`。

文本文件读取相关的函数

函数	说明
`int fgetc(FILE *stream);`	从文件流（`stream`）中读取下一个字符，成功则返回 `unsigned char` 转为 int 的值，失败则返回 EOF(值为-1)
`char *fgets(char *s, int size, FILE *stream);`	从文件流（`stream`）中读取最多 size -1 个字符形成字符串放入 s 指向的内存缓冲区中，读取到文件结束（EOF）或换行符则结束读取。在缓冲区末尾添加尾零 `'\0'`。
`int fscanf(FILE *stream, const char *format, ...);`	从文件流 `stream` 中使用 `scanf` 函数的方式读取数据到相应的变量中。成功返回读取的数据的个数，失败返回 EOF。

示例：

将上述 `myinfo.csv` 文件的内容读取出来，将其打印在终端中。

```c
// filename: text_read.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char * argv[]) {
    FILE * pf = NULL;  // 用于保存已经打开的文件
    int number;  // 用于保存编号
    char name[100];  // 用于保存性命
    int age;  // 用于保存年龄。

    pf = fopen("myinfo.csv", "r");
    if (NULL == pf) {
        perror("打开文件 myinfo.csv");
        return -1;
    }
    // 循环 读取，直至失败时才退出循环
    while (1) {
        // 读取一行数据并打印，数据的格式为: "1,weimingze,35\r\n"。
        int temp; // 用来保存临时的数据
        char buf[120];  // 用于保存一段文件的内容。
        char * find;  // 用于记录查找字符时找到的位置。
        // 使用scanf读取前面的数字，如果返回 EOF 说明没有读取到数据
        if (EOF == fscanf(pf, "%d", &number)) {
            break;  // 没有数据了
        }
        temp = fgetc(pf); // 读取一个逗号分隔符
        if (temp != ',') {
            printf("文件格式出错!\n");
```

```
            break;
        }
        // 读取一行中剩余的部分。格式为: "weimingze,35\r\n"
        if (NULL == fgets(buf, sizeof(buf), pf)) {
            printf("文件格式有错!");
            break;
        }
        // 查找逗号的位置，用于解析出姓名。
        find = strstr(buf, ",");
        if (NULL == find) {
            printf("姓名后没有找到逗号!\n");
            break;
        }
        // 先将 find 指向的逗号该成 `\0`, 作为姓名字符串的尾零。
        *find = '\0';
        // 将buf开始，到 find 之间的字符复制到name
        strcpy(name, buf);
        find++;  // find 指向 下一个字符（年龄的开始）。
        age = atoi(find);  // 将字符串转为整数，末尾的"\r\n" 会被忽略。
        // 打印获取到的姓名等信息
        printf("编号: %d, 姓名: %s, 年龄: %d\n", number, name, age);
    }

    // 关闭文件
    fclose(pf);

    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ gcc -o text_read text_read.c
weimingze@mzstudio:~$ ./text_read
编号: 1, 姓名: weimingze, 年龄: 35
编号: 2, 姓名: laowei, 年龄: 18
```

上述程序中可以看出。当我们从文本文件中读取文件的信息时，需要手动解析文件的格式来获取每一行，每一列的数据。

练习

写两个程序

1. 第一个程序连续输入多个学生的姓名、年龄、成绩，当输入姓名给空字符串时结束输入。将输入的内容保存到文件 `student.txt` 中。
2. 第二个程序从文件 `student.txt` 中获取之前保存的所有学生的 姓名、年龄、成绩 等信息。然后打印到控制台终端上。

要求：使用文本文件的读写方式。自己定义文件 `student.txt` 的格式。

3. 二进制文件的读写操作

当 `fopen` 函数打开文件时，第二个参数 `mode`（打开模式）中含有 `b` 标志时是二进制方式打开文件。当以二进制方式打开文件时，建议使用二进制文件读写相关的函数进行操作。

二进制文件的读/写操作

二进制文件读/写操作常用的函数有 `fread` 和 `fwrite`。

二进制文件读/写入相关的函数

函数	说明
`size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`	从文件流 `stream` 中读取 `size * nmemb` 个字节存入 ptr 指向的缓冲区中。
`size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);`	将 ptr 指向的缓存区中的 `size * nmemb` 个字节写入文件 `stream` 中。

说明：

1. `size` 是每个成员的长度，比如一个结构体的大小，`nmemb` 是要写入成员的个数。最终读写的总字节数是 `size` ✖ `nmemb` 个字节数。
2. `size` 和 `nmemb` 都是大于等于 1 的正整数。
3. 读取时 `ptr` 指向的内存空间要大于等于 `size` ✖ `nmemb`，否则可能会引起读/写越界。
4. 使用以上两个函数需要包含头文件 `stdio.h`。
5. 如果读/写文件成功则返回操作数据块的 `nmemb`, 如果磁盘空间已满（写的情况）或文件到达文件尾（读的情况）则返回值可能小于 `nmemb`，甚至返回 `0`。

示例：

将一个结构体数组的全部内容写入到文件中。

```c
// filename: binary_write.c
#include <stdio.h>

struct student {
    char name[32];
    int age;
    int score;
```

```
};

int main(int argc, char * argv[]) {
    FILE * pf = NULL;  // 用于保存已经打开的文件
    struct student stus[2] = {{"zhangsan", 18, 100}, {"lisi", 19, 80}};
    int rw_value = 0;  // 用于保存读写文件的返回值

    printf("每个结构体的长度是: %ld\n", sizeof(struct student));
    printf("结构体的数据个数是: %ld\n", sizeof(stus)/sizeof(stus[0]));

    // 以二进制写的方式打开文件
    pf = fopen("mydata.txt", "wb");
    if (NULL == pf) {
        perror("打开文件 mydata.tx");
        return -1;
    }
    rw_value = fwrite(stus, sizeof(struct student),
                      sizeof(stus)/sizeof(stus[0]), pf);
    // 检查写入结果
    if (rw_value < sizeof(stus)/sizeof(stus[0])) {
        printf("写入文件的数据不够完整，可能都是数据! \n");
    }
    printf("fwrite returned: %d\n", rw_value);

    // 关闭文件
    fclose(pf);

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ ls
binary_write.c
weimingze@mzstudio:~$ gcc -o binary_write binary_write.c
weimingze@mzstudio:~$ ./binary_write
每个结构体的长度是: 40
结构体的数据个数是: 2
fwrite returned: 2
weimingze@mzstudio:~$ ls
binary_write  binary_write.c  mydata.txt
weimingze@mzstudio:~$ ls -l mydata.txt
-rw-r--r-- 1 weimingze weimingze 80 12月  6 10:16 mydata.txt
weimingze@mzstudio:~$
```

从运行结果可知，每一个结构体变量 `struct student` 在内存中占用 40 个字节，数组中共有两个数据元素。调用函数 `fwrite` 返回值是 2，成功生成了 `mydata.txt` 文件。该文件的长度是 80 个字节，正好是内存中两个结构体占用的内存空间的大小，即二进制文件的写操作就是将内存中的内容复制到文件中。

我们使用 **记事本** 等文本编辑器打开 `mydata.txt` 文件时，发现除了部分字符串可见，其余都是乱码。这是因为以二进制方式保存在文件中的数据是无法解析成文字的内容，因此出现乱码。

注意事项：

1. 使用二进制文件存储数据要注意字节序问题，如`年龄`和`成绩`以小端字节序的形成存入文件中，又将此文件发送给默认是大端字节序的机器读取数据就可能会出错，因此二进制文件要明确规定字节序。
2. 使用二进制文件存储数据要注意结构体字节对齐问题，通常我们使用 `#pragma pack(1)` 对结构体进行压缩以解决此问题。如果不压缩结构体，可能会引起不同计算机结构体对齐字节数不一致问题。
3. 使用二进制文件存储字符串数据时，上述结构体的中的 `name` 数组长度是 40, 但实际可能只使用了前面的部分用来保存姓名。数组后面的不用数据也存入到了文件中。在实际开发中后面这部分数据可能会保存有密码等机密信息。因此会带来安全隐患。因此要保证所有写入的数据的正确性和安全性。比如可以在写入文件前将字符串后面的数据全部清零。

示例:

下面我们写程序将上述保存 `mydata.txt` 文件的内容读取出来，然后将信息打印到控制台终端中。

```c
// filename: binary_read.c
#include <stdio.h>

struct student {
    char name[32];
    int age;
    int score;
};

// 定义数组的最大元素个数是 100
#define STU_LEN (100)

int main(int argc, char * argv[]) {
    FILE * pf = NULL;  // 用于保存已经打开的文件
    struct student stus[STU_LEN];
    int rw_value = 0;  // 用于保存读写文件的返回值
    int i;

    // 以二进制 读 的方式打开文件
    pf = fopen("mydata.txt", "rb");
    if (NULL == pf) {
        perror("打开文件 mydata.txt");
        return -1;
    }
    // 计划读取 100 个数据信息
    rw_value = fread(stus, sizeof(struct student), STU_LEN, pf);
    // 检查写入结果
```

```
    printf("fread returned: %d\n", rw_value);

    // 打印所有学生信息:
    for (i = 0; i < rw_value; i++) {
        printf("姓名: %s, 年龄: %d, 成绩: %d\n",
            stus[i].name, stus[i].age, stus[i].score);
    }

    // 关闭文件
    fclose(pf);

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ ls
binary_read.c  mydata.txt
weimingze@mzstudio:~$ ls -l mydata.txt
-rw-r--r-- 1 weimingze weimingze 80 12月  6 10:16 mydata.txt
weimingze@mzstudio:~$ gcc -o binary_read binary_read.c
weimingze@mzstudio:~$ ls
binary_read  binary_read.c  mydata.txt
weimingze@mzstudio:~$ ./binary_read
fread returned: 2
姓名: zhangsan, 年龄: 18, 成绩: 10
姓名: lisi, 年龄: 19, 成绩: 80
```

从运行结果可知，我们原计划读取 100 个结构体数据，实际只读取到了两个结构体的数据。`fread` 返回 2。

实验

1. 改写上述结构体 `struct student` 年龄（`age`）用一个字节表示，成绩（`score`）用 `float` 类型来表示。使用 `#pragma pack(1)` 压缩结构体后以二进制方式将数据保存到文件中。
2. 写程序，读取上述文件中的内容，并能够打印文件中的数据。

4. 文件的随机读写

文件的随机读写是指能够在文件的任意位置直接读取或写入数据，而无需按顺序从头到尾遍历内容。

文件的随机读写常用于二进制方式读写文件的过程中。对于文本方式读写文件时，由于每一行字符串的长度难于确定。因此很少使用文件的随机读写。

随机读写通过移动文件的读写位置来快速定位到指定位置，实现高效的数据操作。在软件使用的过程中，在听 MP3 音乐时，我们向后拖动滚动条来跳过前面内容进行播放，实际就可能用到文件的随机读操作。

应用场景

1. 需改部分内容。
2. 高效查询。
3. 二进制文件处理。
4. 日志追加。

文件的随机读写相关的函数

函数	说明
`long ftell(FILE *stream);`	返回当前文件流 `stream` 的读写位置。
`int fseek(FILE *stream, long offset, int whence);`	设置当前文件流的读写位置，成功返回0，失败返回 -1。
`void rewind(FILE *stream);`	将文件的读写位置设置为 0。

说明：

1. stream 参数的实际要操作的文件流对象。
2. offset 参数是相对位置的偏移量。
   - 大于 0 的数代表向文件末尾方向移动。
   - 小于 0 的数代表向文件头方向移动。
   - 0 表示相对于某个位置不变。
3. whence 参数是只相对于哪里进行移动。
   - SEEK_SET（值为 0），代表从文件头开始偏移。
   - SEEK_CUR（值为 1），代表从当前读写位置开始偏移。
   - SEEK_END（值为 2），代表从文件尾开始偏移。

示例：

上节课我们已经生成了以个长度为 80 字节的文件 `mydata.txt` 用来保存了两个学生的信息。我们已经知道内部的存储格式，接下来我们使用文件的随机读写获取文件中的部分信息。

```
// filename: file_random_read.c
#include <stdio.h>
```

```c
struct student {
    char name[32];
    int age;
    int score;
};

int main(int argc, char * argv[]) {
    FILE * pf = NULL;  // 用于保存已经打开的文件
    struct student astu;  // 用于保存一个学生的信息
    int age;  // 用于保存一个学生的年龄。

    // 以二进制 读 的方式打开文件
    pf = fopen("mydata.txt", "rb");
    if (NULL == pf) {
        perror("打开文件 mydata.txt");
        return -1;
    }
    // 获取初始时文件的读写位置。
    printf("打开文件时，文件的读写位置是: %ld\n", ftell(pf));
    // 将读写位置定位到文件末尾，然后 ftell 的返回值就是文件的长度。
    fseek(pf, 0, SEEK_END);
    // 获取当前文件的读写位置。
    printf("文件的长度是: %ld\n", ftell(pf));

    // 从当前位置向前移动 40 个字节。
    fseek(pf, -40, SEEK_CUR);
    // 打印文件的当前读写位置。
    printf("第二个学生信息在文件的读写位置是: %ld\n", ftell(pf));
    // 读取第二个学生的信息
    if (fread(&astu, sizeof(astu), 1, pf) < 1) {
        printf("读取第二个学生信息失败\n");
        goto exit_main;  // 关闭文件后退出
    }
    printf("姓名: %s, 年龄: %d, 成绩: %d\n", astu.name, astu.age, astu.score);
    // 打印当前的读写位置信息
    printf("读取第二个学生信息后的文件读写位置是 :%ld\n", ftell(pf));
    // 重新定位文件头 第 32 个字节的位置，读取第一个学生的年龄信息
    fseek(pf, 32, SEEK_SET);
    if (fread(&age, sizeof(age), 1, pf) < 1) {
        printf("读取第一个学生的年龄失败!\n");
        goto exit_main;
    }
    printf("第一个学生的年龄是: %d\n", age);

exit_main:
    // 关闭文件
    fclose(pf);

    return 0;
}
```

编译和运行结果如下：

```
weimingze@mzstudio:~$ ls
file_random_read.c  mydata.txt
weimingze@mzstudio:~$ ls -l mydata.txt
-rw-r--r-- 1 weimingze weimingze 80 12月  6 10:16 mydata.txt
weimingze@mzstudio:~$ gcc -o file_random_read file_random_read.c
weimingze@mzstudio:~$ ./file_random_read
打开文件时，文件的读写位置是: 0
文件的长度是: 80
第二个学生信息在文件的读写位置是: 40
姓名: lisi, 年龄: 19, 成绩: 80
读取第二个学生信息后的文件读写位置是: 80
第一个学生的年龄是: 18
```

从上述运行结果可以看出，使用 `fseek` 可以任意定位一个文件的某个位置进行读写，只要这个文件的格式是已知的，这种定位读写的方法是可行的，并且效率高。

实验：

自己写程序，尝试使用 `fseek` 和 `ftell` 进行文件的随机定位的读写操作。

5. 标准输入输出文件

标准输入输出文件 是计算机程序与操作系统之间进行数据交互的基础文件流。其中包含三个标准文件：`stdin`、`stdout`、`stderr`，这三个文件在程序启动时（进程开始时）就被打开且一直可用。

三个标准输入输出文件：

1. `stdin` - 标准输入文件，默认是键盘；ctrl+d键结束输入。
2. `stdout` - 标准输出，默认是控制台终端，可以重定向到文件。
3. `stderr` - 标准错误输出，用于输出错误提示信息，默认是控制台终端，可以重定向到文件。

注意：

此三个文件指针是全局变量，在 `stdio.h` 中声明，这三个文件指针不需要打开就可以使用，也不要使用 fclose 关闭三个文件指针。

我们在标准输入中使用的 `printf` 函数的实质就是向 `stdout` 中写入数据。`scanf` 函数则是从文件 `stdin` 中读取数据。即:

```c
printf("hello world!");
// 等同于
fprintf(stdout, "hello world!");

scanf("%d", &i);
```

```
// 等同于
fscanf(stdin, "%d", &i);
```

示例：

使用上述三个标准输入输出文件结合文件操作函数代替基本输入输出函数进行控制台终端终端的输入输出操作。

```
// filename: std_io.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    int value;
    // 打印 "hello world!\n" 到屏幕终端。
    fwrite("hello world!\n", 1, 13, stdout);
    fprintf(stderr, "I'm a error info\n");

    fputs("please input a number:", stdout);
    fscanf(stdin, "%d", &value);
    fprintf(stdout, "value: %d\n", value);

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o std_io std_io.c
weimingze@mzstudio:~$ ./std_io
hello world!
I'm a error info
please input a number:666
value: 666
```

可见，使用文件操作，结合标准输入输出文件也同样可以达到 基本输入输出函数的功能。

实验：

使用 fclose 函数关闭 `stdin`、`stdout`、`stderr` 三个文件，然后测试 `scanf`、`printf` 等函数是否可用。

6. 文件缓冲区管理

在 C 语言中使用标准库函数对文件进行操作时，为了提高函数操作的速度，标准库函数通常使用一段内存（通常称之为**缓冲区**）来暂存读写的数据代替真实的文件I/O操作。这样可能 带来的一个问题就是文件系统的数据可能不是应用程序操作的真实数据。数据可能会延迟。

比如在使用 `printf("x");` 的时候，可能在屏幕上看不到 `x` 这个信息。这个因为 `x` 被放入到了内存中，并没有真实的写到控制台终端上。解决这个问题的方法是清空缓冲区。将缓冲区的内容同步到真实的硬件设备。

C 语言标准库中的文件缓冲区管理函数 `fflush` 可以让缓冲区内的数据迅速同步到硬件设备，避免数据延迟。当然频繁的调用 fflush 函数也可能致使程序的运行速度降低。

`fflush` 函数的声明格式如下：

函数	说明
`int fflush(FILE *stream);`	将写入文件流 `stream` 数据立即同步到目标设备。

示例:

```c
// filename: c_fflush.c
#include <stdio.h>

int main(int argc, char * argv[]) {
    FILE * pf = NULL;  // 用于保存已经打开的文件
    int temp_value;

    pf = fopen("mydata.txt", "wb");
    if (NULL == pf) {
        perror("打开文件 mydata.txt");
        return -1;
    }
    fwrite("laowei", 1, 6, pf);
    printf("请输入一个整数让程序继续运行:\n");
     // 此时查看文件 mydata.txt 中并没有任何内容
    scanf("%d", &temp_value);
    // 调用 fflush 来清空缓存
    fflush(pf);
    // 此时在重新查看文件 mydatat.txt 发现有6个字节的内容
    printf("请查看文件 mydata.txt，然后输入一个整数让程序继续运行:\n");
    scanf("%d", &temp_value);

    // 关闭文件
    fclose(pf);

    return 0;
}
```

编译和运行结果如下

```
weimingze@mzstudio:~$ gcc -o c_fflush c_fflush.c
weimingze@mzstudio:~$ ./c_fflush
请输入一个整数让程序继续运行:
```

```
1
请查看文件 mydata.txt，然后输入一个整数让程序继续运行:
2
```

上述程序在输入 `1` 之前查看文件 `mydata.txt` 是没有任何内容的。在输入 `1` 之后再查看文件发现缓冲区的内容被写入到了文件中，可以看到 `fwrite` 函数写入文件的内容了。

第二十二章、动态库和静态库

1. 动态库和静态库概述

在软件开发的过程中经常需要多人合作开发或者公司和公司之间合作开发。那么如何合作呢?

软件合作的方式通常有以下几种

1. 源代码交付，合作者提供源代码，由软件发布者统一编译发布。
2. 静态库交付，合作者提供源代码编译后的二进制静态库，软件发布者在链接阶段合并到可执行程序。 这样可以避免源码泄露，又能达到提供源代码一样的效果。
3. 动态库交付，合作者提供源代码编译后的二进制动态库，软发发布者在可执行程序运行时动态加载动态库，对其中的函数和变量进行使用。

静态和动态的概念

- **静态** 是指编译时状态。
- **动态** 是指运行时状态。

静态库（Static Library） 是指在链接时将库的代码完整的合并到可执行程序中的软件部分。静态库文件的扩展名通常是 `.a`（Linux/Unix）或 `.lib`（Windows）。

动态库（Dynamic Library） 是指在运行时，在需要使用的时候需要动态加载到内存中并进行使用的软件部分。库文件的扩展名通常是 `.so`（Linux）、`.dylib`（Mac OS）或 `.dll`（Windows）。

本章将以 Linux 为例，在 Linux 操作系统平台上用同一段代码来制作静态库和动态库，并使用不同的方式来使用其中的函数。

假设现在有合作方写好的四个文件: `file1.c`、`file1.h`、`file2.c`、`file2.h`，内容如下：

`file1.c` 内容如下：

```c
// filename: file1.c
#include <stdio.h>
#include "file1.h"

void myfunc1(void) {
    printf("库函数 myfunc1 被调用\n");
}
```

`file1.h` 内容如下：

```
// filename: file1.h
#ifndef __FILE1_H
#define __FILE1_H

void myfunc1(void);

#endif // __FILE1_H
```

`file2.c` 内容如下：

```
// filename: file2.c
#include <stdio.h>
#include "file2.h"

void myfunc2(void) {
    printf("库函数 myfunc2 被调用\n");
}
```

`file2.h` 内容如下：

```
// filename: file2.h
#ifndef __FILE2_H
#define __FILE2_H

void myfunc2(void);

#endif // __FILE2_H
```

现在有软件发布方写的一个文件: `main.c`，内容如下：

```
// filename: main.c
#include "file1.h"
#include "file2.h"

int main(int argc, char * argv[]) {
    // 调用合作方的库函数
    myfunc1();
    myfunc2();

    return 0;
}
```

点击此处下载上述源代码 mylib.tar.gz

以上直接使用源代码的方式编译和运行结果如下：

```
weimingze@mzstudio:~$ ls
file1.c  file1.h  file2.c  file2.h  main.c
weimingze@mzstudio:~$ gcc -o file1.o -c file1.c  # 汇编成目标文件
weimingze@mzstudio:~$ gcc -o file2.o -c file2.c
weimingze@mzstudio:~$ gcc -o main.o -c main.c
weimingze@mzstudio:~$ gcc -o myapp main.o file1.o file1.o  # 链接
weimingze@mzstudio:~$ ./myapp
库函数 myfunc1 被调用
库函数 myfunc2 被调用
```

可见使用源代码编译是没有问题的，但如果合作方不给我们源代码，这是时我们可以要求合作方给我们一个动态库或静态库。

2. Linux 下制作动态库

本节课我们来将上节课的合作方写好的四个文件: `file1.c`、`file1.h`、`file2.c`、`file2.h` 在 Linux 操作系统平台上 使用 GCC 编译器制作成静态库 `libmylib.a`。然后将静态库和主程序 `main.c` 合并生成一个可执行文件 `myapp`。

GCC 生成静态库步骤：

1. 将源文件编译为目标文件（`.o`文件）。
2. 使用 `gcc` 工具链的 `ar` 命令打包为静态库。

编译为目标文件可以使用 `gcc` 命令的 `-c` 选项进行编译，使用 `-o` 进行输出，如果给出 `-o` 选项则默认使用 `.c` 的文件名。

如下所示

```
weimingze@mzstudio:~$ ls
file1.c  file1.h  file2.c  file2.h
weimingze@mzstudio:~$ gcc -c file1.c
weimingze@mzstudio:~$ gcc -c file2.c
weimingze@mzstudio:~$ ls
file1.c  file1.h  file1.o  file2.c  file2.h  file2.o
```

下面我们使用编译好的目标文件生成静态库文件`libmylib.a`，Linux/UNIX 下的静态库文件要求以 `lib` 开头，以 `.a` 结尾，中间才是库的名称。

使用 `gcc` 工具链的 `ar` 命令可以将 目标文件打包成为静态库文件。如下所示:

```
weimingze@mzstudio:~$ ar rcs libmylib.a file1.o file2.o
weimingze@mzstudio:~$ ls
file1.c  file1.h  file1.o  file2.c  file2.h  file2.o  libmylib.a
```

ar 命令选项

- r：替换已存在的成员
- c：创建库
- s：创建索引

最后 libmylib.a 就是最后的静态库文件。合作方需要将此文件连同 两个头文件 `file1.h`、`file2.h` 一起提供给 项目发布方才能够编译成为可执行文件。

项目发布方使用 libmylib.a 编译成为可执行文件的过程如下，假设静态库文件即头文件 `file1.h` 和 `file2.h` 都存在于 `mylib1` 文件夹下，结构如下所示

```
weimingze@mzstudio:~$ tree .
.
├── main.c
└── mylib1
    ├── file1.h
    ├── file2.h
    └── libmylib.a
```

使用 GCC 生成可执行程序的过程如下:

```
weimingze@mzstudio:~$ gcc -c main.c  -I mylib1
weimingze@mzstudio:~$ gcc -o myapp main.o -L mylib1 -l mylib
weimingze@mzstudio:~$ ./myapp
库函数 myfunc1 被调用
库函数 myfunc2 被调用
```

`gcc`编译选项说明：

- `-I <路径>` 选项是指定头文件的包含路径。
- `-L <路径>` 选项是指定库文件的搜索路径。
- `-l <名称>` 选项是指定库文件名，不用添加前缀 `lib` 和 后缀`.a`。

这样我们就完成了静态库的制作和链接。使用静态库生成的可执行程序不依赖静态库文件，因此软件发布时可以不携带静态库文件发布。

实验

尝试使用你自己的编译器完成静态库的制作。

3.Linux 下制作静态库

本节课我们来使用四个文件: `file1.c`、`file1.h`、`file2.c`、`file2.h` 在 Linux 操作系统平台上 使用 GCC 编译器制作成动态库 `libmylib.so`。

然后将主程序 `main.c` 编译成可执行文件 `myapp`，在通过动态加载的方式运行此可执行程序。

GCC 生成动态库步骤：

1. 使用 GCC 在编译和汇编阶段将源文件生成和位置无关代码目标文件 (使用 `-fPIC` 选项)。
2. 使用目标文件创建共享库 `libmylib.so`

生成目标文件

```
weimingze@mzstudio:~$ ls
file1.c  file1.h  file2.c  file2.h
weimingze@mzstudio:~$ gcc -fPIC -c file1.c file2.c
weimingze@mzstudio:~$ ls
file1.c  file1.h  file1.o  file2.c  file2.h  file2.o
weimingze@mzstudio:~$ gcc -shared -o libmylib.so file1.o file2.o
weimingze@mzstudio:~$ ls
file1.c  file1.h  file1.o  file2.c  file2.h  file2.o  libmylib.so
```

使用于位置无关的目标文件生成动态库 `libmylib.so`。

```
gcc -shared -o libmylib.so file1.o file2.o
```

gcc 选项说明

- `-fPIC` 选项是生成位置无关的代码，因为动态库在运行时加载的位置不固定。因此所有的变量和函数都要使用相对位置来进行计算，这就是位置无关。
- `-shared` 选项就是生成共享库（即动态库）。

这样我们即生成了动态库文件 `libmylib.so`，合作方需要将此文件连同 两个头文件 `file1.h`、`file2.h` 一起提供给 项目发布方才能够编译成为可执行文件。

下面我们再来讲解一下如何使用动态链接库。

主程序要使用动态链接库有两种方法:

1. 静态链接时加载，就是在程序启动时自动加载。
2. 运行时动态加载，就是使用 dl 系列的API函数在需要时再加载。

1、动态库的静态链接时加载

我们先来讲述静态链接时加载的主程序的编译和运行过程，假设动态的库文件 `libmylib.so` 和头文件 `file1.h`和 `file2.h` 都存在于 `mylib2` 文件夹下，结构如下所示

```
weimingze@mzstudio:~$ tree .
.
├── main.c
└── mylib2
    ├── file1.h
    ├── file2.h
    └── libmylib.so
```

编译和运行过程如下:

```
weimingze@mzstudio:~$ gcc -c main.c -I mylib2
weimingze@mzstudio:~$ gcc -o myapp main.o -L mylib2 -lmylib
weimingze@mzstudio:~$ ./myapp
./myapp: error while loading shared libraries: libmylib.so: cannot open shared o
bject file: No such file or directory
```

从上述运行结果可知，主程序 `myapp` 运行时出错，这是因为在程序运行时需要找到 `libmylib.so` 所在的文件夹。也就是说主程序 `myapp` 启动之前，需要使用 Shell 的环境变量 `LD_LIBRARY_PATH` 来设置动态库的路径，方法如下:

```
weimingze@mzstudio:~$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./mylib2
weimingze@mzstudio:~$ ./myapp
库函数 myfunc1 被调用
库函数 myfunc2 被调用
```

可见在启动应用前，设置了 `LD_LIBRARY_PATH` 的路径为 `./mylib2`，主程序能够正常运行了。

2、动态库的运行时动态加载

下面我们在来说一下如何使用 dl 系列的 API（应用程序接口）来 真正的动态加载和使用 `libmylib.so` 中的函数。

动态加载的主要函数

函数	说明
`void *dlopen(const char *filename, int flags);`	根据动态库的路径 `filename` 加载动态库并返回操作句柄，引用计数加1，如果动态库已经加载则只是将引用计数加1。成功返回非空值，失败返回 `NULL`。
`int dlclose(void *handle);`	减少动态库的引用计数，如果计数达到0 则卸载动态库（真正释放动态库）。成功返回 0，失败返回非零值。
`void *dlsym(void *handle, const char *symbol);`	在动态库中找到用 `symbol`参数指定名字的符号（通常是函数名），并返回符号对应的函数地址。失败返回 `NULL`
`char *dlerror(void);`	获取错误信息。

修改 `main.c` 文件如下:

```c
// filename: main.c
#include <stdio.h>
#include <dlfcn.h>

int main() {
    void *handle;  // 保存动态库的打开句柄
    void (*fn1)(void);  // 用于指向动态库内的函数 myfunc1
    void (*fn2)(void);  // 用于指向动态库内的函数 myfunc2

    // 打开动态库
    handle = dlopen("./mylib2/libmylib.so", RTLD_LAZY);
    if (NULL == handle) {
        fprintf(stderr, "%s\n", dlerror());
        return 1;
    }
    // 获取函数地址
    fn1 = dlsym(handle, "myfunc1");
    if (NULL == fn1) {
        printf("动态库内没有找到 myfunc1函数");
        goto exit_main;
    }

    fn2 = dlsym(handle, "myfunc2");
    if (NULL == fn2) {
        printf("动态库内没有找到 myfunc2函数");
        goto exit_main;
    }

    // 4. 使用函数指针调用动态库中的函数
    fn1();
```

```
    fn2();


exit_main:
    // 5. 关闭动态库
    dlclose(handle);

    return 0;
}
```

程序结构如下:

```
weimingze@mzstudio:~$ tree .
.
├── main.c
└── mylib2
    └── libmylib.so
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o myapp main.c
weimingze@mzstudio:~$ ./myapp
库函数 myfunc1 被调用
库函数 myfunc2 被调用
```

可见，使用动态加载时，编译的时候都不依赖库的头文件，使用 `dlopen` 打开库文件，在使用 `dlsym` 定位到库函数地址，然后就可以使用函数指针来调用动态库中的函数了。

静态库和动态库对比

特性	静态库	动态库
链接时机	编译时	运行时
文件大小	可执行文件较大	可执行文件较小
内存占用	每个程序独立占用	多个程序可共享
更新	需重新编译	只需替换库文件
依赖	无运行时依赖	需要库文件存在于系统中

动态库是现代软件开发的更常见选择，特别是在大型系统中，因为它支持模块化更新和更高效的资源利用。

实验：

尝试在自己的编译环境下制作动态库文件。

第二十三章、C 语言标准库

1. C 语言标准库简介

本章将对 **C 语言常用标准库函数**进行讲解。C 语言的标准库是编译器自带的库，其内部统一定义了一组函数和宏，方便开发人员使用。在不同版本的编译器里都会存在这些函数，并且有统一的调用接口，具有可移植性。

在 Linux 操作系统下 C 标准款对应的静态库文件是 `libc.a`，动态库文件是 `libc.so`，一般的编译器在编译时都会对这些库进行自动链接（不需要指定链接选项）。

在 C 标准库中，每个库函数均在某个头文件中声明，其类型包含函数原型。使用者仅通过 `#include` 预处理指令可使其内容可用。头文件声明了一组相关的函数，以及为便于使用而需要的任何必要类型和附加宏。

标准库的头文件如下:

```
<assert.h>      <math.h>         <stdlib.h>
<complex.h>     <setjmp.h>       <stdnoreturn.h>
<ctype.h>       <signal.h>       <string.h>
<errno.h>       <stdalign.h>     <tgmath.h>
<fenv.h>        <stdarg.h>       <threads.h>
<float.h>       <stdatomic.h>    <time.h>
<inttypes.h>    <stdbool.h>      <uchar.h>
<iso646.h>      <stddef.h>       <wchar.h>
<limits.h>      <stdint.h>       <wctype.h>
<locale.h>      <stdio.h>
```

C 语言的标准库中常用的内容

1. 标准输入/输出（stdio.h）
   - `fopen`、`fclose`、`fread`、`fscanf`、`printf`、`scanf` 等。
2. 字符串处理（string.h）
   - `strcat`、`strcat`、`memset`、`memcpy`等。
3. 数学计算（math.h）
   - `sin`、`cos` 等。
4. 内存管理、类型转换、随机数等（stdlib.h）
   - `malloc`、`free`、`atoi`、`atof`、`rand`、`srand` 等。

5. 时间函数（time.h）
   - `time`、`gmtime`、`mktime`、`ctime`、`localtime` 等。
6. 常用常量定义（stddef.h）
   - 如：`NULL`、`size_t` 等。
7. 字符分类和转换（ctype.h）
   - `isdigit`、`isspace`、`isalpha` 等。
8. 错误处理（errno.h）
   - `extern int errno`、`perror` 等。
9. 整型限制（limits.h）
   - `INT_MAX`、`INT_MIN`、`LONG_MAX`、`LONG_MIN`等。

实验

在已经安装 C 语言编译器的电脑上找到上述 C 语言标准款的头文件，然后查看头文件的内容。

2. 数学函数（math.h）

为了方便数学计算，C 语言标准库中提供和常用的数学函数。

头文件

```
math.h
```

常用的数学函数:

函数	说明
三角函数	
`double sin(double x);`	正弦（x 是弧度）
`double cos(double x);`	余弦（x 是弧度）
`double tan(double x);`	正切（x 是弧度）
`double asin(double x);`	反正弦，返回值范围 [-pi/2, pi/2]
`double acos(double x);`	反余弦，返回值范围 [0, pi]
`double atan(double x);`	反正切，返回值范围 [-pi/2, pi/2]
指数和对数函数	
`double sqrt(double x);`	求平方根
`double pow(double x, double y);`	幂运算，返回 x 的 y 次方。
`double exp(double x);`	返回 e 的 x 次幂
`double log(double x);`	自然对数
`double log10(double x);`	求以 10 为底 x 的对数
其它函数	
`double fabs(double x);`	求 x 的绝对值
`double ceil(double x);`	x 向上取整
`double floor(double x);`	x 向下取整
`double fmod(double x, double y);`	浮点数求余数

示例:

```c
// filename: math.c
#include <stdio.h>
#include <math.h>

#define PI (3.14159265)

int main(int argc, char * argv[]) {
    printf("sin(0): %f\n", sin(0));
    printf("cos(PI/4): %f\n", cos(PI/4));  //  cos(45°)
    printf("sqrt(9.0): %f\n", sqrt(9.0));  // 求 9 的平方根
    printf("log10(1000): %f\n", log10(1000));
```

```
    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o math math.c
weimingze@mzstudio:~$ ./math
sin(0): 0.000000
cos(PI/4): 0.707107
sqrt(9.0): 3.000000
log10(1000): 3.000000
```

练习:

写一个程序，输入一个角度(`0 ~ 360`)，返回这个角度的正弦值，余弦值，正切值。

> 注意：需要将角度转换成弧度再求值。

3. 时间函数（time.h）

C 语言标准库提供了处理日期和时间的相关函数。使用这些函数能够完成获取和转换时间的操作。

C 语言中标准库中关于时间的核心数据类型是 `time_t` 类型，它是 `long int` 类型的别名。它记录的是计算机纪元（1970年1月1日 0:0:0 UTC）至今的秒数。

UTC（Universal Time Coordinated）是**协调世界时** 是指基于 0 时区的时间。使用 **协调世界时** 可以解决跨时区问题。

也就是说计算机计时是记录自 **1970年1月1日 0:0:0 UTC** 时间至今的秒数。每过一秒，此数字加一。

头文件

```
time.h
```

常用的时间函数:

函数	说明
`time_t time(time_t *tloc);`	返回计算机元年至今的秒数，如果参数 `tloc` 不为 `NULL` 则存入 `tloc` 指向的地址中。
`char *ctime(const time_t *timep);`	给出秒数，返回日历时间的字符串起始地址。
`struct tm *gmtime(const time_t *timep);`	给出 **UTC时间** 秒数，返回UTC时间结构体的地址。
`struct tm *localtime(const time_t *timep);`	给出 **UTC时间** 秒数，，返回本地时间结构体地址（由计算机内设置的时区计算得到）
`time_t mktime(struct tm *tm);`	这是 localtime() 的反函数，它根据本地时间结构体内年月日等信息计算UTC时间的秒数并返回。
`char *asctime(const struct tm *tm);`	给出本地时间结构体，返回时间的字符串起始地址。

这里面我们用到了时间结构体 `struct tm`，其结构体定义 如下:

```
struct tm {
    int tm_sec;     /* 秒 [0-60] */
    int tm_min;     /* 分 [0-59] */
    int tm_hour;    /* 时 [0-23] */
    int tm_mday;    /* 月中的天数 [1-31] */
    int tm_mon;     /* 月[0-11]，0代表一月*/
    int tm_year;    /* 年（实际年份 - 1900） */
    int tm_wday;    /* 星期几 [0-6]，0代表星期日 */
    int tm_yday;    /* 年中的第几天 [0-365] */
    int tm_isdst;   /* 夏令时标志（>0：启用，0：禁用，<0：未知） */
};
```

需要注意的是 `gmtime` 和 `localtime` 是标准库内全局变量的首地址，这两个函数共用同一地址，因此当有其中一个函数调用后，其内存将被修改。这两个函数也不能用于多线程的场景中。如果要使用独立的时间结构体 `struct tm` 来保存信息。请使用 `gmtime_r` 和 `localtime_r` 函数，具体函数说明请查看 `Linux/UNIX` 的 `man` 手册。

基于同一原理 `ctime` 和 `asctime` 返回的字符串也是保存在一个全局的缓冲区中。要多次使用并避免冲突请使用 `ctime_r` 和 `asctime_r` 函数代替之。

另外还可以使用 `strftime` 函数将时间结构体格式化为字符串。函数声明如下：

```
size_t strftime(char *str, size_t maxsize,
                const char *format,
                const struct tm *timeptr);
```

`format`常用格式说明符：

- `%Y`：年份（如 2023）
- `%m`：月份（01-12）
- `%d`：日（01-31）
- `%H`：小时（00-23）
- `%M`：分钟（00-59）
- `%S`：秒（00-61，允许闰秒）
- `%A`：完整星期名
- `%B`：完整月份名
- `%c`：本地日期时间表示

示例：

```c
// filename: mytime.c
#include <stdio.h>
#include <time.h>

int main(int argc, char * argv[]) {
    time_t now;
    struct tm *local_time;
    struct tm *utc_time;
    char buf[100];
    struct tm alarm_time;
    time_t alarm_second;
    // 获取当前时间的秒数
    time(&now);
    // 打印当前时间的字符串
    printf("当前时间是: %s", ctime(&now));  // ctime返回值指向的字符串自带换行符

    // time_t 转为 本地时间结构体
    local_time = localtime(&now);
    printf("现在是本地时间: %04d-%02d-%02d %02d:%02d:%02d\n",
        local_time->tm_year+1900, local_time->tm_mon+1, local_time->tm_mday,
        local_time->tm_hour, local_time->tm_sec, local_time->tm_sec);

    // time_t 转为 UTC 时间
    utc_time = gmtime(&now);
    printf("现在是UTC时间: %04d-%02d-%02d %02d:%02d:%02d\n",
        utc_time->tm_year+1900, utc_time->tm_mon+1, utc_time->tm_mday,
        utc_time->tm_hour, utc_time->tm_sec, utc_time->tm_sec);
    // 时间结构体字符串
    printf("UTC时间是: %s", asctime(utc_time));
    strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M:%S %A, %B", local_time);
```

```
    printf("当前时间是: %s\n", buf);

    // 计算今天距离 2030 年元旦的秒数
    alarm_time = (struct tm){0, 0, 0, 1, 0, 2030-1900};
    alarm_second = mktime(&alarm_time);
    printf("现在距离 2030年1月1日还有 %ld 秒\n", alarm_second - now);

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o mytime mytime.c
weimingze@mzstudio:~$ ./mytime
当前时间是: Sun Dec  7 16:37:10 2025
现在是本地时间: 2025-12-07 16:10:10
现在是UTC时间: 2025-12-07 08:10:10
UTC时间是: Sun Dec  7 08:37:10 2025
当前时间是: 2025-12-07 08:37:10 Sunday, December
现在距离 2030年1月1日还有 128330570 秒
```

练习:

写程序，输入你的生日年月日，计算从出生到今天已经过了多少天。

4. 随机数生成函数

在 C 语言标准库中可以使用 `rand` 函数来生成随机数。使用 `rand` 函数生成的随机数是通过程序运行时数据运算的到的。也称之为**伪随机数**。

生成随机数在游戏领域、人工智能等领域非常常用。比如棋牌游戏中的洗牌就要用到随机数生成函数。

在使用 `rand` 函数时，程序每次第一得到的随机数都是固定的，后续也是如此。为解决这个问题，我们需要使用 `srand` 来设置 **随机种子**（算法的扰乱器），通常我们是使用时间作为随机种子，毕竟每次程序的启动时间一定不同。

头文件

```
stdlib.h
```

随机数生成相关函数

函数	说明
`int rand(void);`	返回一个 [0-RAND_MAX] 的随机整数。 RAND_MAX 的值可能根据编译器不同而不同。
`void srand(unsigned int seed);`	设置随机种子来调整新的计算次序。

提示：

如果希望得到 `0 ~ 100` 之间的随机数可以对 `rand()` 的返回值求余数即可，如 `rand() % 101`

示例：

```c
// filename: myrandom.c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main(int argc, char * argv[]) {
    printf("随机数的最大值 RAND_MAX: %d\n", RAND_MAX);

    printf("第一个随机数: %d\n", rand());
    printf("第二个随机数: %d\n", rand());

    // 用当前时间来设置随机种子
    srand(time(NULL));
    printf("第三个随机数: %d\n", rand());
    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o myrandom myrandom.c
weimingze@mzstudio:~$ ./myrandom
随机数的最大值 RAND_MAX: 2147483647
第一个随机数: 1804289383
第二个随机数: 846930886
第三个随机数: 1828154031
weimingze@mzstudio:~$ ./myrandom
随机数的最大值 RAND_MAX: 2147483647
第一个随机数: 1804289383
第二个随机数: 846930886
第三个随机数: 1210571382
```

从上述运行结果可知，重复运行时，每次前两个随机数的值都是一样的，当用时间来是设置随机种子后，第三个随机数的值就不一样了。

练习：

写一个人机对战的石头、剪刀、布猜拳游戏，其中 `0` 代表石头、`1` 代表剪刀、`2` 代表布。让电脑生成一个 `0~2` 的随机数保存在变量中，你输入一个 `0~2` 的数字表示上述选择，然后判断并打印出猜拳结果。

5. 进程相关的函数

C 语言标准库中提供和一些进程相关的函数，比如 `exit` 和 `abort` 用来结束当前进程，`system` 用来使用Shell 开启新的进程等。这些函数都在 `stdlib.h` 中声明。

头文件

```
stdlib.h
```

进程相关的函数

函数	说明
`void exit(int status);`	以 `status` 作为程序返回值正常退出进程。这是一个永不返回的函数。
`void abort(void);`	以异常方式退出进程。这是一个永不返回的函数。也不会调用 atexit注册的回调函数。
`int atexit(void (*function)(void));`	注册程序正常退出时的回调函数。
`int system(const char *command);`	调用 `Shell` 启动新命令，并等待命令执行完毕后返回，失败返回 `-1`，启动 `Shell` 失败返回 `127`，成功则返回命令的退出值。

示例：

```c
// filename: mysystem.c
#include <stdio.h>
#include <stdlib.h>

void this_process_exit(void) {
    printf("进程即将退出!\n");
}
```

```c
int main(int argc, char * argv[]) {
    int ret_value;

    // 注册程序退出时的回调函数
    atexit(this_process_exit);

    // 调用 Linux/UNIX 命令创建 文件夹 mydocs。
    ret_value = system("mkdir mydocs");
    printf("system函数调用的返回值是: %d\n", ret_value);

    exit(0);
    printf("程序在此之前退出，这一行不会打印。\n");

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ ls
mysystem  mysystem.c
weimingze@mzstudio:~$ gcc -o mysystem mysystem.c
weimingze@mzstudio:~$ ./mysystem
system函数调用的返回值是: 0
进程即将退出!
weimingze@mzstudio:~$ ls
mysystem  mysystem.c  mydocs
```

从运行结果可以看出：

1. `system` 函数创建文件夹 `mydocs` 成功了
2. 调用 `exit` 函数后没有返回到主函数。
3. `this_process_exit` 函数在程序退出前被调用了。

实验：

将上述程序中的 `exit(0);` 改为 `abort();`。重新编译并查看程序的运行结果，看有啥变化。

6. 字符分类和转换（ctype.h）

C 语言标准库中提供了 字符分类与处理 的一些函数。包括英文字符的大小写判断，大小写转换等函数。

头文件

```
ctype.h
```

进程相关的函数

函数	说明
字符分类	以下函数分类成功返回非零值，失败返回 0。
`int isalnum(int c);`	是否为字母或数字
`int isalpha(int c);`	是否为字母
`int iscntrl(int c);`	是否为控制字符
`int isdigit(int c);`	是否为十进制数字
`int isgraph(int c);`	是否为图形字符（除空格外的可打印字符）
`int islower(int c);`	是否为小写字母
`int isprint(int c);`	是否为可打印字符
`int ispunct(int c);`	是否为标点符号
`int isspace(int c);`	是否为空白字符（空格、`\t`、`\n`、`\f`、`\r`、`\v`）
`int isupper(int c);`	是否为大写字母
`int isxdigit(int c);`	是否为十六进制数字
`int isascii(int c);`	是否为 `ascii` 字符（值在0-127之间）
`int isblank(int c);`	是否为空格或制表符`\t`
字符转换函数	
`int toupper(int c);`	转换为大写
`int tolower(int c);`	转换为小写

示例：

写函数 `to_upper_str`，将字符串中的小写字符转成大写字符显示。

```
// filename: myctype.c
#include <stdio.h>
#include <ctype.h>

// 将字符串中的英文转为大写字符
void to_upper_str(char * str) {
    while (*str) {
        if (islower(*str))
            *str = toupper(*str);
```

```
        str++;
    }
}

int main(int argc, char * argv[]) {
    char buf[100] = "hElLo worLd!";

    to_upper_str(buf);

    printf("%s\n", buf);

    return 0;
}
```

编译和运行结果如下:

```
weimingze@mzstudio:~$ gcc -o myctype myctype.c
weimingze@mzstudio:~$ ./myctype
HELLO WORLD!
```

练习

写函数 stat_word_count 给出一段英文文章，返回英文文章中英文单词的个数，部分程序内容如下:

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

int stat_word_count(const char * str) {
    // ...
}

int main(int argc, char * argv[]) {
    char buf[100] = "hello world! \nhello china!\n";

    printf("%d\n", stat_word_count(buf));  // 打印 4
    return 0;
}
```

第二十四章、校园信息管理系统项目

信息管理系统（Information Management System, IMS）是一种用于收集、存储、处理和分发信息的软件系统，旨在有效管理和利用信息资源，支持决策制定和业务操作。

如下应用都属于信息管理系统。

- CRM -- 客户关系管理（Customer Relationship Management）
- ERP -- 企业资源计划（Enterprise Resource Planning）
- SCM -- 供应链关系管理（Supply Chain Management）
- HIS -- 医院信息系统(Hospital Information System)

信息管理系统的核心功能如下

- 数据收集：从内部或外部来源获取数据（如传感器、用户输入等）。
- 存储与管理：通过文件、数据库或云存储管理结构化或非结构化数据。
- 处理与分析：清洗、转换数据，并通过算法或工具（如BI工具）生成洞察。
- 信息传递：以报告、仪表盘、通知等形式向用户提供有用信息。
- 控制与安全：确保数据完整性、访问控制和合规性。

1. 校园信息管理系统项目简介

此项目的目标是实现对一个学校的班级、学生信息及学生成绩的管理。以实现小学校园数据的电子化，有利于查阅和数据分析，同时能够实现数据电子存档，减少纸张利用，节能减排。

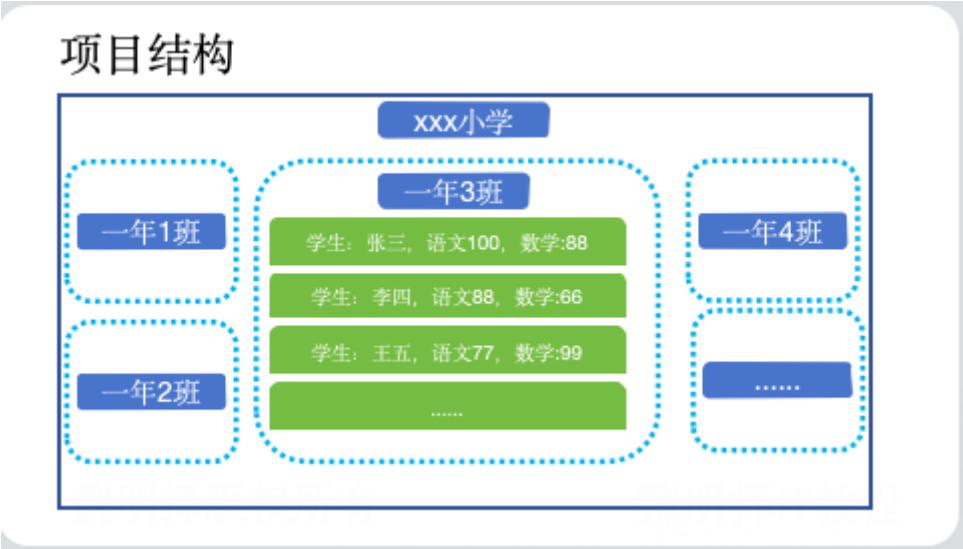
项目需求

本项目要求对一个学校的信息进行管理。项目存储的信息是一个学校有多个班级，一个班级有多个学生，一个学生有姓名、语文成绩和数学成绩这三项数据。

此项目的作用是用来管理学校学生的信息。能够新增班级，删除班级，管理班级内的学生信息等功能，并能将全部信息保存成为 `students.csv` 文件中，可以使用电子表格软件打开和编辑。

项目设计

根据需求我们了解到整体关系如下：



根基项目需求我们设计具体软件功能如下：

学校功能

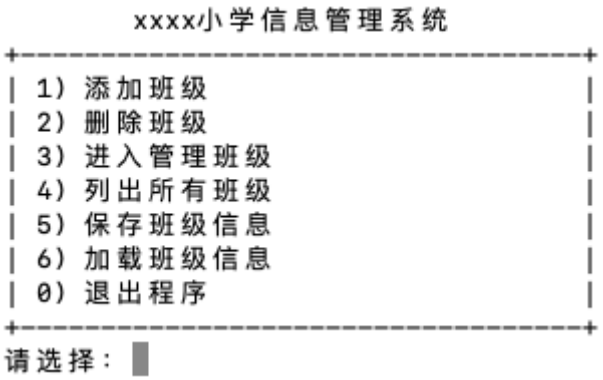
- 添加班级
- 删除班级
- 进入管理班级
- 列出所有班级
- 保存班级信息
- 加载班级信息
- 退出程序

班级功能

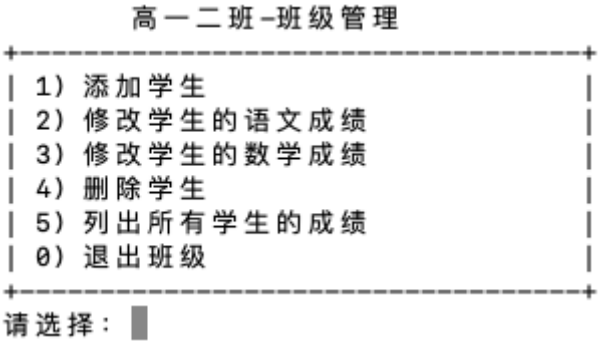
- 添加学生
- 修改学生的语文成绩
- 修改学生的数学成绩
- 删除学生
- 列出所有学生的成绩
- 退出班级

设计应用程序的界面如下：

应用程序操作界面（主界面）



班级管理界面



项目实现

此项目解决的问题的规模是可控的。一般国内的学校都不会超过 50 个班级，每个班级的学生不超过 100 人。因此我们采用固定长度的数组来存储上述信息。

以下介绍核心数据的实现方式。

学生结构体

```c
#define MAX_STU_NAME_LEN (32)   // 学生姓名的最大长度

// 学生类型
typedef struct student
{
    char name[MAX_STU_NAME_LEN];  // 姓名
    int chinese_score;  // 语文成绩
    int math_score;  // 数学成绩
} student_t;
```

上述结构体中规定 学生姓名`name` 最大长度 `MAX_STU_NAME_LEN` 是 32 个字节（包含尾零在内）。

班级结构体


```
// 班级名称的最大长度
#define MAX_CLASS_TITLE_LEN (64)

// 定义每个班级最大学生个数。
#define MAX_STU_COUNT_IN_CLASS_ROOM  (100)

// 班级的结构体
typedef struct class_room {
    char class_title[MAX_CLASS_TITLE_LEN];  // 班级名
    student_t student[MAX_STU_COUNT_IN_CLASS_ROOM]; // 每个班级的学生
    int student_count; // 用来记录具体的学生个数
} class_room_t;
```

上述结构体中规定了一个班级的具体信息，其中宏 `MAX_STU_COUNT_IN_CLASS_ROOM`定义了每个班级最多容纳 100 个学生。班级名称最多 64 个字节（包含尾零在内）。具体学生数由成员变量 `student_count` 记录。

学校的定义

如下的宏定义的每个学校容纳的最大班级数。

```
// 每个学校最大的班级数量
#define MAX_CLASS_COUNT_IN_SCHOOL (50)
```

使用全局的数组来存储学校的信息。定义如下

```
// 用于存放班级对象，初始状态为空
static class_room_t class_rooms[MAX_CLASS_COUNT_IN_SCHOOL] = {};
static int class_room_count = 0;  // 班级的个数
```

上述全局变量 `class_room_count` 用来记录数组中的数据个数，班级的数据从数组 `class_rooms` 索引 0 位置顺序排列，中间不留空隙。

项目文件

- 学校相关模块：`school.c`、`school.h`；
- 班级相关模块：`class_room.c`、`class_room.h`；
- 学生相关模块：`student.h`；
- 工具相关模块：`tools.c`、`tools.h`；
- 主模块：`main.c`。

初始程序框架内容如下

- 学校相关模块：`school.c`

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

#include "tools.h"
#include "school.h"
#include "class_room.h"

// 用于存放班级对象，初始状态为空
static class_room_t class_rooms[MAX_CLASS_COUNT_IN_SCHOOL] = {};
static int class_room_count = 0;  // 班级的个数


void show_school_menu(void)
{
    printf("        xxxx小学信息管理系统\n");
    printf("+-------------------------------------+\n");
    printf("| 1) 添加班级                          |\n");
    printf("| 2) 删除班级                          |\n");
    printf("| 3) 进入管理班级                      |\n");
    printf("| 4) 列出所有班级                      |\n");
    printf("| 5) 保存班级信息                      |\n");
    printf("| 6) 加载班级信息                      |\n");
    printf("| 0) 退出程序                          |\n");
    printf("+-------------------------------------+\n");
    printf("请选择: ");
    fflush(stdout);  // 清空缓冲区
}
// 此函数用来管理班级数据
void class_manager(void)
{
    while (1)
    {
        int sel = 0;
        show_school_menu();
        scanf("%d", &sel);
        switch (sel)
        {
            case 1:  // 1) 添加班级
                // add_class_room();
                break;
            case 2:  // 2) 删除班级
                // del_class_room();
                break;
            case 3:  // 3) 进入管理班级
                // enter_class_manager();
                break;
            case 4:  // 4) 列出所有班级
                // list_all_class_room();
```

```
                break;
            case 5:  // 5) 保存班级信息
                // save_to_csv_file(DEFAULT_DOC_PATHNAME);
                break;
            case 6:  // 6) 加载班级信息
                // load_from_csv_file(DEFAULT_DOC_PATHNAME);
                break;
            case 0:  // 0) 退出程序
                return;
            default:
                printf("不存在的选项，请重新输入\n");
                sleep(2);
        }
    }
}
```

班级相关模块：`class_room.c`

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

#include "tools.h"
#include "student.h"
#include "class_room.h"


// 此函数用来显示操作菜单
static void show_class_menu(class_room_t *aclass)
{
    printf("        %s-班级管理\n", aclass->class_title);
    printf("+-------------------------------------+\n");
    printf("| 1) 添加学生                          |\n");
    printf("| 2) 修改学生的语文成绩                |\n");
    printf("| 3) 修改学生的数学成绩                |\n");
    printf("| 4) 删除学生                          |\n");
    printf("| 5) 列出所有学生的成绩                |\n");
    printf("| 0) 退出班级                          |\n");
    printf("+-------------------------------------+\n");
    printf("请选择: ");
    fflush(stdout);
}

// 此函数用来管理学生数据
void student_manager(class_room_t *aclass)
{
    while(1) {
        int sel = 0;
        show_class_menu(aclass);
        scanf("%d", &sel);
        switch (sel)
        {
            case 1:  // 1) 添加学生
                // add_student(aclass);
```

```
                break;
            case 2:  // 2) 修改学生的语文成绩
                // modify_chinese_score(aclass);
                break;
            case 3:  // 3) 修改学生的数学成绩
                // modify_math_score(aclass);
                break;
            case 4:  // 4) 删除学生
                // del_student(aclass);
                break;
            case 5:  // 5) 列出所有学生的成绩
                // list_all_student_info(aclass);
                break;
            case 0:  // 0) 退出班级
                return;
            default:
                printf("不存在的选项，请重新输入\n");
                sleep(2);  // 让程序睡眠2秒
        }
    }
}
```

- 主模块：`main.c`。

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

int main(int argc, char *argv[])
{
    // 加载 csv 文件中的信息
    load_from_csv_file(DEFAULT_DOC_PATHNAME);
    // 进入学生信息管理主界面
    class_manager();

    return 0;
}
```

2. 工具模块的实现

此项目设计一个工具模块主要是将处理文字相关的功能提取出来，独立编写，独立测试。

英文的一个字符在内存中都是占用一个字节，在控制台终端中也是显示一个字符的宽度。

汉字的编码有两种：

- UNICODE
  - 与国际文字统一编码，与他国文字兼容。UNICODE 转码后使用 `UTF8` 编码保存。一个汉字一般占用三个字节。

- GB 系列的编码（GB2312、GBK、GB18030）
  - 国标编码，只与 ASCII 兼容。一个汉字使用 2 个字节或 4 个字节（极少数GB18030编码的汉字）保存。

中文的一个汉字，在 `UTF8` 编码时占用3个字节，在控制台终端中是显示 2 个字符的宽度。而在 `GB` 系列编码时大多占用 2 个字节，在控制台终端中同样是显示 2 个字符的宽度。因此需要特殊处理一下。

为了在控制台终端上能够兼容两种不同的编码。本项目编写了一个 名为 `tools` 的模块，相关文件有 `tools.c` 和 `tools.h`。

1. 函数 `is_utf8_code` 用来返回当前编译环境是哪种编码，如果是 UTF8 编码则返回 1 , 否则返回 0。
2. 函数 `get_display_width` 用于计算并返回 字符串 `s` 在当前环境下显示占用的宽度（以字符为单位）。
3. 函数 `center_to_display_width` 是将原字符串 `s` 按 `width` 显示宽度生成一个新字符串并存入 `target` 指向的地址中，左右两则用空格填充非内容部分。

完整代码如下:

文件 `tools.c`

```c
#include <string.h>
#include "tools.h"

// 测试当前文件的编码，如果是 UTF8 编码，则中文两个字占 6 字节，如果是 GBK 则占 4 字节
static int is_utf8_code(void)
{
    if (strlen("中文") == 6)
        return 1;
    return 0;
}

/* 返回字符串s的显示宽度，中文占两个英文字符的宽度
   如:
      "abc" -->3
      "中文" --> 4
      "ABC中文" --> 7
*/
int get_display_width(const char * s)
{
    int display_width = 0;
    int is_utf8 = is_utf8_code();
    // 计算字符串 s 占用屏幕控制台终端的宽度。
    while (*s) {
        if (*s >= 0 && *s <= 127) {  // 英文编码
            display_width++;  // 英文占一个字符宽
            s++;
```

```
        } else if (is_utf8) {  // 中文 UTF8 编码
            display_width += 2;  // 中文占两个字符宽
            s += 3;  // UTF8 编码一个汉字占 3 字节内存
        } else { // 中文 GBK 编码
            display_width += 2;
            s += 2;  // GBK 编码一个汉字占 2 字节内存
        }
    }

    return display_width;
}

/* 将字符串s的左右两端添加空格，使其达到 width 的显示宽度，存入 target 中
   如:
      s = 'ABC中文', width = 10
      返回:' ABC中文  '
*/
void center_to_display_width(const char * s, int width, char * target)
{
    // 得到当前字符的显示宽度
    int s_width = get_display_width(s);
    // 计算需要补充的空格数
    int fill_blank_count = width - s_width;

    // 计算左侧需要填充的空格数
    int left_blank = fill_blank_count / 2;
    // 计算右侧需要填充的空格数
    int right_blanks = fill_blank_count - left_blank;
    // 使用指针指向 target 的内容
    char * ptar = target;
    int i;

    if (fill_blank_count < 0) {  // 宽度小于 字符串宽度
        strcpy(target, s);
        return;
    }
    // 将 target 的左侧填充 fill 空格
    for (i = 0; i < left_blank; i++, ptar++) {
        *ptar = ' ';  // 填充空格
    }
    // 将 s 追加到 target 后面
    while(*s) {
        *ptar = *s;
        s++;
        ptar++;
    }
    // 将 target 的右侧填充空格
    for (i = 0; i < right_blanks; i++, ptar++) {
        *ptar = ' ';  // 填充空格
    }
    *ptar = '\0';   // 尾零
}
```

文件 `tools.h`

```
#ifndef __TOOLS_H
#define __TOOLS_H

/* 返回字符串s的显示宽度，中文占两个英文字符的宽度*/
int get_display_width(const char * s);

/* 将字符串s的左右两端添加空格，使其达到 width 的显示宽度，存入 target 中*/
void center_to_display_width(const char * s, int width, char * target);

#endif  // __TOOLS_H
```

3. 添加班级功能的实现

本小节我们来说一下如何添加班级到数组 `class_rooms` 中。添加班级需要先输入班级的名称，然后校验输入班级的名称是否合法，如果班级名称合法，则将班级作为数组 `class_rooms` 的下一个数据元素放入数组中，再将 `class_room_count` 做加一操作。

具体代码如下

文件 `school.c`

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

#include "tools.h"
#include "school.h"
#include "class_room.h"

// 用于存放班级对象，初始状态为空
static class_room_t class_rooms[MAX_CLASS_COUNT_IN_SCHOOL] = {};
static int class_room_count = 0;  // 班级的个数

// 添加班级
void add_class_room(void)
{
    char class_title[MAX_CLASS_TITLE_LEN*2];

    // 判断是否达到班级的最大数量
    if (class_room_count >= MAX_CLASS_COUNT_IN_SCHOOL) {
        printf("已经达到了编辑的最大数量!\n");
        return;
    }

    printf("请输入班级名称: ");
    fflush(stdout);
    scanf("%s", class_title);
    if (get_display_width(class_title) <= 10) {
```

```
        strcpy(class_rooms[class_room_count].class_title, class_title);
        class_room_count++;
        printf("添加班级%s成功!\n", class_title);
    } else {
        printf("添加班级失败，班级名太长!\n");
    }

    sleep(2);
}
```

4. 列出所有班级功能的实现

此功能是显示目前学校内有多少个班级，每一个班级的名称是什么。在显示时，每一个班级都有一个编号，这个编号是按着创建班级的顺序进行编号的。如果前面创建的班级删除了，后续班级的编号会依次前移。

实际班级编号就是班级在数组中的索引数加一，当班级编号减一后就得到了班级的索引位置。

具体代码如下

文件 `school.c`

```
void list_all_class_room(void)
{
    int i;
    char class_title[MAX_CLASS_TITLE_LEN*2];

    printf("+------+------------+\n");
    printf("| 序号 |  班级名称  |\n");
    printf("+------+------------+\n");
    for (i = 0; i < class_room_count; i++) {
        center_to_display_width(class_rooms[i].class_title, 10, class_title);
        printf("| %4d | %s |\n", i+1, class_title);
    }
    if (class_room_count)
        printf("+------+------------+\n");
}
```

5. 删除班级功能的实现

删除班级功能中，需要先显示班级列表，然后在班级列表中选择班级的编号。根据班号来确认要删除班级的位置，即班级编号减一。

删除班级时，我们将此班级后面的班级的数据依次前移，用后面一个覆盖前一个。然后将 `class_room_count` 做减一操作。

具体代码如下

文件 `school.c`

```
// 删除班级
void del_class_room(void)
{
    int number = 0;
    int index;

    list_all_class_room();
    printf("请输入删除班级的序号: ");
    fflush(stdout);
    scanf("%d", &number);
    index = number - 1;  // 对应列表的索引
    if (index < 0 || index >= class_room_count) {
        printf("您输入的序号有误，删除失败!\n");
        sleep(2);
    }
    // 将后续编辑依次向前覆盖
    for (; index < class_room_count-1; index++) {
        class_rooms[index] = class_rooms[index+1];
    }
    class_room_count--;
    printf("删除成功!\n");
    sleep(2);
}
```

6. 管理班级功能的实现

班级管理功能是要对一个班级进行管理，功能包括添加学生，删除学生等，在进入班级管理前，我们要先根据班级编号确认管理哪一个班级，然后再将此班级交给 `student_manager` 函数进行管理。

具体代码如下

文件 `school.c`

```
// 进入管理班级界面
void enter_class_manager(void)
{
    int number = 0;
    int index;
    class_room_t *aclass;

    list_all_class_room();
    printf("请输入一个要管理班级的序号: ");
    fflush(stdout);
    scanf("%d", &number);
```



```
    index = number - 1;  // 对应列表的索引
    if (index < 0 || index >= class_room_count) {
        printf("您输入的班级序号有误! \n");
        return;
    }
    // 将后续编辑依次向前覆盖
    for (; index < class_room_count-1; index++) {
        class_rooms[index] = class_rooms[index+1];
    }
    aclass = &class_rooms[index];
    student_manager(aclass);
}
```

7. 添加学生功能的实现

我将对学生的操作放在了 `class_room` 这个模块中。其中包括添加学生、删除学生，修改成绩等功能。在此模块的系列函数中都传入一个指针 `class_room_t *aclass`, 这个指针 `aclass` 指向要操作班级的结构体。

在添加学生前需要判断当前班级的学生数（`student_count`）是否达到了班级最大容纳的学生数（`MAX_STU_COUNT_IN_CLASS_ROOM`），如果达到最大容量则放弃添加。

添加学生是先要输入姓名、语文成绩、数学成绩，验证是否符合逻辑要求，如果不符合要求则退出添加。否则将如下信息填入班级的学生数组 `aclass->student` 的末尾，然后将班级学生数 `aclass->student_count` 做加一操作。

具体代码如下

文件 `class_room.c`

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

#include "tools.h"
#include "student.h"
#include "class_room.h"

// 添加学生信息
void add_student(class_room_t *aclass)
{
    char student_name[MAX_STU_NAME_LEN*2];
    int chinese_score;
    int math_score;

    if (aclass->student_count >= MAX_STU_COUNT_IN_CLASS_ROOM) {
        printf("班级 %s 学生已满，如法添加学生信息\n", aclass->class_title);
        return;
    }
```

```
    printf("请输入学生姓名: ");
    fflush(stdout);
    scanf("%s", student_name);
    // fgets(student_name, sizeof(student_name), stdin);
    if (strlen(student_name) >= MAX_STU_NAME_LEN) {
        printf("学生的名字太长，添加失败!\n");
        return;
    }
    printf("请输入学生的语文成绩: ");
    fflush(stdout);
    scanf("%d", &chinese_score);
    if (chinese_score < 0 || chinese_score > 100) {
        printf("学生成绩不在合法范围内，添加失败!\n");
        return;
    }
    printf("请输入学生的数学成绩: ");
    fflush(stdout);
    scanf("%d", &math_score);
    if (math_score < 0 || math_score > 100) {
        printf("学生成绩不在合法范围内，添加失败!\n");
        return;
    }

    student_t astu;
    strcpy(astu.name, student_name);
    astu.chinese_score = chinese_score;
    astu.math_score = math_score;

    aclass->student[aclass->student_count] = astu;
    aclass->student_count++;
    printf("添加学生 %s 成功!", student_name);
}
```

8. 列出所有学生信息功能的实现

列出所有学生信息就是将 `aclass` 指向的班级的所有学生的姓名、语文成绩、数学成绩信息显示出来。此功能需要为每个学生添加一个编号，这个编号是数组的索引位置加一，这个编号将用于后续修改和删除学生信息。

具体代码如下

文件 `class_room.c`

```
// 显示所有学生的信息
void list_all_student_info(class_room_t *aclass)
{
    int i;
    const student_t *astu;
    char stu_name_buf[MAX_STU_NAME_LEN*2];
```

```
    printf("+------+----------------------+--------+--------+\n");
    printf("| 序号 |         姓名         |语文成绩|数学成绩|\n");
    printf("+------+----------------------+--------+--------+\n");
    for (i = 0; i < aclass->student_count; i++) {
        astu = &aclass->student[i];
        center_to_display_width(astu->name, 20, stu_name_buf);
        printf("| %4d | %s |  %4d  |  %4d  |\n",
            i+1, stu_name_buf, astu->chinese_score, astu->math_score);
    }
    if (aclass->student_count)
        printf("+------+----------------------+--------+--------+\n");
}
```

9. 删除学生功能的实现

删除学生功能是根据学生的编号来删除 `aclass` 指向的班级的学生信息，一旦删除成功，则信息不可恢复。

删除学生需要先输入学生的序号，再验证序号的合法性。接下来根据序号定位位置，将此位置后的学生信息整体前移，然后将 `aclass->student_count` 做减一操作。

具体代码如下

文件 `class_room.c`

```
// 删除学生信息
void del_student(class_room_t *aclass)
{
    int number = 0;
    int index;

    list_all_student_info(aclass);

    printf("请输入要修改数学成绩的学生的序号: ");
    fflush(stdout);

    scanf("%d", &number);
    index = number - 1;
    if (index < 0 || index >= aclass->student_count) {
        printf("您输入的序号有错，删除失败!\n");
        return;
    }
    // 循环将 index +1 位置的数据覆盖 index 位置的数据
    for(; index < aclass->student_count-1; index++) {
        aclass->student[index] = aclass->student[index+1];
    }
    // 学生数减1
```

```
    aclass->student_count--;
}
```

10. 修改学生成绩功能的实现

修改学生成绩分为修改语文成绩和数学成绩两项、其做法几乎完全一致。

在修改成绩前先要用学生序号定位那个学生，然后输入新的成绩、再对该学生的成绩进行修改。

具体代码如下

文件 `class_room.c`

```c
// 修改语文成绩功能
void modify_chinese_score(class_room_t *aclass)
{
    int number = 0;
    int index;
    int new_score = 0;
    list_all_student_info(aclass);
    printf("请输入要修改语文成绩的学生的序号: ");
    fflush(stdout);

    scanf("%d", &number);
    index = number - 1;
    if (index < 0 || index >= MAX_STU_COUNT_IN_CLASS_ROOM) {
        printf("您输入的序号有误，修改失败!\n");
        return;
    }
    student_t *astu = &aclass->student[index];
    printf("请输入%s的新的语文成绩: ", astu->name);
    fflush(stdout);
    scanf("%d", &new_score);
    astu->chinese_score = new_score;
    printf("修改%s的语文成绩成功! \n", astu->name);
}

// 修改数学成绩功能
void modify_math_score(class_room_t *aclass)
{
    int number = 0;
    int index;
    int new_score = 0;
    list_all_student_info(aclass);
    printf("请输入要修改数学成绩的学生的序号: ");
    fflush(stdout);

    scanf("%d", &number);
    index = number - 1;
    if (index < 0 || index >= MAX_STU_COUNT_IN_CLASS_ROOM) {
        printf("您输入的序号有误，修改失败!\n");
```

```
        return;
    }
    student_t *astu = &aclass->student[index];
    printf("请输入%s的新的数学成绩: ", astu->name);
    fflush(stdout);
    scanf("%d", &new_score);
    astu->math_score = new_score;
    printf("修改%s的数学成绩成功! \n", astu->name);
}
```

11. 保存班级信息功能的实现

保存班级信息功能是将 `class_rooms` 结构体数据存储于某个文件中。文件格式指定为 `csv` 文件格式。

其默认的存档文件定义如下:

```
// 默认的文档存储路径
#define DEFAULT_DOC_PATHNAME   "./students.csv"
```

存档后的 `csv`文件格式如下:

班级名称	学生姓名	语文成绩	数学成绩
一年1班	张三	100	99
一年1班	李四	98	97
一年2班	王五	96	95
一年2班	赵六	94	93

从上述表格可以看出，班级名称相同，则属于一个班内的学生。

具体实现方法分为两部分，第一部分 `school` 模块中的 `save_to_csv_file` 负责打开文件，然后遍历每一个班级。第二部分再由各个班级模块 `class_room` 中的 `writer_to_csv_writer` 负责自己班级的学生信息写入文件中。

具体代码如下

文件 `school.c`

```
// 保存班级信息，成功返回 1，失败返回 0;
int save_to_csv_file(const char *path_name)
{
```

```
    const char * header = "班级名称,学生姓名,语文成绩,数学成绩";
    FILE * file = fopen(path_name, "w");  // 打开文件
    int i;

    if (NULL == file) {
        printf("打开文件失败，保存文件失败!\n");
        return 0;
    }
    fprintf(file, "%s\r\n", header);

    for (i = 0; i < class_room_count; i++) {
        writer_to_csv_writer(&class_rooms[i], file);
    }

    fclose(file);
    return 1;
}
```

文件 `class_room.c`

```
// 写入一个班级的学生数据
void writer_to_csv_writer(class_room_t *aclass, FILE *csv_file)
{
    int i;
    for (i = 0; i < aclass->student_count; i++) {
        student_t *astu = &aclass->student[i];
        fprintf(csv_file, "%s,%s,%d,%d\r\n", aclass->class_title,
            astu->name, astu->chinese_score, astu->math_score);
    }
}
```

12. 加载班级信息功能的实现

加载班级信息功能需要从 `csv` 文件中读取数据，然后解析这些数据保存到 `class_rooms` 数组中。需要注意的一点是我们将所有班级的信息都存入到了一个文本文件中，我们在读取数据时需要将这些学生信息根据第一列的班级名称来进行分组，将这一行数据归属于对应的班级。因此在读取一个新行数据并处理完成后，我们将当前行的班级名称存储于字符数组 `last_class_title` 中供下次比较使用。

在读取一行数据后，我们要解析逗号的位置，然后，将逗号替换成字符串结束符 `'\0'`, 然后用指针分别记录每个字符串的位置。解析完毕后，在调用 `class_room` 模块内的 `add_student_by_info` 函数，将此数据添加到某个班级。

具体代码如下

文件 `school.c`


```c
// 加载班级信息，成功返回 1，失败返回 0;
int load_from_csv_file(const char *path_name)
{
    FILE * file = fopen(path_name, "r");  // 打开文件
    // 记录之前的班级名，重复班级名的学生是同一班的学生
    char last_class_title[MAX_CLASS_TITLE_LEN] = "";
    char line_buf[100] = ""; // 读取一行数据的缓冲区。
    char *pclass_title;  // 指向班级名
    char *pstudent_name;  //  指向学生名
    char *pchinese_score;  // 指向语文成绩的字符串
    char *pmath_score;  // 指向语文成绩的字符串
    char *p;  // 用于指向每一行行数据的临时指针

    if (NULL == file) {
        printf("打开文件失败，读取文件失败!\n");
        return 0;
    }

    class_room_count = 0;  //  清空原有数据

    // 读取一行，跳过头部的第一行标题
    fgets(line_buf, sizeof(line_buf), file);
    // 每次读取一行，然后解析每一行的数据
    while(NULL != fgets(line_buf, sizeof(line_buf), file)) {
        // 解析每一行的数据
        pclass_title = line_buf;
        // 找到逗号，逗号改成班级名的尾零
        p = strstr(line_buf, ",");
        if (NULL == p)
            goto finished;
        *p = '\0';
        p++;  // 指向学生姓名
        // 指向学生姓名的起始地址。
        pstudent_name = p;
        p = strstr(p, ",");
        if (NULL == p)
            goto finished;
        *p = '\0';
        p++; // 指向语文成绩;

        // 指向学生语文成绩的起始地址。
        pchinese_score = p;
        p = strstr(p, ",");
        if (NULL == p)
            goto finished;
        *p = '\0';
        p++; // 指向语文成绩;

        // 指向学生数学成绩的起始地址。
        pmath_score = p;

        // 判断 班级名称发生改变。则说明已经是不同的班级
        if (0 != strcmp(last_class_title, pclass_title)) {
            // 如果已经超出了班级个数，则直接返回
```

```
            if (class_room_count == MAX_CLASS_COUNT_IN_SCHOOL) {
                goto finished;
            } else {
                // 初始化当前的班级信息
                strcpy(class_rooms[class_room_count].class_title, pclass_title);
                class_rooms[class_room_count].student_count = 0;
                class_room_count++;
            }
            strcpy(last_class_title, pclass_title);
        }
        //
        // 将此学生信息添加到 class_room_count-1 这个班级中
        add_student_by_info(&class_rooms[class_room_count-1],
            pstudent_name, atoi(pchinese_score), atoi(pmath_score));
        // 解析完毕,
    }
finished:
    fclose(file);
    return 1;
}
```

文件 `class_room.c`

```
// 将此学生添加到 aclass 这个班级中。成功返回1， 失败返回 0。
int add_student_by_info(class_room_t *aclass, const char * student_name,
    int chinese_score,int math_score)
{
    if(aclass->student_count >= MAX_STU_COUNT_IN_CLASS_ROOM)
        return 0;  // 满了
    strcpy(aclass->student[aclass->student_count].name, student_name);
    aclass->student[aclass->student_count].chinese_score = chinese_score;
    aclass->student[aclass->student_count].math_score = math_score;
    aclass->student_count++;
    return 1;
}
```

13. 项目大结局

至此，项目已经完成，下面我们对项目做一个总结。

前几节已将给出的项目中各个部分的说明和源代码。如果你需要项目完整的代码请点击 校园信息管理系统项目源代码 下载。

校园信息管理系统项目源代码结构如下：

```
school_info_manager/
├── class_room.c
├── class_room.h
├── main.c
├── Makefile
```



```
├── school.c
├── school.h
├── student.h
├── students.csv
├── tools.c
└── tools.h
```

此项目共分为 5 个模块，对应 4 个 `.c` 文件，模块说明和对应文件内容如下:

- 学校模块，负责班级的管理，相关文件：`school.c`、`school.h`;
- 班级模块，负责班级内学生的管理，相关文件：`class_room.c`、`class_room.h`;
- 学生模块，只有一个文件`student.h`，其中定义了学生的信息;
- 工具模块，用于计算一个字符串占用的显示宽度和左右填充空格等功能，相关文件：`tools.c`、`tools.h`;
- 主模块，用于调用其他模块，相关文件：`main.c`。

以下列出的上述项目的源代码。供参考。

文件 `main.c`

```c
#include <stdio.h>
#include "school.h"

int main(int argc, char *argv[])
{
    // 加载 csv 文件中的信息
    load_from_csv_file(DEFAULT_DOC_PATHNAME);
    // 进入学生信息管理主界面
    class_manager();

    return 0;
}
```

文件 `tools.h`

```c
#ifndef __TOOLS_H
#define __TOOLS_H

/* 返回字符串s的显示宽度，中文占两个英文字符的宽度*/
int get_display_width(const char * s);

/* 将字符串s的左右两端添加空格，使其达到 width 的显示宽度，存入 target 中*/
void center_to_display_width(const char * s, int width, char * target);

#endif  // __TOOLS_H
```

文件 `tools.c`

```c
#include <string.h>
#include "tools.h"

// 测试当前文件的编码，如果是 UTF8 编码，则中文两个字占 6 字节，如果是 GBK 则占 4 字节
static int is_utf8_code(void)
{
    if (strlen("中文") == 6)
        return 1;
    return 0;
}

/* 返回字符串s的显示宽度，中文占两个英文字符的宽度
   如:
      "abc" -->3
      "中文" --> 4
      "ABC中文" --> 7
*/
int get_display_width(const char * s)
{
    int display_width = 0;
    int is_utf8 = is_utf8_code();
    // 计算字符串 s 占用屏幕控制台终端的宽度。
    while (*s) {
        if (*s >= 0 && *s <= 127) {  // 英文编码
            display_width++;  // 英文占一个字符宽
            s++;
        } else if (is_utf8) {  // 中文 UTF8 编码
            display_width += 2;  // 中文占两个字符宽
            s += 3;  // UTF8 编码一个汉字占 3 字节内存
        } else { // 中文 GBK 编码
            display_width += 2;
            s += 2;  // GBK 编码一个汉字占 2 字节内存
        }
    }

    return display_width;
}

/* 将字符串s的左右两端添加空格，使其达到 width 的显示宽度，存入 target 中
   如:
      s = 'ABC中文', width = 10
      返回:' ABC中文  '
*/
void center_to_display_width(const char * s, int width, char * target)
{
    // 得到当前字符的显示宽度
    int s_width = get_display_width(s);
    // 计算需要补充的空格数
    int fill_blank_count = width - s_width;

    // 计算左侧需要填充的空格数
    int left_blank = fill_blank_count / 2;
    // 计算右侧需要填充的空格数
    int right_blanks = fill_blank_count - left_blank;
```

```
    // 使用指针指向 target 的内容
    char * ptar = target;
    int i;

    if (fill_blank_count < 0) {  // 宽度小于 字符串宽度
        strcpy(target, s);
        return;
    }
    // 将 target 的左侧填充 fill 空格
    for (i = 0; i < left_blank; i++, ptar++) {
        *ptar = ' ';  // 填充空格
    }
    // 将 s 追加到 target 后面
    while(*s) {
        *ptar = *s;
        s++;
        ptar++;
    }
    // 将 target 的右侧填充空格
    for (i = 0; i < right_blanks; i++, ptar++) {
        *ptar = ' ';  // 填充空格
    }
    *ptar = '\0';   // 尾零
}
```

文件 `student.h`

```
#ifndef __STUDENT_H
#define __STUDENT_H

#define MAX_STU_NAME_LEN (32)   // 学生姓名的最大长度

// 学生类型
typedef struct student
{
    char name[MAX_STU_NAME_LEN];  // 姓名
    int chinese_score;  // 语文成绩
    int math_score;  // 数学成绩
} student_t;

#endif  // __STUDENT_H
```

文件 `class_room.h`

```
#ifndef __CLASS_ROOM_H
#define __CLASS_ROOM_H

#include <stdio.h>
#include "student.h"

// 班级名称的最大长度
#define MAX_CLASS_TITLE_LEN (64)
```

```
// 定义每个班级最大学生个数。
#define MAX_STU_COUNT_IN_CLASS_ROOM  (100)

// 班级的结构体
typedef struct class_room {
    char class_title[MAX_CLASS_TITLE_LEN];  // 班级名
    student_t student[MAX_STU_COUNT_IN_CLASS_ROOM]; // 每个班级的学生
    int student_count; // 用来记录具体的学生个数
} class_room_t;


// 添加学生信息
void add_student(class_room_t *aclass);
// 修改语文成绩功能
void modify_chinese_score(class_room_t *aclass);
// 修改数学成绩功能
void modify_math_score(class_room_t *aclass);
// 删除学生信息
void del_student(class_room_t *aclass);
// 显示所有学生的信息
void list_all_student_info(class_room_t *aclass);
// 写入一个班级的学生数据
void writer_to_csv_writer(class_room_t *aclass, FILE *csv_file);
// 将此学生添加到 aclass 这个班级中。成功返回1， 失败返回 0。
int add_student_by_info(class_room_t *aclass, const char * student_name,
    int chinese_score,int math_score);
// 此函数用来管理学生数据
void student_manager(class_room_t *aclass);

#endif  // __CLASS_ROOM_H
```

文件 `class_room.c`

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

#include "tools.h"
#include "student.h"
#include "class_room.h"

// 添加学生信息
void add_student(class_room_t *aclass)
{
    char student_name[MAX_STU_NAME_LEN*2];
    int chinese_score;
    int math_score;

    if (aclass->student_count >= MAX_STU_COUNT_IN_CLASS_ROOM) {
        printf("班级 %s 学生已满，如法添加学生信息\n", aclass->class_title);
        return;
    }
```

```
    printf("请输入学生姓名: ");
    fflush(stdout);
    scanf("%s", student_name);
    // fgets(student_name, sizeof(student_name), stdin);
    if (strlen(student_name) >= MAX_STU_NAME_LEN) {
        printf("学生的名字太长，添加失败!\n");
        return;
    }
    printf("请输入学生的语文成绩: ");
    fflush(stdout);
    scanf("%d", &chinese_score);
    if (chinese_score < 0 || chinese_score > 100) {
        printf("学生成绩不在合法范围内，添加失败!\n");
        return;
    }
    printf("请输入学生的数学成绩: ");
    fflush(stdout);
    scanf("%d", &math_score);
    if (math_score < 0 || math_score > 100) {
        printf("学生成绩不在合法范围内，添加失败!\n");
        return;
    }

    student_t astu;
    strcpy(astu.name, student_name);
    astu.chinese_score = chinese_score;
    astu.math_score = math_score;

    aclass->student[aclass->student_count] = astu;
    aclass->student_count++;
    printf("添加学生 %s 成功!", student_name);
}

// 修改语文成绩功能
void modify_chinese_score(class_room_t *aclass)
{
    int number = 0;
    int index;
    int new_score = 0;
    list_all_student_info(aclass);
    printf("请输入要修改语文成绩的学生的序号: ");
    fflush(stdout);

    scanf("%d", &number);
    index = number - 1;
    if (index < 0 || index >= MAX_STU_COUNT_IN_CLASS_ROOM) {
        printf("您输入的序号有误，修改失败!\n");
        return;
    }
    student_t *astu = &aclass->student[index];
    printf("请输入%s的新的语文成绩: ", astu->name);
    fflush(stdout);
    scanf("%d", &new_score);
    astu->chinese_score = new_score;
    printf("修改%s的语文成绩成功! \n", astu->name);
}
```

```c
// 修改数学成绩功能
void modify_math_score(class_room_t *aclass)
{
    int number = 0;
    int index;
    int new_score = 0;
    list_all_student_info(aclass);
    printf("请输入要修改数学成绩的学生的序号: ");
    fflush(stdout);

    scanf("%d", &number);
    index = number - 1;
    if (index < 0 || index >= MAX_STU_COUNT_IN_CLASS_ROOM) {
        printf("您输入的序号有误，修改失败!\n");
        return;
    }
    student_t *astu = &aclass->student[index];
    printf("请输入%s的新的数学成绩: ", astu->name);
    fflush(stdout);
    scanf("%d", &new_score);
    astu->math_score = new_score;
    printf("修改%s的数学成绩成功! \n", astu->name);
}

// 删除学生信息
void del_student(class_room_t *aclass)
{
    int number = 0;
    int index;

    list_all_student_info(aclass);

    printf("请输入要修改数学成绩的学生的序号: ");
    fflush(stdout);

    scanf("%d", &number);
    index = number - 1;
    if (index < 0 || index >= aclass->student_count) {
        printf("您输入的序号有错，删除失败!\n");
        return;
    }
    // 循环将 index +1 位置的数据覆盖 index 位置的数据
    for(; index < aclass->student_count-1; index++) {
        aclass->student[index] = aclass->student[index+1];
    }
    // 学生数减1
    aclass->student_count--;
}

// 显示所有学生的信息
void list_all_student_info(class_room_t *aclass)
{
    int i;
    const student_t *astu;
    char stu_name_buf[MAX_STU_NAME_LEN*2];
```

```
    printf("+------+----------------------+--------+--------+\n");
    printf("| 序号 |         姓名         |语文成绩|数学成绩|\n");
    printf("+------+----------------------+--------+--------+\n");
    for (i = 0; i < aclass->student_count; i++) {
        astu = &aclass->student[i];
        center_to_display_width(astu->name, 20, stu_name_buf);
        printf("| %4d | %s |  %4d  |  %4d  |\n",
            i+1, stu_name_buf, astu->chinese_score, astu->math_score);
    }
    if (aclass->student_count)
        printf("+------+----------------------+--------+--------+\n");
}

// 写入一个班级的学生数据
void writer_to_csv_writer(class_room_t *aclass, FILE *csv_file)
{
    int i;
    for (i = 0; i < aclass->student_count; i++) {
        student_t *astu = &aclass->student[i];
        fprintf(csv_file, "%s,%s,%d,%d\r\n", aclass->class_title,
            astu->name, astu->chinese_score, astu->math_score);
    }
}

// 将此学生添加到 aclass 这个班级中。成功返回1， 失败返回 0。
int add_student_by_info(class_room_t *aclass, const char * student_name,
    int chinese_score,int math_score)
{
    if(aclass->student_count >= MAX_STU_COUNT_IN_CLASS_ROOM)
        return 0;  // 满了
    strcpy(aclass->student[aclass->student_count].name, student_name);
    aclass->student[aclass->student_count].chinese_score = chinese_score;
    aclass->student[aclass->student_count].math_score = math_score;
    aclass->student_count++;
    return 1;
}

// 此函数用来显示操作菜单
static void show_class_menu(class_room_t *aclass)
{
    printf("        %s-班级管理\n", aclass->class_title);
    printf("+-----------------------------------+\n");
    printf("| 1) 添加学生                       |\n");
    printf("| 2) 修改学生的语文成绩             |\n");
    printf("| 3) 修改学生的数学成绩             |\n");
    printf("| 4) 删除学生                       |\n");
    printf("| 5) 列出所有学生的成绩             |\n");
    printf("| 0) 退出班级                       |\n");
    printf("+-----------------------------------+\n");
    printf("请选择: ");
    fflush(stdout);
}

// 此函数用来管理学生数据
void student_manager(class_room_t *aclass)
```

```
{
    while(1) {
        int sel = 0;
        show_class_menu(aclass);
        scanf("%d", &sel);
        switch (sel)
        {
            case 1:  // 1) 添加学生
                add_student(aclass);
                break;
            case 2:  // 2) 修改学生的语文成绩
                modify_chinese_score(aclass);
                break;
            case 3:  // 3) 修改学生的数学成绩
                modify_math_score(aclass);
                break;
            case 4:  // 4) 删除学生
                del_student(aclass);
                break;
            case 5:  // 5) 列出所有学生的成绩
                list_all_student_info(aclass);
                break;
            case 0:  // 0) 退出班级
                return;
            default:
                printf("不存在的选项，请重新输入\n");
                sleep(2);  // 让程序睡眠2秒
        }
    }
}
```

文件 `school.h`

```
#ifndef __SCHOOL_H
#define __SCHOOL_H

// 每个学校最大的班级数量
#define MAX_CLASS_COUNT_IN_SCHOOL (50)

// 默认的文档存储路径
#define DEFAULT_DOC_PATHNAME   "./students.csv"

// 加载班级信息，成功返回 1，失败返回 0;
int load_from_csv_file(const char *path_name);
// 此函数用来管理班级数据
void class_manager(void);

#endif // __SCHOOL_H
```

文件 `school.c`

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

#include "tools.h"
#include "school.h"
#include "class_room.h"

// 用于存放班级对象，初始状态为空
static class_room_t class_rooms[MAX_CLASS_COUNT_IN_SCHOOL] = {};
static int class_room_count = 0;  // 班级的个数

// 添加班级
void add_class_room(void)
{
    char class_title[MAX_CLASS_TITLE_LEN*2];

    // 判断是否达到班级的最大数量
    if (class_room_count >= MAX_CLASS_COUNT_IN_SCHOOL) {
        printf("已经达到了编辑的最大数量!\n");
        return;
    }

    printf("请输入班级名称: ");
    fflush(stdout);
    scanf("%s", class_title);
    if (get_display_width(class_title) <= 10) {
        strcpy(class_rooms[class_room_count].class_title, class_title);
        class_room_count++;
        printf("添加班级%s成功!\n", class_title);
    } else {
        printf("添加班级失败，班级名太长!\n");
    }

    sleep(2);
}

void list_all_class_room(void)
{
    int i;
    char class_title[MAX_CLASS_TITLE_LEN*2];

    printf("+------+------------+\n");
    printf("| 序号 |  班级名称  |\n");
    printf("+------+------------+\n");
    for (i = 0; i < class_room_count; i++) {
        center_to_display_width(class_rooms[i].class_title, 10, class_title);
        printf("| %4d | %s |\n", i+1, class_title);
    }
    if (class_room_count)
        printf("+------+------------+\n");
}

// 删除班级
```

```c
void del_class_room(void)
{
    int number = 0;
    int index;

    list_all_class_room();
    printf("请输入删除班级的序号: ");
    fflush(stdout);
    scanf("%d", &number);
    index = number - 1;  // 对应列表的索引
    if (index < 0 || index >= class_room_count) {
        printf("您输入的序号有误，删除失败!\n");
        sleep(2);
    }
    // 将后续编辑依次向前覆盖
    for (; index < class_room_count-1; index++) {
        class_rooms[index] = class_rooms[index+1];
    }
    class_room_count--;
    printf("删除成功!\n");
    sleep(2);
}

// 进入管理班级界面
void enter_class_manager(void)
{
    int number = 0;
    int index;
    class_room_t *aclass;

    list_all_class_room();
    printf("请输入一个要管理班级的序号: ");
    fflush(stdout);
    scanf("%d", &number);
    index = number - 1;  // 对应列表的索引
    if (index < 0 || index >= class_room_count) {
        printf("您输入的班级序号有误! \n");
        return;
    }
    aclass = &class_rooms[index];
    student_manager(aclass);
}

// 保存班级信息，成功返回 1，失败返回 0;
int save_to_csv_file(const char *path_name)
{
    const char * header = "班级名称,学生姓名,语文成绩,数学成绩";
    FILE * file = fopen(path_name, "w");  // 打开文件
    int i;

    if (NULL == file) {
        printf("打开文件失败，保存文件失败!\n");
        return 0;
    }
    fprintf(file, "%s\r\n", header);
```

```
    for (i = 0; i < class_room_count; i++) {
        writer_to_csv_writer(&class_rooms[i], file);
    }

    fclose(file);
    return 1;
}

// 加载班级信息, 成功返回 1, 失败返回 0;
int load_from_csv_file(const char *path_name)
{
    FILE * file = fopen(path_name, "r");  // 打开文件
    // 记录之前的班级名，重复班级名的学生是同一班的学生
    char last_class_title[MAX_CLASS_TITLE_LEN] = "";
    char line_buf[100] = ""; // 读取一行数据的缓冲区。
    char *pclass_title;  // 指向班级名
    char *pstudent_name;  //  指向学生名
    char *pchinese_score;  // 指向语文成绩的字符串
    char *pmath_score;  // 指向语文成绩的字符串
    char *p;  // 用于指向每一行行数据的临时指针

    if (NULL == file) {
        printf("打开文件失败，读取文件失败!\n");
        return 0;
    }

    class_room_count = 0;  //  清空原有数据

    // 读取一行，跳过头部的第一行标题
    fgets(line_buf, sizeof(line_buf), file);
    // 每次读取一行，然后解析每一行的数据
    while(NULL != fgets(line_buf, sizeof(line_buf), file)) {
        // 解析每一行的数据
        pclass_title = line_buf;
        // 找到逗号，逗号改成班级名的尾零
        p = strstr(line_buf, ",");
        if (NULL == p)
            goto finished;
        *p = '\0';
        p++;  // 指向学生姓名
        // 指向学生姓名的起始地址。
        pstudent_name = p;
        p = strstr(p, ",");
        if (NULL == p)
            goto finished;
        *p = '\0';
        p++; // 指向语文成绩;

        // 指向学生语文成绩的起始地址。
        pchinese_score = p;
        p = strstr(p, ",");
        if (NULL == p)
            goto finished;
        *p = '\0';
        p++; // 指向语文成绩;
```

```
        // 指向学生数学成绩的起始地址。
        pmath_score = p;

        // 判断 班级名称发生改变。则说明已经是不同的班级
        if (0 != strcmp(last_class_title, pclass_title)) {
            // 如果已经超出了班级个数，则直接返回
            if (class_room_count == MAX_CLASS_COUNT_IN_SCHOOL) {
                goto finished;
            } else {
                // 初始化当前的班级信息
                strcpy(class_rooms[class_room_count].class_title, pclass_title);
                class_rooms[class_room_count].student_count = 0;
                class_room_count++;
            }
            strcpy(last_class_title, pclass_title);
        }
        //
        // 将此学生信息添加到 class_room_count-1 这个班级中
        add_student_by_info(&class_rooms[class_room_count-1],
            pstudent_name, atoi(pchinese_score), atoi(pmath_score));
        // 解析完毕,
    }
finished:
    fclose(file);
    return 1;
}

void show_school_menu(void)
{
    printf("        xxxx小学信息管理系统\n");
    printf("+-------------------------------------+\n");
    printf("| 1) 添加班级                          |\n");
    printf("| 2) 删除班级                          |\n");
    printf("| 3) 进入管理班级                      |\n");
    printf("| 4) 列出所有班级                      |\n");
    printf("| 5) 保存班级信息                      |\n");
    printf("| 6) 加载班级信息                      |\n");
    printf("| 0) 退出程序                          |\n");
    printf("+-------------------------------------+\n");
    printf("请选择: ");
    fflush(stdout);  // 清空缓冲区
}
// 此函数用来管理班级数据
void class_manager(void)
{
    while (1)
    {
        int sel = 0;
        show_school_menu();
        scanf("%d", &sel);
        switch (sel)
        {
            case 1:  // 1) 添加班级
                add_class_room();
```

```
                break;
            case 2:  // 2) 删除班级
                del_class_room();
                break;
            case 3:  // 3) 进入管理班级
                enter_class_manager();
                break;
            case 4:  // 4) 列出所有班级
                list_all_class_room();
                break;
            case 5:  // 5) 保存班级信息
                save_to_csv_file(DEFAULT_DOC_PATHNAME);
                break;
            case 6:  // 6) 加载班级信息
                load_from_csv_file(DEFAULT_DOC_PATHNAME);
                break;
            case 0:  // 0) 退出程序
                return;
            default:
                printf("不存在的选项，请重新输入\n");
                sleep(2);
        }
    }
}
```

下面总结一下项目的优缺点：

优点

- 模块化设计，设计思路清晰。
- 代码量少，功能齐全。
- 数据能够长期存储。
- 数据用电子表格软件（如:wps）可读。

缺点

- 学生只有两门成绩，难于扩展。
- 只适合一个学校的单机使用。
- 如果班级里面没有学生，则保存是会丢失班级信息，导致加载时缺少班级信息。
- 班级重名，保存后加载会合并班级。
- 没有异常处理机制，程序崩溃，数据丢失。

缺点的改进型存储方法

班级表:

classes.csv

格式

班级ID	班级名称
1	一年1班
2	一年2班

学生表

student.csv

格式

班级ID	学生姓名	语文成绩	数学成绩
1	张三	100	99
1	李四	98	97
2	王五	96	95
2	赵六	94	93

改进为关系型数据库存储

班级表（classes）

班级ID	班级名称
1	一年1班
2	一年2班

学生表（student）

学生ID	学生姓名	班级ID
1	张三	1
2	李四	1
3	王五	2
4	赵六	2

课程表（subject）

课程ID	课程
1	语文
2	数学

成绩表（score）

学生ID	课程ID	成绩
1	1	100
1	2	99
2	1	98
2	2	97
…	…	…

使用上述四个数据表来存储学生信息，如果再需要其他信息时可以通过数据表关系运算计算得到，这样将更有利于程序的扩展，也可让程序更具有可维护性。

总结

C 语言总结

学过的章节（共计 24 章）

章节	内容	章节	内容
1	开发环境的搭建	13	字符串
2	初步认识 C 语言	14	编译预处理
3	基础数据类型	15	函数
4	基本输入输出函数	16	结构体
5	运算符与表达式	17	联合体
6	选择语句	18	枚举
7	迭代语句	19	C 语言高级语法
8	跳转语句	20	动态内存管理
9	表达式语句和空语句	21	文件操作
10	其它语句	22	动态库和静态库
11	指针	23	C 语言标准库
12	数组	24	校园信息管理系统项目

以下是上述章节学过的内容。

运算符和表达式

名称	符号
后置自增/减	`++` `--`
函数调用	`()`
数组下标	`[]`
结构体(联合体)成员访问	`.`
结构体(联合体)指针成员访问	`->`
复合字面值	`(type){list}`
前置自增/减	`++` `--`
正负号	`+`（正号） `-`（负号）
逻辑非	`!`
按位取反	`~`
强制类型转换	`(type)`
解引用	`*`
取地址	`&`
求占用字节数	`sizeof`
查询对齐	`_Alignof`
乘、除、取模（求余数）	`*` `/` `%`
加、减	`+` `-`
左移、右移	`<<` `>>`
关系运算符比较	`<` `<=` `>` `>=` `==` `!=`
位运算	`&` `^` `|`
逻辑运算	`&&` `||`
条件运算符（三元运算符）	`?:`
赋值运算符	`=` `+=` `-=` `*=` `/=` `%=` `<<=` `>>=` `&=` `^=` `|=`
逗号运算符	`,`

选择语句

if 语句

```
if ( expression ) statement
if ( expression ) statement else statement
```

switch 语句

```
switch ( expression ) statement
```

迭代语句

for 语句

```
for ( expression ; expression ; expression ) statement
for ( declaration expression ; expression ) statement
```

while 语句

```
while ( expression ) statement
```

do-while语句

```
do statement while ( expression ) ;
```

跳转语句

break 语句

```
break;
```

continue 语句

```
continue;
```

goto 语句

```
goto 标识符(标签);
```

return 语句

```
return [表达式];
```

其他语句

表达式语句

```
表达式;
```

空语句

```
;
```

复合语句

```
{
    ...
}
```

标签语句

```
identifier : statement
case constant-expression : statement
default : statement
```

编译预处理指令

文件包含指令

```
#include <pathname>
#include "pathname"
```

宏定义指令

```
#define 标识符 [宏内容]
#define 标识符(...) [宏内容]
```

取消宏定义指令

```
#undef 标识符
```

条件编译指令

```
#if 常量表达式
...
#elif  常量表达式
...
#else
...
#endif
```

```
#ifdef 宏名
...
#elif  常量表达式
...
#else
...
#endif
```

```
#ifndef 宏名
...
#elif  常量表达式
...
#else
...
#endif
```

停止编译报错指令

```
#error [字符串]
```

编译参数设定

```
#pragma pack(n)
#pragma pack()
```

函数

函数定义

```
返回值的数据类型 函数名(数据类型1 形参变量1, 数据类型2 形参变量2, ...) {
    语句
}
```

函数调用

```
函数名(表达式1, 表达式2, ...)
```

函数声明

```
返回值的类型 函数名(数据类型1 [变量名1], 数据类型2 [变量名2], ...);
```

函数说明符

```
inline
_Noreturn
```

存储类别说明符

```
typedef
extern
static
_Thread_local (未讲解)
auto
register
```

类型修饰词

```
const
restrict
volatile
_Atomic (未讲解)
```

类型说明符

```
void
char
short
int
long
float
double
signed
unsigned
_Bool (未讲解)
_Complex (未讲解)
atomic-type-specifier (未讲解)
struct-or-union-specifier  (结构体或联合体)
```

```
enum-specifier  (枚举类型说明符)
typedef-name (类型别名)
```

结构体/联合体类型

```
struct/union [结构体或联合体名] {
    数据类型1  成员变量名1[: 占用位宽1], 成员变量名2[: 占用位宽2], ...;
    数据类型2  成员变量名4;
    // ... 其它成员变量
}[变量名1[={初始化列表1}]][,变量名2[={初始化列表2}], ...];
```

枚举类型

```
enum [枚举类型名] {
    枚举常量名1[ = 整数常量表达式1],
    枚举常量名2[ = 整数常量表达式2],
    ...
} [变量名 [= 整数表达式]];
```

标准库

类型	函数
标准输入输出 `stdio.h`	`printf`、`scanf`、`fopen`、`fclose`、`fgetc`、`fgets`、`fscanf`、`fputc`、`fputs`、`fprintf`、`fread`、`fwrite`、`ftell`、`fseek`、`rewind`
数学函数 `math.h`	`sin`、`cos`、`tan`、`asin`、`acos`、`atan`、`sqrt`、`pow`、`exp`、`log`、`log10`、`fabs`、`ceil`、`floor`、`fmod`
时间函数 `time.h`	`time`、`ctime`、`gmtime`、`localtime`、`mktime`、`asctime`、`strftime`
通用工具函数 `stdlib.h`	`rand`、`srand`、`malloc`、`calloc`、`realloc`、`free`、`abort`、`exit`、`atexit`、`atoi`、`atol`、`atoll`、`atof`、`system`、`abs`
字符分类和转换 `ctype.h`	`isalnum`、`isalpha`、`iscntrl`、`isdigit`、`isgraph`、`islower`、`isprint`、`ispunct`、`isspace`、`isupper`、`isxdigit`、`isascii`、`isblank`、`toupper`、`tolower`
标准宏定义 `stddef.h`	`NULL`、`size_t`

附录

ASCII 编码表

ASCII（American Standard Code for Information Interchange），中文名称为**美国信息交换标准代码**。它是计算机及通信领域常用的编码标准。

以下是 ASCII 中的 128 个字符的编码。

八进制值	十进制值	十六进制值	字符
000	0	00	NUL '\0' (null character)
001	1	01	SOH (start of heading)
002	2	02	STX (start of text)
003	3	03	ETX (end of text)
004	4	04	EOT (end of transmission)
005	5	05	ENQ (enquiry)
006	6	06	ACK (acknowledge)
007	7	07	BEL '\a' (bell)
010	8	08	BS '\b' (backspace)
011	9	09	HT '\t' (horizontal tab)
012	10	0A	LF '\n' (new line)
013	11	0B	VT '\v' (vertical tab)
014	12	0C	FF '\f' (form feed)
015	13	0D	CR '\r' (carriage ret)
016	14	0E	SO (shift out)
017	15	0F	SI (shift in)
020	16	10	DLE (data link escape)
021	17	11	DC1 (device control 1)
022	18	12	DC2 (device control 2)
023	19	13	DC3 (device control 3)
024	20	14	DC4 (device control 4)
025	21	15	NAK (negative ack.)
026	22	16	SYN (synchronous idle)
027	23	17	ETB (end of trans. blk)
030	24	18	CAN (cancel)
031	25	19	EM (end of medium)

032	26	1A	SUB (substitute)
033	27	1B	ESC (escape)
034	28	1C	FS (file separator)
035	29	1D	GS (group separator)
036	30	1E	RS (record separator)
037	31	1F	US (unit separator)
040	32	20	SPACE
041	33	21	!
042	34	22	"
043	35	23	#
044	36	24	$
045	37	25	%
046	38	26	&
047	39	27	'
050	40	28	(
051	41	29	)
052	42	2A	*
053	43	2B	+
054	44	2C	,
055	45	2D	-
056	46	2E	.
057	47	2F	/
060	48	30	0
061	49	31	1
062	50	32	2
063	51	33	3
064	52	34	4
065	53	35	5

066	54	36	6
067	55	37	7
070	56	38	8
071	57	39	9
072	58	3A	:
073	59	3B	;
074	60	3C	<
075	61	3D	=
076	62	3E	>
077	63	3F	?
100	64	40	@
101	65	41	A
102	66	42	B
103	67	43	C
104	68	44	D
105	69	45	E
106	70	46	F
107	71	47	G
110	72	48	H
111	73	49	I
112	74	4A	J
113	75	4B	K
114	76	4C	L
115	77	4D	M
116	78	4E	N
117	79	4F	O
120	80	50	P
121	81	51	Q

122	82	52	R
123	83	53	S
124	84	54	T
125	85	55	U
126	86	56	V
127	87	57	W
130	88	58	X
131	89	59	Y
132	90	5A	Z
133	91	5B	[
134	92	5C	\ '\'
135	93	5D	]
136	94	5E	^
137	95	5F	_
140	96	60	`
141	97	61	a
142	98	62	b
143	99	63	c
144	100	64	d
145	101	65	e
146	102	66	f
147	103	67	g
150	104	68	h
151	105	69	i
152	106	6A	j
153	107	6B	k
154	108	6C	l
155	109	6D	m

156	110	6E	n
157	111	6F	o
160	112	70	p
161	113	71	q
162	114	72	r
163	115	73	s
164	116	74	t
165	117	75	u
166	118	76	v
167	119	77	w
170	120	78	x
171	121	79	y
172	122	7A	z
173	123	7B	{
174	124	7C	|
175	125	7D	}
176	126	7E	~
177	127	7F	DEL

运算符优先级表

C 语言的运算符有十五个**优先级**，优先级数值越小，优先级越高。相同的优先级的运算符，具体要看运算符的**结合性**来决定哪个运算符先计算。

C 语言运算符优先级表

优先级	运算符	说明	结合性
1	`++` `--` `()` `[]` `.` `->` `(type){list}`	后置自增/减、函数调用、数组下标、结构体(联合体)成员访问、结构体(联合体)指针成员访问、复合字面值(C99)	自左向右
2	`++` `--` `+` `-` `!` `~` `(type)` `*` `&` `sizeof` `_Alignof`	前置自增/减、正负号、逻辑非、按位取反、强制类型转换、解引用、取地址、求占用字节数、查询对齐(C11)	自右向左
3	`*` `/` `%`	乘、除、取模（求余数）	自左向右
4	`+` `-`	加、减	自左向右
5	`<<` `>>`	左移、右移	自左向右
6	`<` `<=` `>` `>=`	关系运算符比较	自左向右
7	`==` `!=`	关系运算符等于、不等于	自左向右

8	`&`	按位与	自左向右
9	`^`	按位异或	自左向右
10	`|`	按位或	自左向右
11	`&&`	逻辑与	自左向右
12	`||`	逻辑或	自左向右
13	`?:`	条件运算符（三元运算符）	自右向左
14	`= += -= *= /= %= <<= >>= &= ^= |=`	赋值运算符	自右向左
15	`,`	逗号运算符	自左向右

C89 和 C99 标准的区别

此文档列出 C89 和 C99 两个标准的主要区别。

1、C99 新增特性

内容	说明
单杠注释	支持 // 注释（C89只支持 / /）
变量声明位置	允许在代码块中任意位置声明变量（C89要求在所有执行语句之前声明变量）
布尔类型	引入 `_Bool` 类型和 新增 `stdbool.h` 头文件
长整型扩展	新增 `long long int` 和 `stdint.h` 头文件
复数支持：	新增 `_Complex` 和 `complex.h` 头文件

2、数组增强

1. 变长数组，数组长度可以用运行时表达式定义。
2. 结构体/数组成员初始化，支持指定初始化列表，如: `struct point p1 = { .x = 1, .y = 2 };`。

3、 函数改进

1. 内联函数，支持 `inline` 关键字。
2. 受限指针，引入 `restrict` 关键字优化指针别名。

4、语法和库扩展

1. 复合字面量，如: `(struct point) {.x = 1, .y = 2}`。
2. 数学函数更新，如 增加 `sqrtf()`函数。（单精度版本）

C99 和 C11 标准的区别

此文档列出 C99 和 C11 两个标准的主要区别。

1、新增多线程支持

引入 `threads.h` 头文件，包括 `thrd_create()` 创建线程的函数 和 `mtx_t`（互斥锁类型）和 `cnd_t`（条件变量类型）。

2、静态断言

支持 _Static_assert（编译时断言）。如:

```
_Static_assert(sizeof(int) == 4, "int must be 4 bytes");
```

3、对齐控制

1. `_Alignas`，指定对齐要求。
2. `_Alignof`，获取对齐值。
3. 新增 `stdalign.h` 头文件，提供 `alignas`、`alignof` 宏。

4、Unicode支持增强

加入 UTF-8/16/32 字符串字面值，新增头文件 `uchar.h`（Unicode工具）如:

```
char u8str[] = u8"UTF-8字符串";
char16_t u16str[] = u"UTF-16字符串";
char32_t u32str[] = U"UTF-32字符串";
```

5、移除和弃用

1. 移除 `gets()` 函数， C11 正式从标准库移除这个不安全函数。
2. 变长数组改为可选，从 C99 的强制要求改为可选特性。

其它略!