C语言

1. C语言概述

1.1 C语言

一提到语言这个词语，自然会想到的是像英语、汉语等这样的自然语言，因为它是人和人交换信息不可缺少的工具。

而今天计算机遍布了我们生活的每一个角落，除了人和人的相互交流之外，我们必须和计算机角落。

用什么的什么样的方式和计算机做最直接的交流呢？人们自然想到的是最古老也最方便的方式——语言，而C语言就是人和计算机交流的一种语言。

语言是用来交流沟通的。有一方说，有另一方听，必须有两方参与，这是语言最重要的功能：

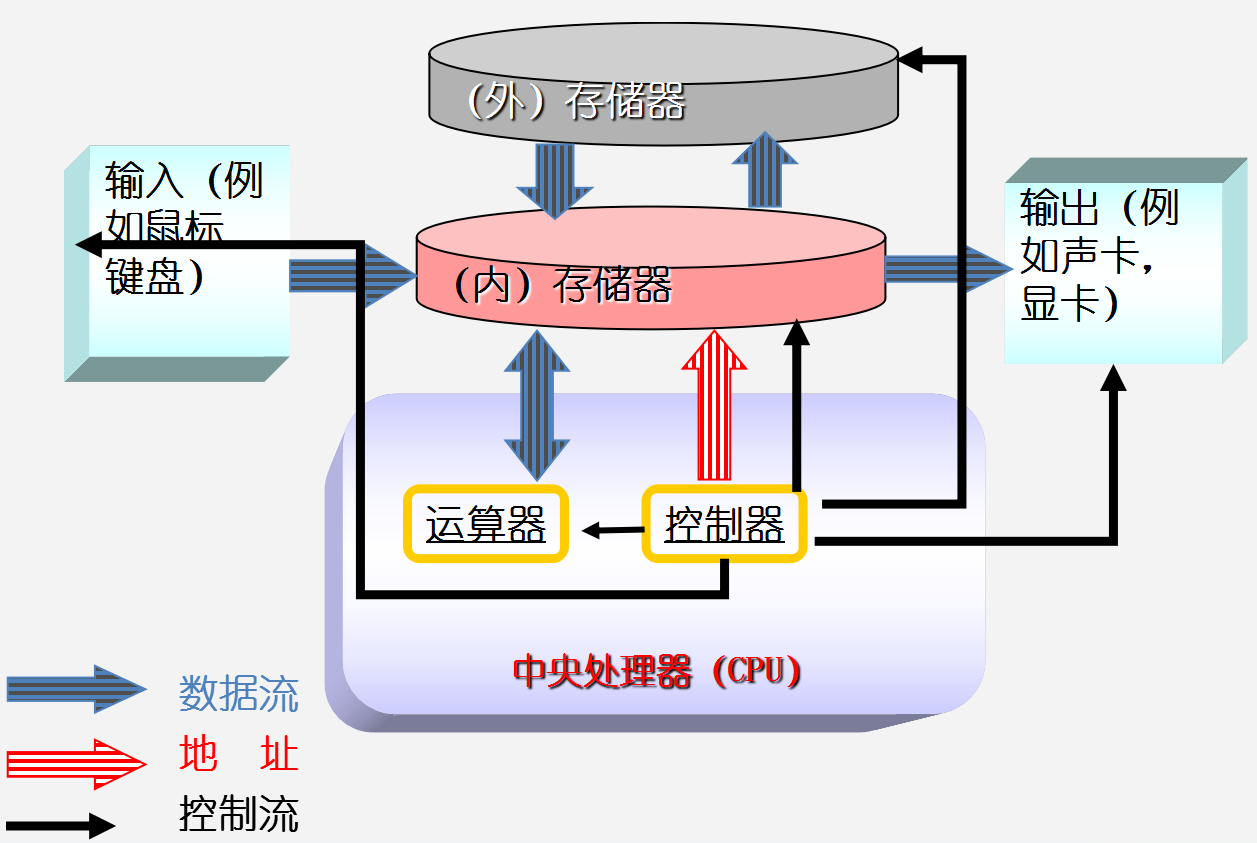
* 说的一方传递信息，听的一方接收信息；
* 说的一方下达指令，听的一方遵循命令做事情。

语言是人和人交流，C语言是人和机器交流。只是，人可以不听另外一个人，但是，计算机是无条件服从。

语言有独特的语法规则和定义，双方必须遵循这些规则和定义才能实现真正的交流。

* + 1. 计算机结构组成

寄存器



1.1.2 计算机系统组成

1.2 为什么要学习C语言

1.2.1 C语言特点

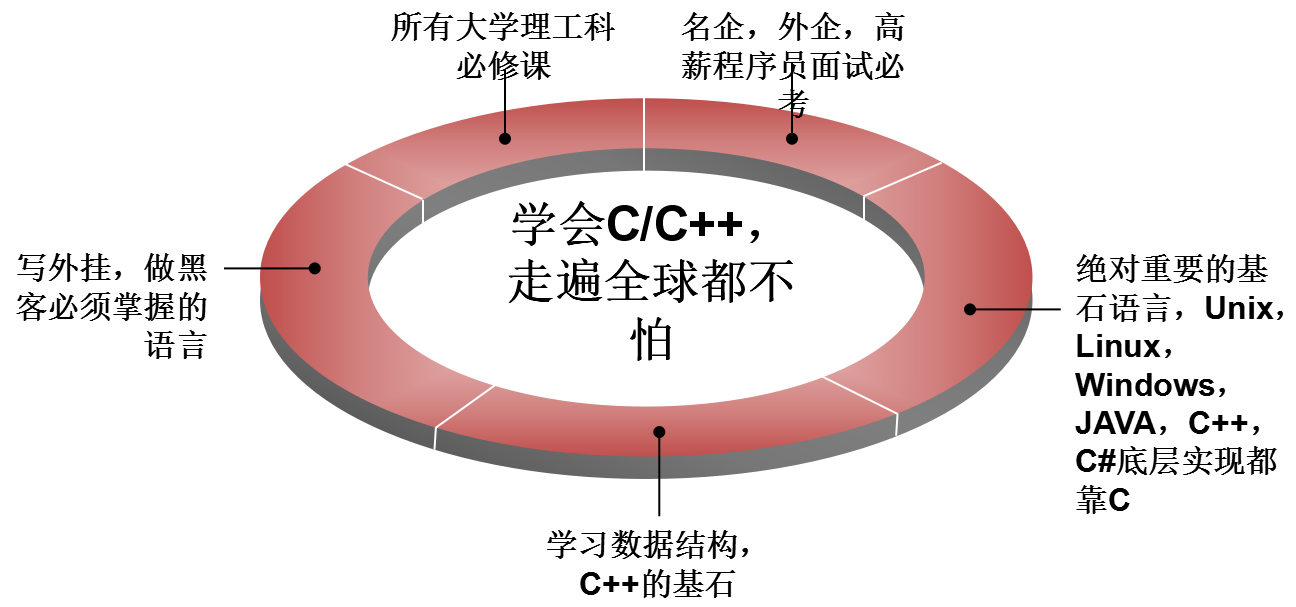
1) 优点

* + 代码量小
  + 执行速度快
  + 内存管理能力强
  + 可移植性强
  + 语言标准化

2) 缺点

* + 写代码实现周期长
  + 经验不足易出错
  + 对平台库依赖较多

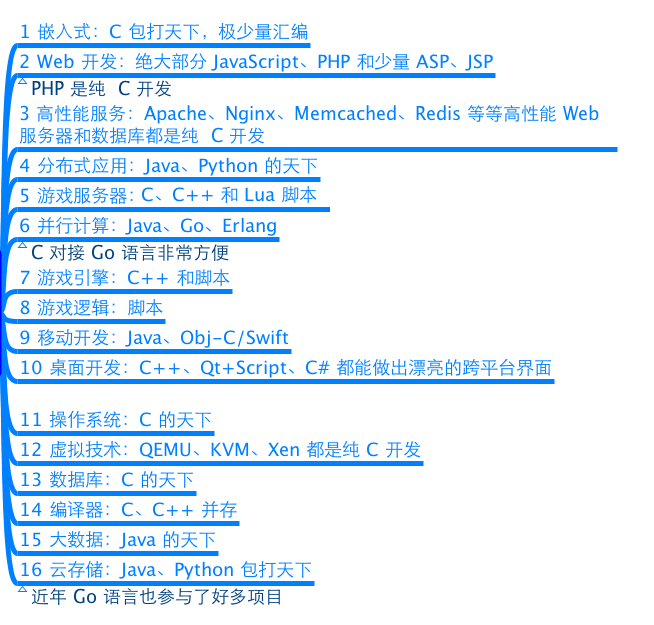
1.2.2 学习C语言理由



1.2.3 C语言应用领域

C语言的应用极其广泛，从网站后台，到底层操作系统，从多媒体应用到大型网络游戏，均可使用C语言来开发：

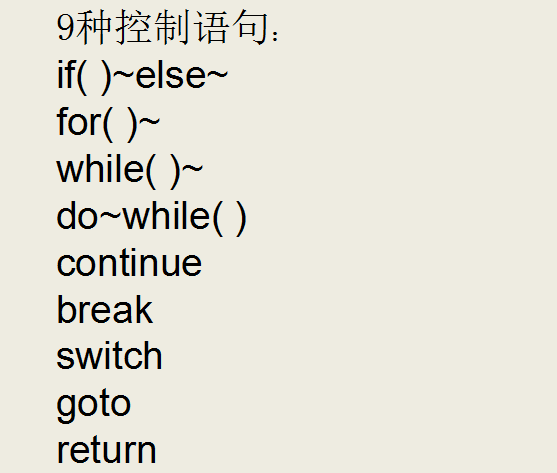
* C语言可以写网站后台程序
* C语言可以专门针对某个主题写出功能强大的程序库
* C语言可以写出大型游戏的引擎
* C语言可以写出另一个语言来
* C语言可以写操作系统和驱动程序，并且只能用C语言编写
* 任何设备只要配置了微处理器，就都支持C语言。从微波炉到手机，都是由C语言技术来推动的

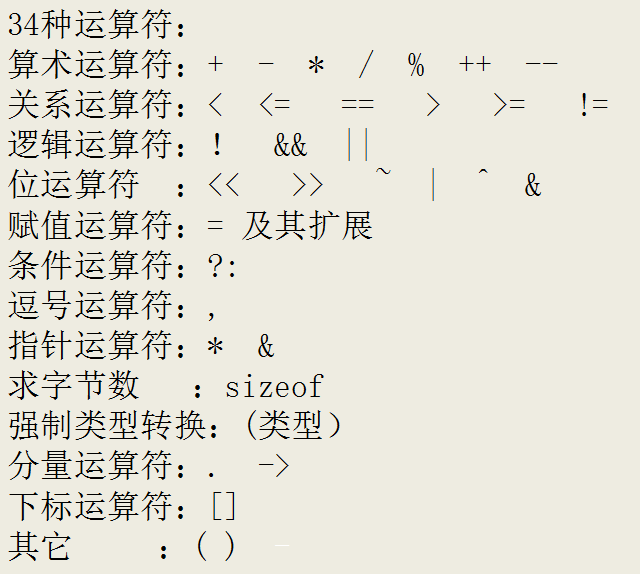


1.2.4 C语言关键字

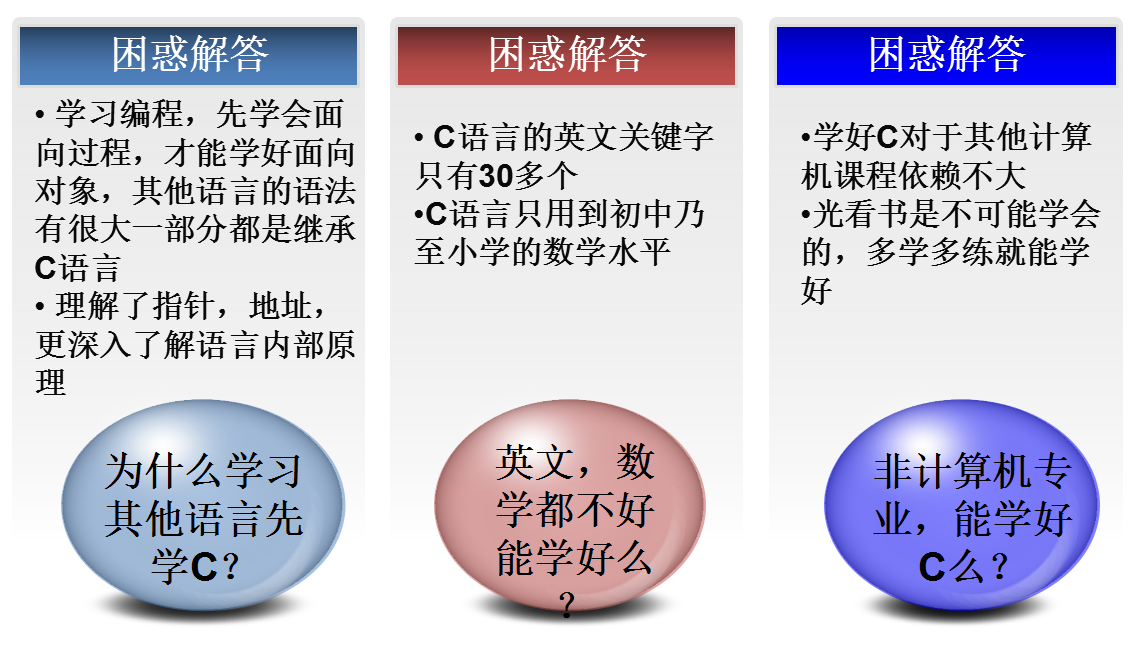
C语言仅有32个关键字，9种控制语句，34种运算符，却能完成无数的功能：







1.2.5 学习C常见困惑



1.3 第一个C语言程序：HelloWorld

1.3.1 编写C语言代码：hello.c

#include <stdio.h>

int *main*()

{

//这是第一个C语言代码

*printf*("hello world\n");

return 0;

}

C语言的源代码文件是一个普通的文本文件，但扩展名必须是.c。

1.3.2 通过gcc编译C代码

1) gcc编译器介绍

编辑器(如vi、记事本)是指我用它来写程序的（编辑代码），而我们写的代码语句，电脑是不懂的，我们需要把它转成电脑能懂的语句，编译器就是这样的转化工具。就是说，我们用编辑器编写程序，由编译器编译后才可以运行！

编译器是将易于编写、阅读和维护的高级计算机语言翻译为计算机能解读、运行的低级机器语言的程序。

gcc（GNU Compiler Collection，GNU 编译器套件），是由 GNU 开发的编程语言编译器。gcc原本作为GNU操作系统的官方编译器，现已被大多数类Unix操作系统（如Linux、BSD、Mac OS X等）采纳为标准的编译器，gcc同样适用于微软的Windows。

gcc最初用于编译C语言，随着项目的发展gcc已经成为了能够编译C、C++、Java、Ada、fortran、Object C、Object C++、Go语言的编译器大家族。

编译命令格式：

gcc [-option1] ... <filename>

g++ [-option1] ... <filename>

* 命令、选项和源文件之间使用空格分隔
* 一行命令中可以有零个、一个或多个选项
* 文件名可以包含文件的绝对路径，也可以使用相对路径
* 如果命令中不包含输出可执行文件的文件名，可执行文件的文件名会自动生成一个默认名，Linux平台为a.out，Windows平台为a.exe

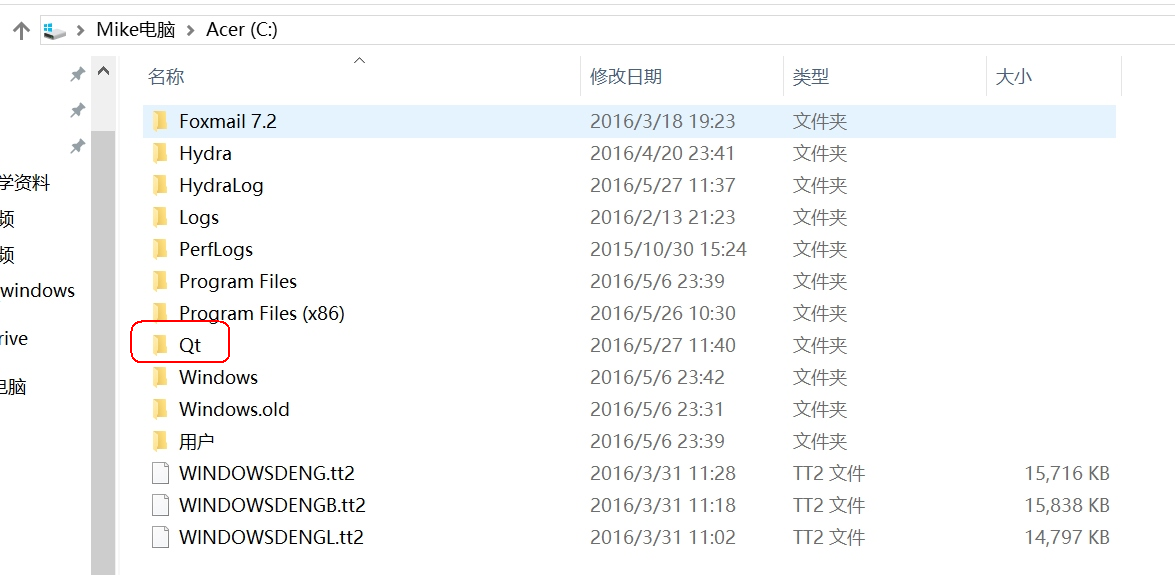
gcc、g++编译常用选项说明：

| **选项** | **含义** |
| --- | --- |
| -o file | 指定生成的输出文件名为file |
| -E | 只进行预处理 |
| -S(大写) | 只进行预处理和编译 |
| -c(小写) | 只进行预处理、编译和汇编 |

2) Windows平台下gcc环境配置

windows命令行界面下，默认是没有gcc编译器，我们需要配置一下环境。由于我们安装了Qt，Qt是一个集成开发环境，内部集成gcc编译器，配置一下环境变量即可使用gcc。

1. 进入Qt安装目录：



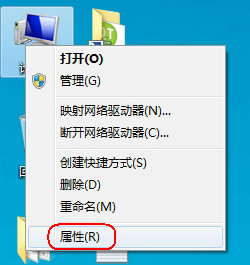
1. 拷贝gcc所在的路径（不同安装目录有所差异）

C:\Qt\Qt5.5.0\Tools\mingw492\_32\bin

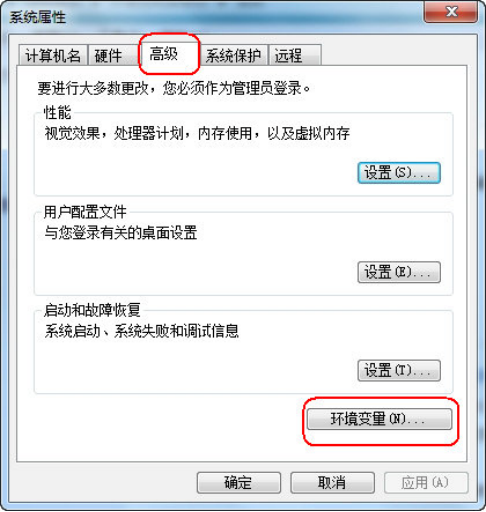


c) 设置windows环境变量

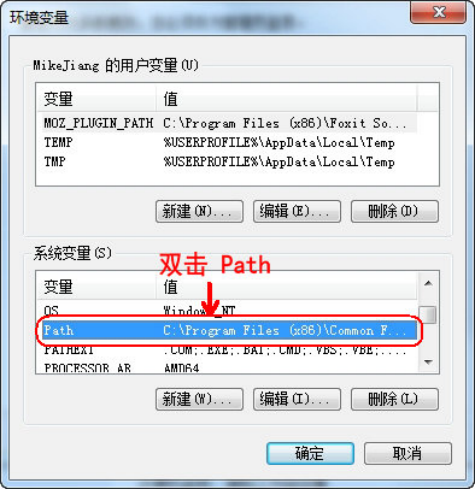
计算机（右击）－> 属性：



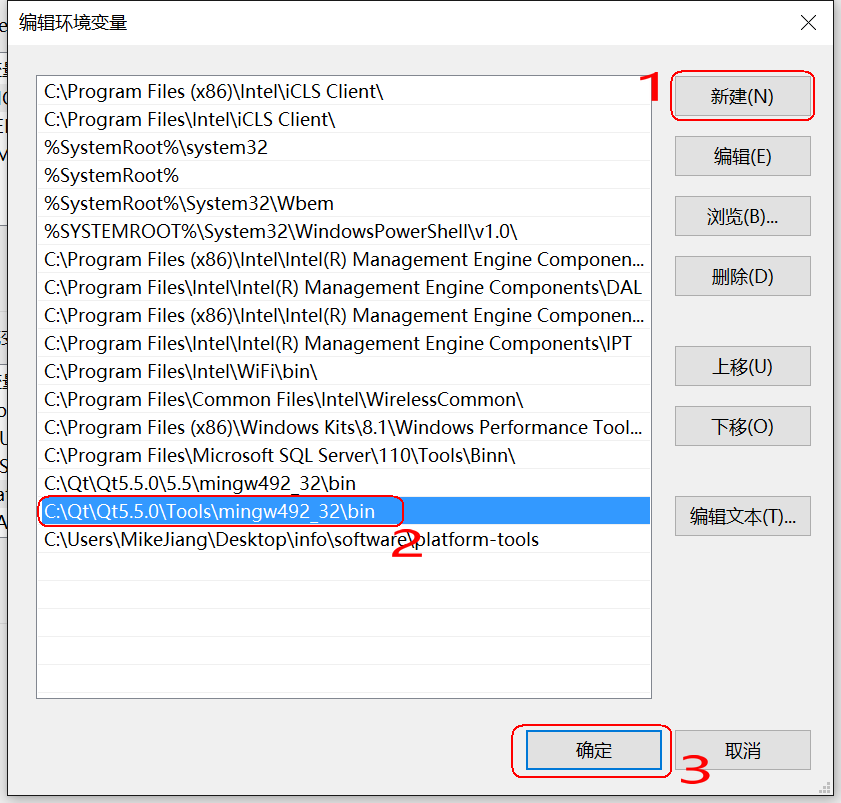




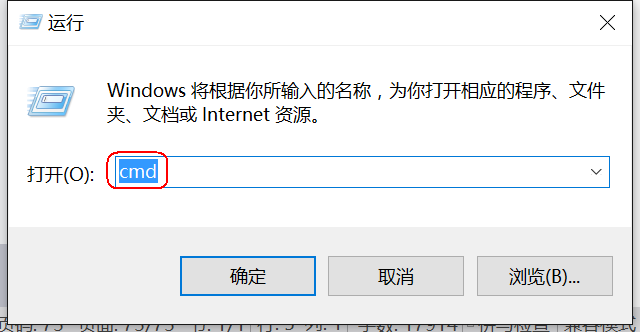
如果是win7，双击Path后，把拷贝的路径追加到后面，路径之间需要英文” **;** ”分隔：

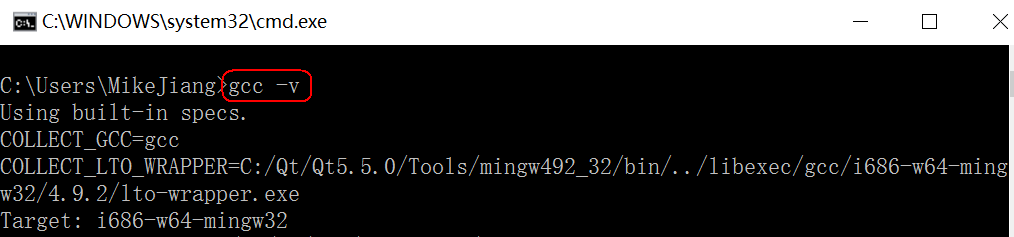


如果是win10，界面做得友好一下，新建添加路径即可：

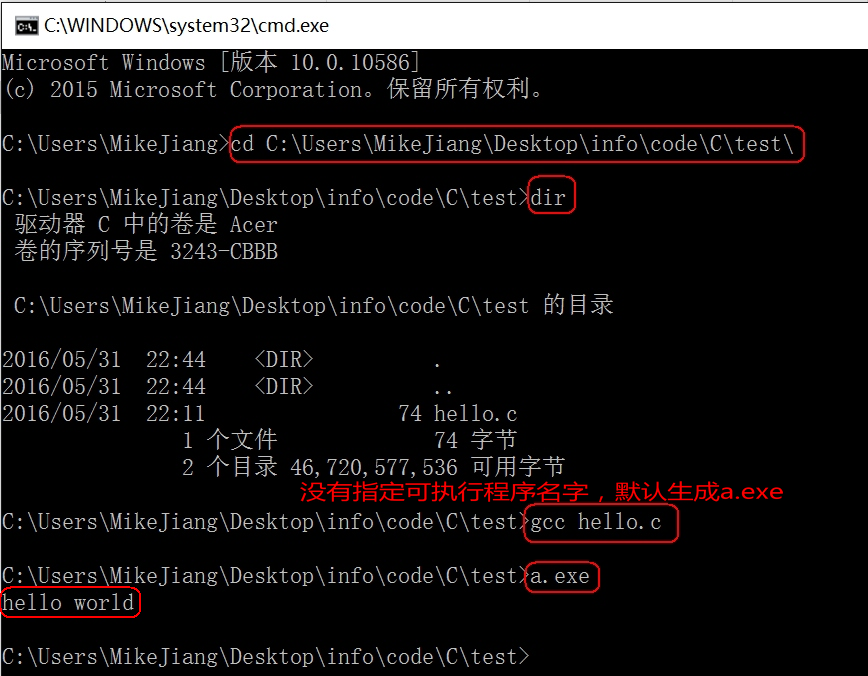


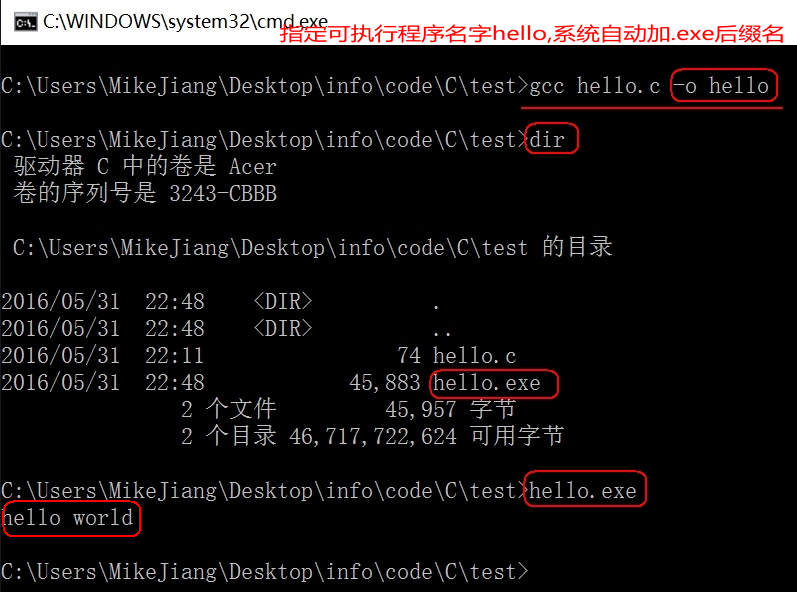
d)测试gcc命令：





3) Windows平台下编译代码





1.3.3 代码分析

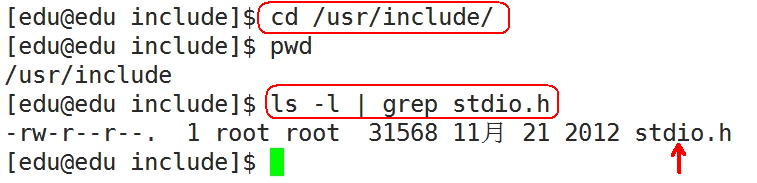
1) include头文件包含

* #include的意思是头文件包含，#include <stdio.h>代表包含stdio.h这个头文件
* 使用C语言库函数需要提前包含库函数对应的头文件，如这里使用了printf()函数，需要包含stdio.h头文件
* 可以通过man 3 printf查看printf所需的头文件

#include< > 与 #include ""的区别：

* < > 表示系统直接按系统指定的目录检索
* "" 表示系统先在 "" 指定的路径(没写路径代表当前路径)查找头文件，如果找不到，再按系统指定的目录检索

stdio.h是在操作系统的系统目录下：



2) main函数

* 一个完整的C语言程序，是由一个、且只能有一个main()函数(又称主函数，必须有)和若干个其他函数结合而成（可选）。
* main函数是C语言程序的入口，程序是从main函数开始执行。

3) {} 括号，程序体和代码块

* {}叫代码块，一个代码块内部可以有一条或者多条语句
* C语言每句可执行代码都是"**;**"分号结尾
* 所有的#开头的行，都代表预编译指令，预编译指令行结尾是没有分号的
* 所有的可执行语句必须是在代码块里面

4) 注释

* //叫行注释，注释的内容编译器是忽略的，注释主要的作用是在代码中加一些说明和解释，这样有利于代码的阅读
* /\*\*/叫块注释
* 块注释是C语言标准的注释方法
* 行注释是从C++语言借鉴过来的

5) printf函数

* printf是C语言库函数，功能是向标准输出设备输出一个字符串
* printf(“hello world\n”);//\n的意思是回车换行

6) return语句

* return代表函数执行完毕，返回return代表函数的终止
* 如果main定义的时候前面是int，那么return后面就需要写一个整数；如果main定义的时候前面是void，那么return后面什么也不需要写
* 在main函数中return 0代表程序执行成功，return -1代表程序执行失败
* int main()和void main()在C语言中是一样的，但C++只接受int main这种定义方式

1.4 system函数

1.4.1 system函数的使用

#include <stdlib.h>

int *system*(const char \*command);

功能：在已经运行的程序中执行另外一个外部程序

参数：外部可执行程序名字

返回值：

成功：0

失败：任意数字

示例代码：

#include <stdio.h>

#include <stdlib.h>

int *main*()

{

//system("calc"); // mspaint画图、notepad记事本、cls清屏

return 0;

}

1.5 C语言编译过程

1.5.1 C程序编译步骤

C代码编译成可执行程序经过4步：

1. 预处理：宏定义展开、头文件展开、条件编译等，同时将代码中的注释删除，这里并不会检查语法
2. 编译：逐行检查语法错误（最耗时的一步），将预处理后文件编译生成汇编文件
3. 汇编：将汇编文件生成（翻译）目标文件(二进制文件)
4. 链接：C语言写的程序是需要依赖各种库的，所以编译之后还需要把库链接到最终的可执行程序中去



1.5.2 gcc编译过程

1) 分步编译

预处理：gcc -E hello.c -o hello.i

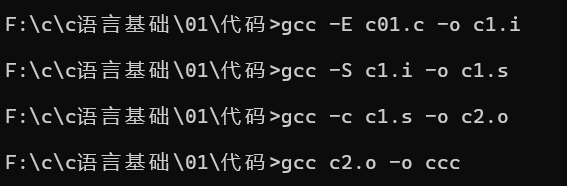
编 译：gcc -S hello.i -o hello.s

汇 编：gcc -c hello.s -o hello.o

链 接：gcc hello.o -o hello

| **选项** | **含义** |
| --- | --- |
| -E | 只进行预处理 |
| -S(大写) | 只进行预处理和编译 |
| -c(小写) | 只进行预处理、编译和汇编 |
| -o file | 指定生成的输出文件名为 file |

| **文件后缀** | **含义** |
| --- | --- |
| .c | C 语言文件 |
| .i | 预处理后的 C 语言文件 |
| .s | 编译后的汇编文件 |
| .o | 编译后的目标文件 |

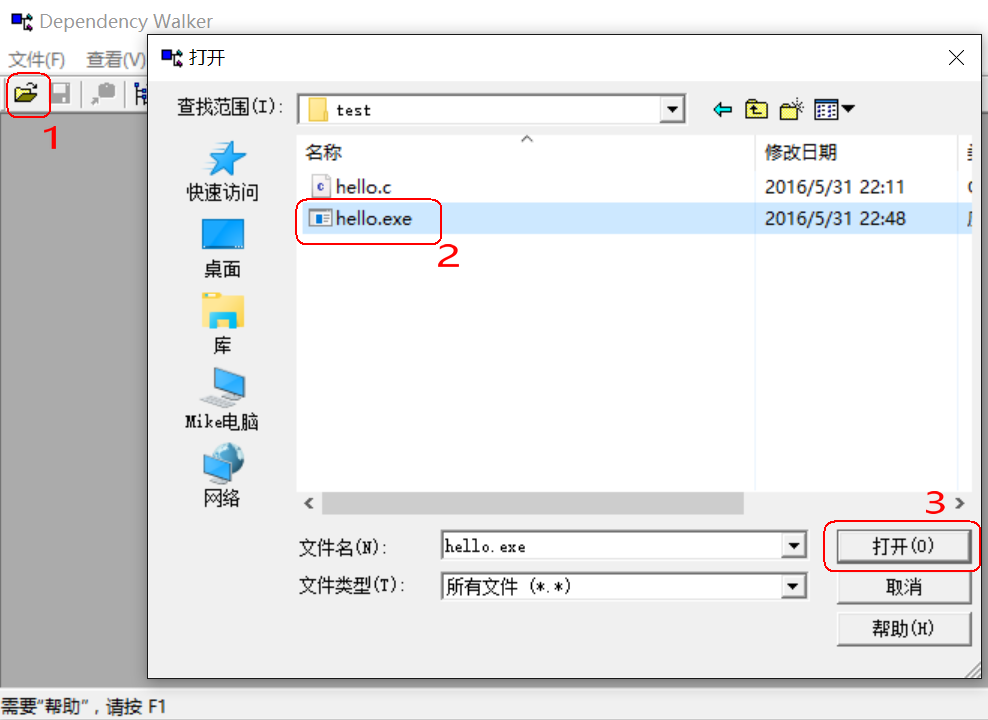


2) 一步编译

gcc hello.c -o demo（还是经过：预处理、编译、汇编、链接的过程）：

1.5.3 查找程序所依赖的动态库

Windows平台下，需要相应软件(Depends.exe)：





1.6 CPU内部结构与寄存器(了解)

1.6.1 64位和32位系统区别

* 寄存器是CPU内部最基本的存储单元
* CPU对外是通过总线(地址、控制、数据)来和外部设备交互的，总线的宽度是8位，同时CPU的寄存器也是8位，那么这个CPU就叫8位CPU
* 如果总线是32位，寄存器也是32位的，那么这个CPU就是32位CPU
* 有一种CPU内部的寄存器是32位的，但总线是16位，准32为CPU
* 所有的64位CPU兼容32位的指令，32位要兼容16位的指令，所以在64位的CPU上是可以识别32位的指令
* 在64位的CPU构架上运行了64位的软件操作系统，那么这个系统是64位
* 在64位的CPU构架上，运行了32位的软件操作系统，那么这个系统就是32位
* 64位的软件不能运行在32位的CPU之上

1.6.2 寄存器名字(了解)

| **8位** | **16位** | **32位** | **64位** |
| --- | --- | --- | --- |
| A | AX | EAX | RAX |
| B | BX | EBX | RBX |
| C | CX | ECX | RCX |
| D | DX | EDX | RDX |

1.6.3 寄存器、缓存、内存三者关系

按与CPU远近来分，离得最近的是寄存器，然后缓存(CPU缓存)，最后内存。

CPU计算时，先预先把要用的数据从硬盘读到内存，然后再把即将要用的数据读到寄存器。于是 CPU<--->寄存器<--->内存，这就是它们之间的信息交换。

那为什么有缓存呢？因为如果经常操作内存中的同一址地的数据，就会影响速度。于是就在寄存器与内存之间设置一个缓存。

因为从缓存提取的速度远高于内存。当然缓存的价格肯定远远高于内存，不然的话，机器里就没有内存的存在。

由此可以看出，从远近来看：CPU〈---〉寄存器〈---> 缓存 <---> 内存--硬盘、外存

1.7 汇编语言

1.7.1 VS中C语言嵌套汇编代码(了解)

#include <stdio.h>

int *main*()

{

//定义整型变量a, b, c

int a;

int b;

int c;

\_\_asm

{

mov a, 3 //3的值放在a对应内存的位置

mov b, 4 //4的值放在b对应内存的位置

mov eax, a //把a内存的值放在eax寄存器

*add* eax, b //eax和b相加，结果放在eax

mov c, eax //eax的值放在c中

}

*printf*("%d\n", c);//把c的值输出

return 0;//成功完成

}

1.7.2 C语言实现

#include <stdio.h>

int *main*()

{

//定义整型变量a, b, c

int a;

int b;

int c;

a = 3;

b = 4;

c = a + b;

// = 赋值

*printf*("%d\n", c);//把c的值输出

return 0;//成功完成

}

1.8 集成开发环境IDE

集成开发环境（IDE，Integrated Development Environment ）是用于提供程序开发环境的应用程序，一般包括代码编辑器、编译器、调试器和图形用户界面工具。集成了代码编写功能、分析功能、编译功能、调试功能等一体化的开发软件服务套。所有具备这一特性的软件或者软件套（组）都可以叫集成开发环境。

1.8.1 Qt Creator

Qt Creator是跨平台的 Qt IDE， Qt Creator 是 Qt 被 Nokia 收购后推出的一款新的轻量级集成开发环境（IDE）。此 IDE 能够跨平台运行，支持的系统包括 Linux（32 位及 64 位）、Mac OS X 以及 Windows。根据官方描述，Qt Creator 的设计目标是使开发人员能够利用 Qt 这个应用程序框架更加快速及轻易的完成开发任务。

| **快捷键** | **含义** |
| --- | --- |
| Ctrl + i | 自动格式化代码 |
| Ctrl + / | 注释/取消注释 |
| Alt + Enter | 自动完成类函数定义 |
| F4 | .h 文件和对应.cpp 文件切换 |
| F9 | 设置断点 |
| F5 | 调试运行 |
| Ctrl + r | 编译，但不调试运行 |
| Ctrl + b | 编译，不运行 |
| F10 | next调试 |
| F11 | step调试 |

1.8.2 Microsoft Visual Studio

Microsoft Visual Studio（简称VS）是美国微软公司的开发工具包系列产品。VS是一个基本完整的开发工具集，它包括了整个软件生命周期中所需要的大部分工具，如UML工具、代码管控工具、集成开发环境(IDE)等等,所写的目标代码适用于微软支持的所有平台。Visual Studio是目前最流行的Windows平台应用程序的集成开发环境。

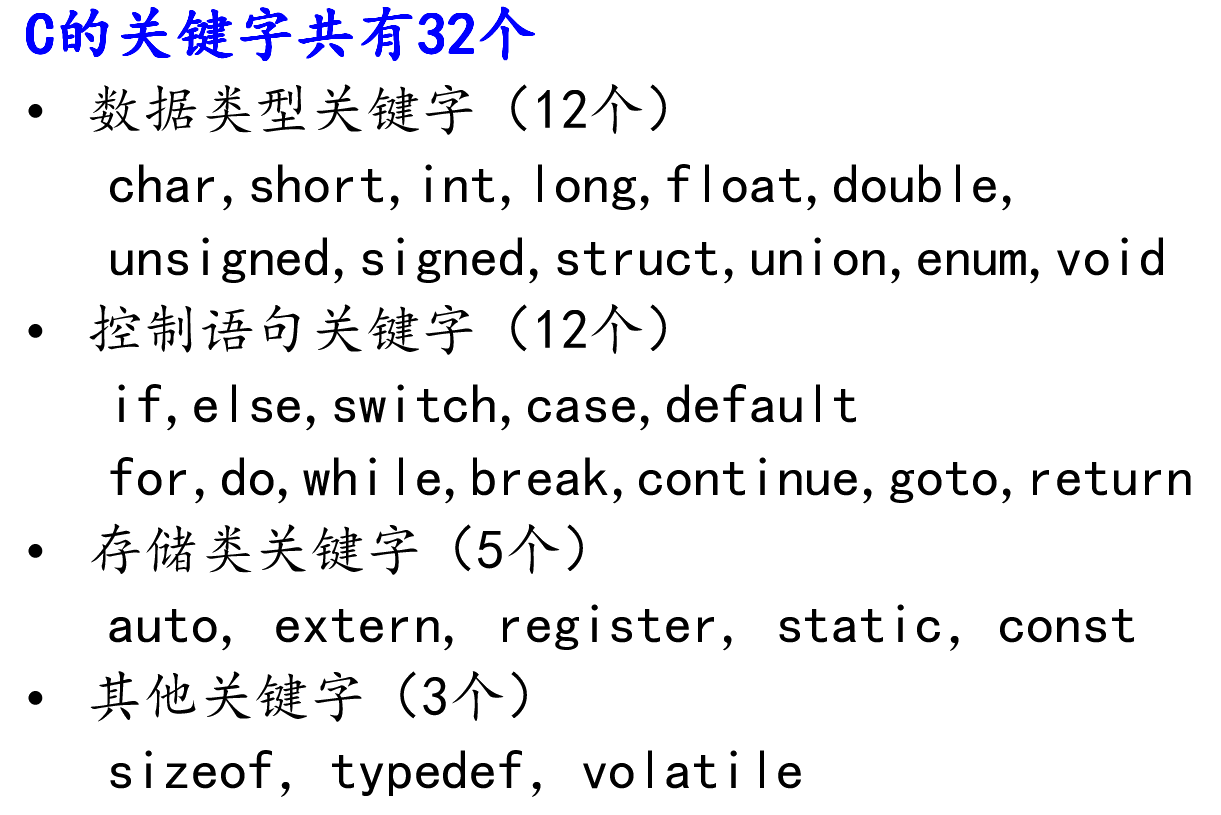
1) VS常用快捷键

| **快捷键** | **含义** |
| --- | --- |
| Ctrl + k,Ctrl + f | 自动格式化代码 |
| Ctrl + k,Ctrl + c | 注释代码 ctrl + shift + / |
| Ctrl + k,Ctrl + u | 取消注释代码 |
| F9 | 设置断点 |
| F5 | 调试运行 |
| Ctrl + F5 | 不调试运行 |
| Ctrl + Shift + b | 编译，不运行 |
| F10 | next调试 |
| F11 | step调试 |

2. 数据类型

2.1 常量与变量

2.1.1 关键字



2.1.2 常量

常量：

* 在程序运行过程中，其值不能被改变的量
* 常量一般出现在表达式或赋值语句中

|  |  |
| --- | --- |
| 整型常量 | 100，200，-100，0 |
| 实型、浮点常量 | 3.14 ， 0.125，-3.123 |
| 字符型常量 | ‘a’,‘b’,‘1’,‘\n’ |
| 字符串常量 | “a”,“ab”，“12356” |

2.1.3 变量

1) 变量

变量：

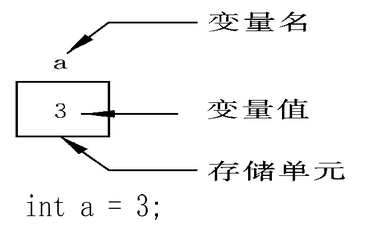
* 在程序运行过程中，其值可以改变
* 变量在使用前必须先定义，定义变量前必须有相应的数据类型

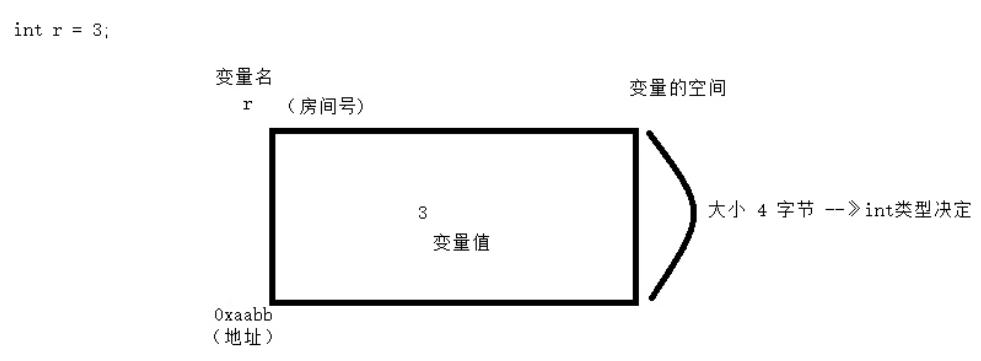
语法：

* 类型名 变量名 = 变量值;
* int a = 3;

变量特点：

* 变量在编译时为其分配相应的内存空间
* 可以通过其名字和地址访问相应内存





标识符(变量名字)命名规则：

* 标识符不能是关键字
* 标识符只能由字母、数字、下划线组成
* 第一个字符必须为字母或下划线（不能数字开头）
* 标识符中字母区分大小写

2) 声明和定义区别

* 声明变量不需要建立存储空间，如：extern int a;
* 定义变量需要建立存储空间，如：int b;

#include <stdio.h>

int *main*()

{

//extern 关键字只做声明，不能做任何定义，后面还会学习，这里先了解

//声明一个变量a，a在这里没有建立存储空间

extern int a;

a = 10; //err, 没有空间，就不可以赋值

int b = 10; //定义一个变量b，b的类型为int，b赋值为10

return 0;

}

从广义的角度来讲声明中包含着定义，即定义是声明的一个特例，所以并非所有的声明都是定义：

* int b 它既是声明，同时又是定义
* 对于 extern b来讲它只是声明不是定义

一般的情况下，把建立存储空间的声明称之为“定义”（叫定义或声明都可以）而把不需要建立存储空间的声明称之为“声明”。

2.1.4 使用示例

#include <stdio.h>

#define MAX 10 //声明了一个常量，名字叫MAX，值是10，常量的值一旦初始化不可改

int *main*()

{

int a; //定义了一个变量，其类型为int，名字叫a

const int b = 10; //定义一个const常量，名为叫b，值为10

//b = 11; //err,常量的值不能改变

//MAX = 100; //err,常量的值不能改变

a = MAX;

a = 123;

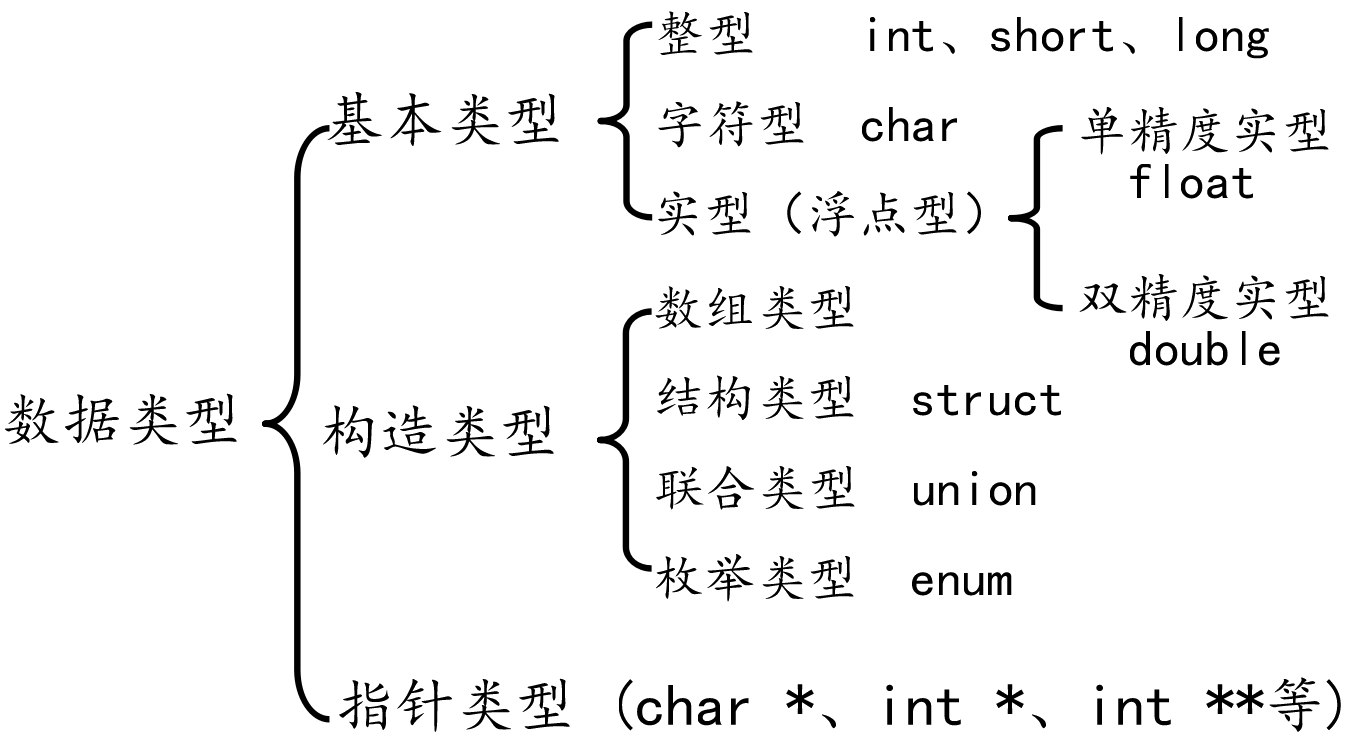
*printf*("%d\n", a); //打印变量a的值

return 0;

}

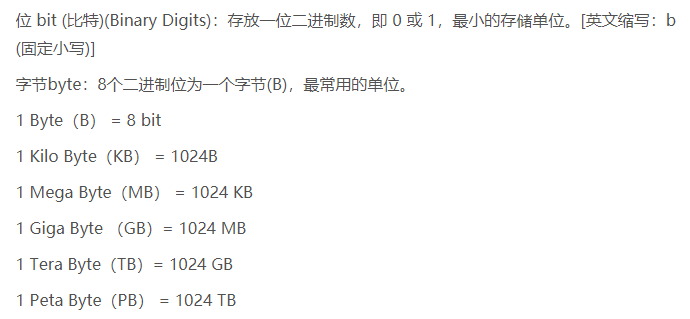
2.1.5 数据类型

数据类型的作用：编译器预算对象（变量）分配的内存空间大小。



2.2整型：int

2.2.1 计算机常用单位



2.2.2 short、int、long、long long

| **数据类型** | **占用空间** |
| --- | --- |
| short(短整型) | 2字节 |
| int(整型) | 4字节 |
| long(长整形) | Windows为4字节，Linux为4字节(32位)，8字节(64位) |
| long long(长长整形) | 8字节 |

| **整型常量（了解）** | **所需类型** |
| --- | --- |
| 10 | 代表int类型 |
| 10l, 10L | 代表long类型 |
| 10ll, 10LL | 代表long long类型 |
| 10u, 10U | 代表unsigned int类型 |
| 10ul, 10UL | 代表unsigned long类型 |
| 10ull, 10ULL | 代表unsigned long long类型 |

| **打印格式** | **含义** |
| --- | --- |
| %hd | 输出short类型 |
| %d | 输出int类型 |
| %ld | 输出long类型 |
| %lld | 输出long long类型 |
| %hu | 输出unsigned short类型 |
| %u | 输出unsigned int类型 |
| %lu | 输出unsigned long类型 |
| %llu | 输出unsigned long long类型 |

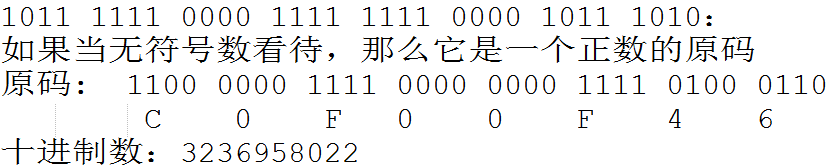
2.2.4 有符号数和无符号数区别

1) 有符号数

有符号数是最高位为符号位，0代表正数，1代表负数。

2) 无符号数

无符号数最高位不是符号位，而就是数的一部分，无符号数不可能是负数。



当我们写程序要处理一个不可能出现负值的时候，一般用无符号数，这样可以增大数的表达最大值。

3) 有符号和无符号整型取值范围

| **数据类型** | **占用空间** | **取值范围** |
| --- | --- | --- |
| short | 2字节 | -32768 到 32767 (-215 ~ 215-1) |
| int | 4字节 | -2147483648 到 2147483647 (-231 ~ 231-1) |
| long | 4字节 | -2147483648 到 2147483647 (-231 ~ 231-1) |
| unsigned short | 2字节 | 0 到 65535 (0 ~ 216-1) |
| unsigned int | 4字节 | 0 到 4294967295 (0 ~ 232-1) |
| unsigned long | 4字节 | 0 到 4294967295 (0 ~ 232-1) |

2.3 sizeof关键字

* sizeof不是函数，所以不需要包含任何头文件，它的功能是计算一个数据类型的大小，单位为字节
* sizeof的返回值为size\_t，属于一种无符号整数类型

#include <stdio.h>

int *main*()

{

int a;

int b = sizeof(a);//sizeof得到指定值占用内存的大小，单位：字节

*printf*("b = %d\n", b);

*size\_t* c = sizeof(a);

*printf*("c = %u\n", c);//用无符号数的方式输出c的值

return 0;

}

2.4字符型：char

2.4.1 字符变量的定义和输出

字符型变量用于存储一个单一字符，在 C 语言中用 char 表示，其中每个字符变量都会占用 1 个字节。在给字符型变量赋值时，需要用一对英文半角格式的单引号(' ')把字符括起来。

字符变量实际上并不是把该字符本身放到变量的内存单元中去，而是将该字符对应的 ASCII 编码放到变量的存储单元中。char的本质就是一个1字节大小的整型。

#include <stdio.h>

int *main*()

{

char char\_a = 'a';

printf("打印%d\n", char\_a); // 打印%d,把对应的数值打印出来

printf("打印字符%c\n", char\_a); // %c才是对应的字符型变量的值

// 定义个大写的M（字符型）。打印出对用的小写字母m

char ch\_m = 'M';

printf("打印对应的小写字母%c", ch\_m + 32);

return 0;

}

2.4.2 ASCII对照表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **ASCII值** | **控制字符** | **ASCII值** | **字符** | **ASCII值** | **字符** | **ASCII值** | **字符** |
| 0 | NUT | 32 | (space) | 64 | @ | 96 | 、 |
| 1 | SOH | 33 | ! | 65 | A | 97 | a |
| 2 | STX | 34 | " | 66 | B | 98 | b |
| 3 | ETX | 35 | # | 67 | C | 99 | c |
| 4 | EOT | 36 | $ | 68 | D | 100 | d |
| 5 | ENQ | 37 | % | 69 | E | 101 | e |
| 6 | ACK | 38 | & | 70 | F | 102 | f |
| 7 | BEL | 39 | , | 71 | G | 103 | g |
| 8 | BS | 40 | ( | 72 | H | 104 | h |
| 9 | HT | 41 | ) | 73 | I | 105 | i |
| 10 | LF | 42 | \* | 74 | J | 106 | j |
| 11 | VT | 43 | + | 75 | K | 107 | k |
| 12 | FF | 44 | , | 76 | L | 108 | l |
| 13 | CR | 45 | - | 77 | M | 109 | m |
| 14 | SO | 46 | . | 78 | N | 110 | n |
| 15 | SI | 47 | / | 79 | O | 111 | o |
| 16 | DLE | 48 | 0 | 80 | P | 112 | p |
| 17 | DCI | 49 | 1 | 81 | Q | 113 | q |
| 18 | DC2 | 50 | 2 | 82 | R | 114 | r |
| 19 | DC3 | 51 | 3 | 83 | S | 115 | s |
| 20 | DC4 | 52 | 4 | 84 | T | 116 | t |
| 21 | NAK | 53 | 5 | 85 | U | 117 | u |
| 22 | SYN | 54 | 6 | 86 | V | 118 | v |
| 23 | TB | 55 | 7 | 87 | W | 119 | w |
| 24 | CAN | 56 | 8 | 88 | X | 120 | x |
| 25 | EM | 57 | 9 | 89 | Y | 121 | y |
| 26 | SUB | 58 | : | 90 | Z | 122 | z |
| 27 | ESC | 59 | ; | 91 | [ | 123 | { |
| 28 | FS | 60 | < | 92 | / | 124 | | |
| 29 | GS | 61 | = | 93 | ] | 125 | } |
| 30 | RS | 62 | > | 94 | ^ | 126 | ` |
| 31 | US | 63 | ? | 95 | \_ | 127 | DEL |

ASCII 码大致由以下两部分组成：

* ASCII 非打印控制字符： ASCII 表上的数字 0-31 分配给了控制字符，用于控制像打印机等一些外围设备。
* ASCII 打印字符：数字 32-126 分配给了能在键盘上找到的字符，当查看或打印文档时就会出现。数字 127 代表 Del 命令。

2.4.3 转义字符

| **转义字符** | **含义** | **ASCII码值（十进制）** |
| --- | --- | --- |
| \a | 警报 | 007 |
| \b | 退格(BS) ，将当前位置移到前一列 | 008 |
| \f | 换页(FF)，将当前位置移到下页开头 | 012 |
| \n | 换行(LF) ，将当前位置移到下一行开头 | 010 |
| \r | 回车(CR) ，将当前位置移到本行开头 | 013 |
| \t | 水平制表(HT) （跳到下一个TAB位置） | 009 |
| \v | 垂直制表(VT) | 011 |
| \\ | 代表一个反斜线字符"\" | 092 |
| \' | 代表一个单引号（撇号）字符 | 039 |
| \" | 代表一个双引号字符 | 034 |
| \? | 代表一个问号 | 063 |
| \0 | 数字0 | 000 |
| \ddd | 8进制转义字符，d范围0~7 | 3位8进制 |
| \xhh | 16进制转义字符，h范围0~9，a~f，A~F | 3位16进制 |

注意：红色字体标注的为不可打印字符。

2.5实型(浮点型)：float、double

实型变量也可以称为浮点型变量，浮点型变量是用来存储小数数值的。在C语言中， 浮点型变量分为两种： 单精度浮点数(float)、 双精度浮点数(double)， 但是double型变量所表示的浮点数比 float 型变量更精确。

由于浮点型变量是由有限的存储单元组成的，因此只能提供有限的有效数字。在有效位以外的数字将被舍去，这样可能会产生一些误差。

不以f结尾的常量是double类型，以f结尾的常量(如3.14f)是float类型。

#include <stdio.h>

int *main*()

{

//传统方式赋值

float a = 3.14f; //或3.14F

double b = 3.14;

*printf*("a = %f\n", a);

*printf*("b = %lf\n", b);

//科学法赋值

a = 3.2e3f; //3.2\*1000 = 3200，e可以写E

*printf*("a1 = %f\n", a);

a = 100e-3f; //100\*0.001 = 0.1

*printf*("a2 = %f\n", a);

a = 3.1415926f;

*printf*("a3 = %f\n", a); //结果为3.141593

return 0;

}

2.6 进制

进制也就是进位制，是人们规定的一种进位方法。 对于任何一种进制—X进制，就表示某一位置上的数运算时是逢X进一位。 十进制是逢十进一，十六进制是逢十六进一，二进制就是逢二进一，以此类推，x进制就是逢x进位。

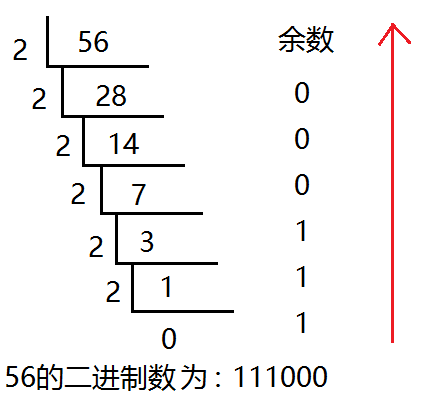
| **十进制** | **二进制** | **八进制** | **十六进制** |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 2 | 10 | 2 | 2 |
| 3 | 11 | 3 | 3 |
| 4 | 100 | 4 | 4 |
| 5 | 101 | 5 | 5 |
| 6 | 110 | 6 | 6 |
| 7 | 111 | 7 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |
| 16 | 10000 | 20 | 10 |

2.6.1 二进制

二进制是计算技术中广泛采用的一种数制。二进制数据是用0和1两个数码来表示的数。它的基数为2，进位规则是“逢二进一”，借位规则是“借一当二”。

当前的计算机系统使用的基本上是二进制系统。

十进制转化二进制的方法：用十进制数除以2，分别取余数和商数，商数为0的时候，将余数倒着数就是转化后的结果。



二进制转十进制：乘权求和(任何数的0次幂都等于1)

110101011

1\*2^8+1\*2^7+1\*2^5+1\*2^3+1\*2^1+1\*2^0

1 + 2 + 8 + 32 + 128 + 256 = 427



2.6.2 八进制

八进制，Octal，缩写OCT或O，一种以8为基数的计数法，采用0，1，2，3，4，5，6，7八个数字，逢八进1。一些编程语言中常常以数字0开始表明该数字是八进制。

八进制的数和二进制数可以按位对应（八进制一位对应二进制三位），因此常应用在计算机语言中。

八进制转十进制：乘权求和

0753

3 \* 8^0 = 3 \* 1 = 3

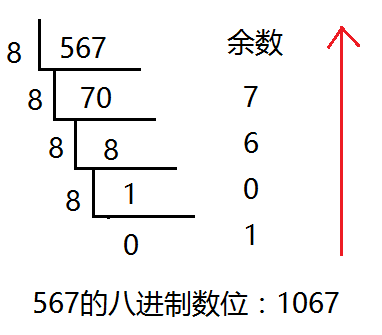
5 \* 8^1 = 5 \* 8 = 40

7 \* 8^2 = 7 \* 64 = 448

491

十进制转化八进制的方法：

用十进制数除以8，分别取余数和商数，商数为0的时候，将余数倒着数就是转化后的结果。



八进制转二进制：将八进制数的毎一位转换成二进制的3位，最后组合起来

八进制0对应二进制000，

八进制1对应二进制001，

八进制2对应二进制010，

八进制3对应二进制011，

八进制4对应二进制100，

八进制5对应二进制101，

八进制6对应二进制110，

八进制7对应二进制111。

例：八进制数0531转换为二进制数就是 101 011 001

二进制转八进制：

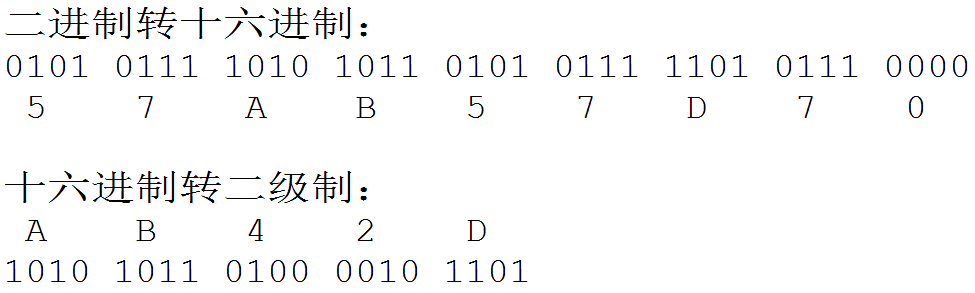
将整数部分从低位向高位每三位用一个等值的八进制数来替换,最后不足三位时在高位补0凑满三位

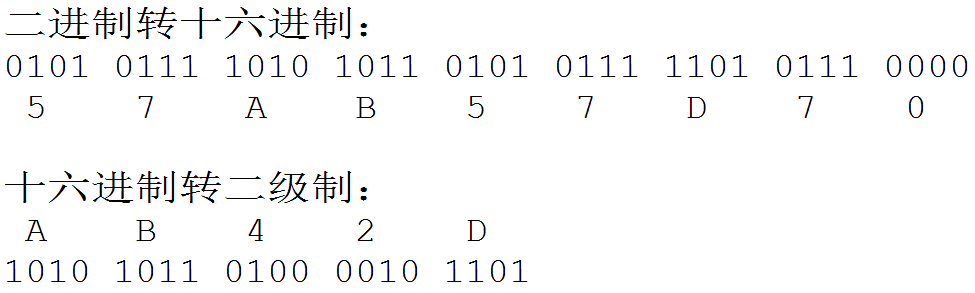
001 010 111 010 110 012726

2.6.3 十六进制

十六进制（英文名称：Hexadecimal），同我们日常生活中的表示法不一样，它由0-9，A-F组成，字母不区分大小写。与10进制的对应关系是：0-9对应0-9，A-F对应10-15。以0x开头

十六进制的数和二进制数可以按位对应（十六进制一位对应二进制四位），因此常应用在计算机语言中。





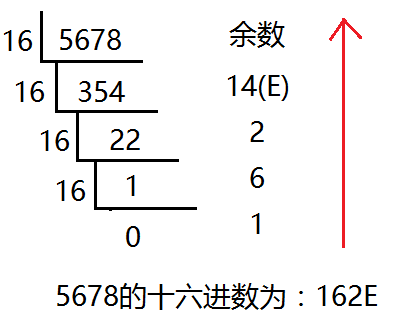
十六进制转十进制：乘权求和

0x13F

1\*16^2 + 3\*16^1 + 15 319

十进制转化十六进制的方法：

用十进制数除以16，分别取余数和商数，商数为0的时候，将余数倒着数就是转化后的结果。



# 十六进制转化成八进制：

方法1：十六进制转化为二进制，再转化为八进制。

方法2：十六进制先转化为十进制，再转化为八进制。

# 八进制转化成十六进制：

方法1：八进制转化为二进制，再转化为十六进制。

方法2：八进制先转化为十进制，再转化为十六进制。

2.6.4 C语言如何表示相应进制数

|  |  |
| --- | --- |
| 十进制 | 以正常数字1-9开头，如123 |
| 八进制 | 以数字0开头，如0123 |
| 十六进制 | 以0x开头，如0x123 |
| 二进制 | C语言不能直接书写二进制数 |

#include <stdio.h>

int *main*()

{

int a = 123; //十进制方式赋值

int b = 0123; //八进制方式赋值， 以数字0开头

int c = 0xABC; //十六进制方式赋值

//如果在printf中输出一个十进制数那么用%d，八进制用%o，十六进制是%x

*printf*("十进制：%d\n",a );

*printf*("八进制：%o\n", b); //%o,为字母o,不是数字

*printf*("十六进制：%x\n", c);

return 0;

}

2.7 计算机内存数值存储方式（了解）

2.7.1 原码

一个数的原码(原始的二进制码)有如下特点：

* 最高位做为符号位，0表示正,为1表示负
* 其它数值部分就是数值本身绝对值的二进制数
* 负数的原码是在其绝对值的基础上，最高位变为1

下面数值以1字节(8bit、一个bit位就是一个二进制位)的大小描述：

| **十进制数** | **原码** |
| --- | --- |
| +15 | 0000 1111 |
| -15 | 1000 1111 |
| +0 | 0000 0000 |
| -0 | 1000 0000 |

2.7.2 反码

* 对于正数，反码与原码相同
* 对于负数，符号位不变，其它部分取反(1变0,0变1)

| **十进制数** | **反码** |
| --- | --- |
| +15 | 0000 1111 |
| -15 | 1111 0000 |
| +0 | 0000 0000 |
| -0 | 1111 1111 |

反码运算也不方便，通常用来作为求补码的中间过渡。

2.7.3 补码

**在计算机系统中，数值一律用补码来存储。**

补码特点：

* 对于正数，原码、反码、补码相同
* 对于负数，其补码为它的反码加1
* 补码符号位不动，其他位求反，最后整个数加1，得到原码

| **十进制数** | **补码** |
| --- | --- |
| +15 | 0000 1111 |
| -15 | 1111 0001 |
| +0 | 0000 0000 |
| -0 | 0000 0000 |

2.7.4 补码的意义

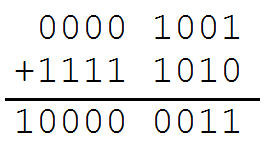
但是如果以补码方式存储，补码统一了零的编码：

| **十进制数** | **补码** |
| --- | --- |
| +0 | 0000 0000 |
| -0 | 10000 0000由于只用8位描述，最高位1丢弃，变为0000 0000 |

示例2：计算9-6的结果

以补码方式相加：

| **十进制数** | **补码** |
| --- | --- |
| 9 | 0000 1001 |
| -6 | 1111 1010 |



最高位的1溢出,剩余8位二进制表示的是3，正确。

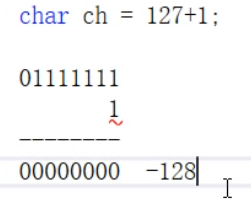
**在计算机系统中，数值一律用补码来存储**，主要原因是：

* 统一了零的编码
* 将符号位和其它位统一处理
* 将减法运算转变为加法运算
* 两个用补码表示的数相加时，如果最高位(符号位)有进位，则进位被舍弃

人为规定：00000000代表-128

2.7.5 数值溢出

当超过一个数据类型能够存放最大的范围时，数值会溢出。



| **数据类型** | **占用空间** | **取值范围** |
| --- | --- | --- |
| char | 1字节 | -128到 127(-27 ~ 27-1) |
| unsigned char | 1字节 | 0 到 255(0 ~ 28-1) |

2.8类型限定符

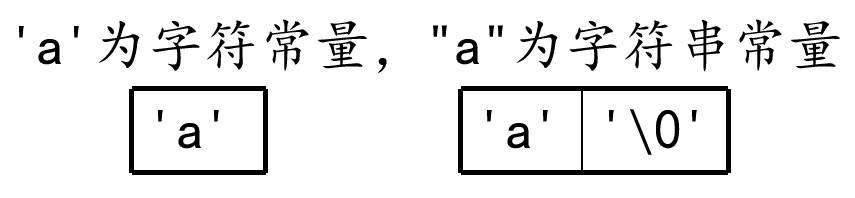
| **限定符** | **含义** |
| --- | --- |
| extern | 声明一个变量，extern声明的变量没有建立存储空间。  extern int a;//变量在定义的时候创建存储空间 |
| const | 定义一个常量，常量的值不能修改。  const int a = 10; |

2.9字符串格式化输出和输入

2.9.1 字符串常量

* 字符串是内存中一段连续的char空间，以'\0'(数字0)结尾。
* 字符串常量是由双引号括起来的字符序列，如“china”、“C program”，“$12.5”等都是合法的字符串常量。

字符串常量与字符常量的不同：



每个字符串的结尾，编译器会自动的添加一个结束标志位'\0'，即 "a" 包含两个字符'a'和’\0’。

2.9.2 printf函数和putchar函数

printf是输出一个字符串，putchar输出一个char。

printf格式字符：

| **打印格式** | **对应数据类型** | **含义** |
| --- | --- | --- |
| %d | int | 接受整数值并将它表示为有符号的十进制整数 |
| %hd | short int | 短整数 |
| %hu | unsigned short | 无符号短整数 |
| %o | unsigned int | 无符号8进制整数 |
| %u | unsigned int | 无符号10进制整数 |
| %x,%X | unsigned int | 无符号16进制整数，x对应的是abcdef，X对应的是ABCDEF |
| %f | float | 单精度浮点数 |
| %lf | double | 双精度浮点数 |
| %e,%E | double | 科学计数法表示的数，此处"e"的大小写代表在输出时用的"e"的大小写 |
| %c | char | 字符型。可以把输入的数字按照ASCII码相应转换为对应的字符 |
| %s | char \* | 字符串。输出字符串中的字符直至字符串中的空字符（字符串以'\0‘结尾，这个'\0'即空字符） |
| %p | void \* | 以16进制形式输出指针 |
| %% | % | 输出一个百分号 |

printf附加格式：

| **字符** | **含义** |
| --- | --- |
| l(字母l) | 附加在d,u,x,o前面，表示长整数 |
| - | 左对齐 |
| m(代表一个整数) | 数据最小宽度 |
| 0(数字0) | 将输出的前面补上0直到占满指定列宽为止不可以搭配使用- |
| m.n(代表一个整数) | m指域宽，即对应的输出项在输出设备上所占的字符数。n指精度，用于说明输出的实型数的小数位数。对数值型的来说，未指定n时，隐含的精度为n=6位。 |

#include <stdio.h>

int *main*()

{

char a[] = “hello”;

*printf*("a = %s\n", a);//格式化输出一个字符串

*printf*("%p\n", &a);//输出变量a在内存中的地址编号

*printf*("%10s\n", a);//10位，默认右对齐

*printf*("%-10s\n", a);//10位，左对齐

*printf("%x\n", a);* // 16进制数

*printf*("%%d\n"); // 两个%%，打印出一个

char a = 'a';

*putchar*(a);//putchar只有一个参数，不能输出字符串

printf("\n");

putchar(97); // asc码 也输出a

printf("\n");

putchar('abc'); // 错乱显示

printf("\n");

putchar("abc"); // 错乱显示

return 0;

}

2.9.3 scanf函数

* scanf通过%转义的方式可以得到用户通过标准输入设备输入的数据。

#include <stdio.h>

// 从键盘接收用户的输入：

// 1、整数 2、字符 3、字符串

//

// 1、整数

//int int\_a; // 只定义，未赋值

//scanf("%d", &int\_a); // scanf中变量名前要加&，代表此变量的地址

//printf("打印刚刚你输出的内容为：%d", int\_a);

//int int\_a, int\_b, int\_c; // 一次性定义多个变量

//printf("请输入三个数字，我来帮你打印出来\n");

//// scanf中多个整数变量不加空格，默认使用回车进行区分

//// 多个整数变量的格式化输入，中间加空格分割

//scanf("%d %d %d",&int\_a, &int\_b, &int\_c);

//printf("您刚刚输入的数字为：%d-----%d----%d", int\_a, int\_b, int\_c);

// 2、字符

//char cha\_a;

//printf("请输入一个字符\n");

//scanf("%c", &cha\_a);

//printf("您刚刚输入的内容为：%c", cha\_a);

// 字符是一个，不需要注意空格

//char cha\_a,cha\_b,cha\_c;

//printf("请输入三个字符\n");

//scanf("%c%c%c", &cha\_a,&cha\_b,&cha\_c);

//printf("您刚刚输入的内容为：%c%c%c", cha\_a, cha\_b, cha\_c);

// 3、字符串

char str\_a[5]; // 字符串只定义，不赋值的话，务必在中括号中，设置一个宽度

printf("请输入字符串\n");

// 要注意字符串最后默认有\0作为结束，你设置宽度的时候，别忘了呦

scanf("%s",str\_a); // 字符串其实就是数组，不需要使用&符号

printf("您刚刚输入的内容为：%s", str\_a);

// 自行输入内容的时候，如果输入了空格，空格后的内容，不会被存储

return 0;

}

2.9.4 getchar函数

* 获取用户输入的一个字符，返回asc码

#include <stdio.h>

int *main*()

{

char ch\_a = getchar(); // 接收用户输入的一个字符，返回asc码

printf("打印%c\n", ch\_a);

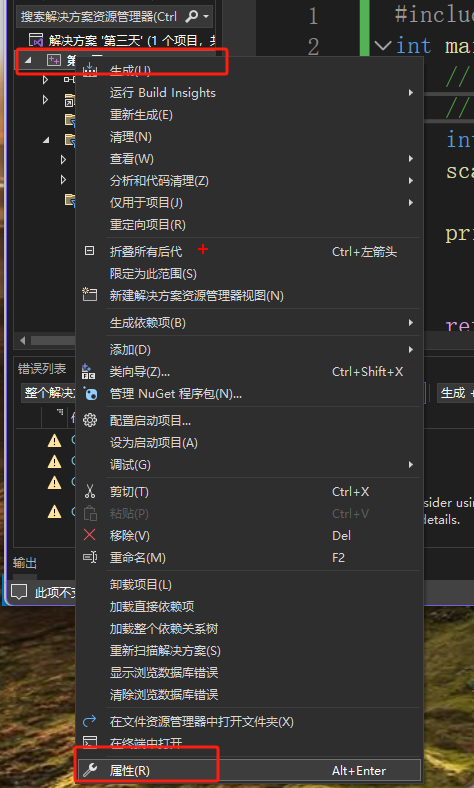
printf("打印%d\n", ch\_a);// 打印的就是对应的asc码

putchar(ch\_a);

return 0;

}

Scanf错误提示解决办法：





3. 运算符与表达式

3.1 常用运算符分类

| **运算符类型** | **作用** |
| --- | --- |
| 算术运算符 | 用于处理四则运算 |
| 赋值运算符 | 用于将表达式的值赋给变量 |
| 比较运算符 | 用于表达式的比较，并返回一个真值或假值 |
| 逻辑运算符 | 用于根据表达式的值返回真值或假值 |
| sizeof运算符 | 用于求字节数长度 |

3.2 算术运算符

| **运算符** | **术语** | **示例** | **结果** |
| --- | --- | --- | --- |
| + | 正号 | +3 | 3 |
| - | 负号 | -3 | -3 |
| + | 加 | 10 + 5 | 15 |
| - | 减 | 10 - 5 | 5 |
| \* | 乘 | 10 \* 5 | 50 |
| / | 除 | 10 / 5 | 2 |
| % | 取模(取余) | 10 % 3 | 1 |
| ++ | 前自增 | a=2; b=++a; | a=3; b=3; |
| ++ | 后自增 | a=2; b=a++; | a=3; b=2; |
| -- | 前自减 | a=2; b=--a; | a=1; b=1; |
| -- | 后自减 | a=2; b=a--; | a=1; b=2; |

3.3 赋值运算符

| **运算符** | **术语** | **示例** | **结果** |
| --- | --- | --- | --- |
| = | 赋值 | a=2; b=3; | a=2; b=3; |
| += | 加等于 | a=0; a+=2; | a=2; |
| -= | 减等于 | a=5; a-=3; | a=2; |
| \*= | 乘等于 | a=2; a\*=2; | a=4; |
| /= | 除等于 | a=4; a/=2; | a=2; |
| %= | 模等于 | a=3; a%2; | a=1; |

3.4 比较运算符

C 语言的比较运算中， “真”用数字“1”来表示， “假”用数字“0”来表示。

| **运算符** | **术语** | **示例** | **结果** |
| --- | --- | --- | --- |
| == | 相等于 | 4 == 3 | 0 |
| != | 不等于 | 4 != 3 | 1 |
| < | 小于 | 4 < 3 | 0 |
| > | 大于 | 4 > 3 | 1 |
| <= | 小于等于 | 4 <= 3 | 0 |
| >= | 大于等于 | 4 >= 1 | 1 |

3.5 逻辑运算符

| **运算符** | **术语** | **示例** | **结果** |
| --- | --- | --- | --- |
| ! | 非 | !a | 如果a为假，则!a为真；  如果a为真，则!a为假。 |
| && | 与 | a && b | 如果a和b都为真，则结果为真，否则为假。 |
| || | 或 | a || b | 如果a和b有一个为真，则结果为真，二者都为假时，结果为假。 |

3.6 运算符优先级

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **优先级** | **运算符** | **名称或含义** | **使用形式** | **结合方向** | **说明** |
| **1** | **[]** | 数组下标 | 数组名[常量表达式] | 左到右 | -- |
| **()** | 圆括号 | (表达式）/函数名(形参表) | -- |
| **.** | 成员选择（对象） | 对象.成员名 | -- |
| **->** | 成员选择（指针） | 对象指针->成员名 | -- |
|  | | | | | |
| **2** | **-** | 负号运算符 | -表达式 | **右到左** | 单目运算符 |
| **~** | 按位取反运算符 | ~表达式 |
| **++** | 自增运算符 | ++变量名/变量名++ |
| **--** | 自减运算符 | --变量名/变量名-- |
| **\*** | 取值运算符 | \*指针变量 |
| **&** | 取地址运算符 | &变量名 |
| **!** | 逻辑非运算符 | !表达式 |
| **(类型)** | 强制类型转换 | (数据类型)表达式 | -- |
| **sizeof** | 长度运算符 | sizeof(表达式) | -- |
|  | | | | | |
| **3** | **/** | 除 | 表达式/表达式 | 左到右 | 双目运算符 |
| **\*** | 乘 | 表达式\*表达式 |
| **%** | 余数（取模） | 整型表达式%整型表达式 |
| **4** | **+** | 加 | 表达式+表达式 | 左到右 | 双目运算符 |
| **-** | 减 | 表达式-表达式 |
| **5** | **<<** | 左移 | 变量<<表达式 | 左到右 | 双目运算符 |
| **>>** | 右移 | 变量>>表达式 |
| **6** | **>** | 大于 | 表达式>表达式 | 左到右 | 双目运算符 |
| **>=** | 大于等于 | 表达式>=表达式 |
| **<** | 小于 | 表达式<表达式 |
| **<=** | 小于等于 | 表达式<=表达式 |
| **7** | **==** | 等于 | 表达式==表达式 | 左到右 | 双目运算符 |
| **!=** | 不等于 | 表达式!= 表达式 |
|  | | | | | |
| **8** | **&** | 按位与 | 表达式&表达式 | 左到右 | 双目运算符 |
| **9** | **^** | 按位异或 | 表达式^表达式 | 左到右 | 双目运算符 |
| **10** | **|** | 按位或 | 表达式|表达式 | 左到右 | 双目运算符 |
| **11** | **&&** | 逻辑与 | 表达式&&表达式 | 左到右 | 双目运算符 |
| **12** | **||** | 逻辑或 | 表达式||表达式 | 左到右 | 双目运算符 |
|  | | | | | |
| **13** | **?:** | 条件运算符 | 表达式1?  表达式2: 表达式3 | **右到左** | 三目运算符 |
| **14** | **=** | 赋值运算符 | 变量=表达式 | **右到左** | -- |
| **/=** | 除后赋值 | 变量/=表达式 | -- |
| **\*=** | 乘后赋值 | 变量\*=表达式 | -- |
| **%=** | 取模后赋值 | 变量%=表达式 | -- |
| **+=** | 加后赋值 | 变量+=表达式 | -- |
| **-=** | 减后赋值 | 变量-=表达式 | -- |
| **<<=** | 左移后赋值 | 变量<<=表达式 | -- |
| **>>=** | 右移后赋值 | 变量>>=表达式 | -- |
| **&=** | 按位与后赋值 | 变量&=表达式 | -- |
| **^=** | 按位异或后赋值 | 变量^=表达式 | -- |
| **|=** | 按位或后赋值 | 变量|=表达式 | -- |
|  | | | | | |
| **15** | **，** | 逗号运算符 | 表达式,表达式,… | 左到右 | -- |

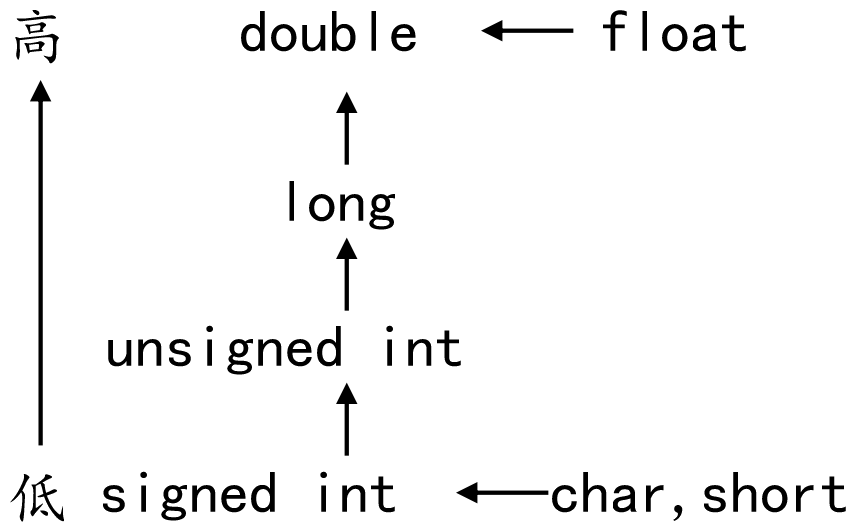
3.7 类型转换

数据有不同的类型，不同类型数据之间进行混合运算时必然涉及到类型的转换问题。

转换的方法有两种：

* 自动转换(隐式转换)：遵循一定的规则,运算的过程中由编译系统自动完成。
* 强制类型转换：把表达式的运算结果强制转换成所需的数据类型。

类型转换的原则：占用内存字节数少(值域小)的类型，向占用内存字节数多(值域大)的类型转换，以保证精度不降低。



3.7.1 隐式转换

#include <stdio.h>

int *main*()

{

int r = 3;

float s = 3.14 \* r \* r; // 3.14后没有f，代表double类型。和int类型相乘运算后，隐式转换为double，再赋值给float类型的s。

int a = 320;

char ch = a; // 赋值的时候也会隐式转换，大转小有可能发送数据丢失

printf(“%d”,ch); //

return 0;

}

3.7.2 强制转换

强制类型转换指的是使用强制类型转换运算符，将一个变量或表达式转化成所需的类型，其基本语法格式如下所示：

(类型说明符) (表达式)

#include <stdio.h>

int *main*()

{

float price = 3.2;

int weight = 4;

// 强制类型的转换，小括号中写目标类型

//double result = price \* weight;

//double result = (int)price \* weight;

double result = (int)(price \* weight);

printf("价格%lf", result);

return 0;

}

4. 程序流程结构

4.1 概述

C语言支持最基本的三种程序运行结构：顺序结构、选择结构、循环结构。

* 顺序结构：程序按顺序执行，不发生跳转。
* 选择结构：依据是否满足条件，有选择的执行相应功能。
* 循环结构：依据条件是否满足，循环多次执行某段代码。

4.2 选择结构

4.2.1 if语句



#include <stdio.h>

int *main*()

{

int age;

scanf("%d", &age);

if (age >= 18) { // 小括号写判断，满足条件才会进入到大括号中

printf("%d已经成年了，可以来上网了\n",age);

}

return 0;

}

4.2.2 if…else语句



#include <stdio.h>

int *main*()

{

int age;

scanf("%d", &age);

if (age >= 18) { // 小括号写判断，满足条件才会进入到大括号中

printf("%d已经成年了，可以来上网了\n", age);

}

else {

printf("%d没成年就回家写作业去把\n", age);

}

return 0;

}

4.2.3 if…else if…else语句



#include <stdio.h>

int *main*()

{

int score;

printf("请输入您的成绩:");

scanf("%d", &score);

if (score >= 90 && score <= 100) {

printf("很优秀\n");

}

else if (score < 90 && score >= 80) {

printf("良好\n");

}

else if (score < 80 && score >= 70) {

printf("中等\n");

}

else if (score < 70 && score >= 60) {

printf("及格\n");

}

else {

printf("再接再厉，下次争取好成绩\n");

}

return 0;

}

4.2.4 switch语句

和if明显有区别：

if模糊；switch精确



#include <stdio.h>

int *main*()

{

int c;

scanf("%d", &c);

switch (c)

{

case 1:

*printf*("OK\n");

break;//switch遇到break就中断了

case 2:

*printf*("not OK\n");

break;

default://如果上面的条件都不满足，那么执行default

*printf*("are u ok?\n");

}

return 0;

}

4.3 循环结构

4.3.1 while语句



#include <stdio.h>

int *main*()

{

int a = 0;

while (a < 10)

{

*printf*("a = %d\n", a);

a++;

}

return 0;

}

4.3.2 do…while语句



#include <stdio.h>

int *main*()

{

int a = 1;

do

{

a++;

*printf*("a = %d\n", a);

} while (a < 10);

return 0;

}

4.3.3 for语句

#include <stdio.h>

int *main*()

{

int i;

int sum = 0;

for (i = 0; i <= 100; i++) // （初始值;循环条件;自增）

{

sum += i;

}

*printf*("sum = %d\n", sum);

return 0;

}

4.3.4 嵌套循环

循环语句之间可以相互嵌套：

#include <stdio.h>

int *main*()

{

int num = 0;

int i, j, k;

for (i = 0; i < 10; i++)

{

for (j = 0; j < 100; j++)

{

for (k = 0; k < 10; k++)

{

*printf*("hello world\n");

num++;

}

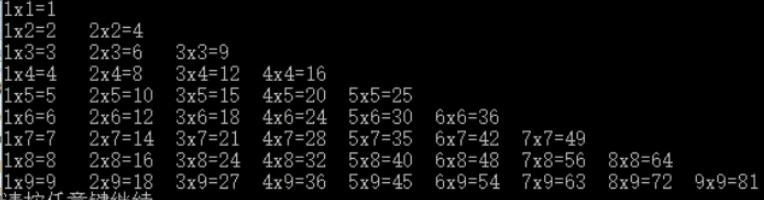
}

}

*printf*("num = %d\n", num);

return 0;

}



4.4 跳转语句break、continue、goto

4.3.1 break语句

在switch条件语句和循环语句中都可以使用break语句：

* 当它出现在switch条件语句中时，作用是终止某个case并跳出switch结构。
* 当它出现在循环语句中，作用是跳出当前内循环语句，执行后面的代码。
* 当它出现在嵌套循环语句中，跳出最近的内循环语句，执行后面的代码。

#include <stdio.h>

int *main*()

{

int flag = 0;

int m = 0;

int n = 0;

for (m = 0; m < 10; m++)

{

for (n = 0; n < 10; n++)

{

if (n == 5)

{

flag = 1;

break; //跳出for (n = 0; n < 10; n++)

}

}

if (flag == 1)

{

break; //跳出for (m = 0; m < 10; m++)

}

}

return 0;

}

4.3.2 continue语句

在循环语句中，如果希望立即终止本次循环，并执行下一次循环，此时就需要使用continue语句。

#include<stdio.h>

int *main*()

{

return 0;

}

4.3.3 goto语句(无条件跳转，不要用！)

#include <stdio.h>

int *main*()

{

hehe:

*printf*("aaaaaaaaa\n");

*printf*("bbbbbbbb\n");

goto hehe; //无条件跳转到End的标识

*printf*("ccccccccc\n");

return 0;

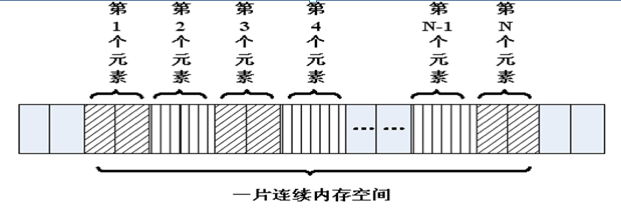
}

5. 数组和字符串

5.1 概述

在程序设计中，为了方便处理数据把具有相同类型的若干变量按有序且连续的形式组织起来——称为数组。

数组就是在内存中连续的相同类型的变量空间。同一个数组所有的成员都是相同的数据类型，同时所有的成员在内存中的地址是连续的。



数组属于构造数据类型：

* 一个数组可以分解为多个数组元素：这些数组元素可以是基本数据类型或构造类型。

int a[10];

struct Stu boy[10];

* 按数组元素类型的不同，数组可分为：数值数组、字符数组、指针数组、结构数组等类别。

int a[10];

char s[10];

char \*p[10];

通常情况下，数组元素下标的个数也称为维数，根据维数的不同，可将数组分为一维数组、二维数组、三维数组、四维数组等。通常情况下，我们将二维及以上的数组称为多维数组。

5.2 一维数组

5.2.1 一维数组的定义和使用

* 数组名字符合标识符的书写规定(数字、英文字母、下划线)
* 数组名不能与其它变量名相同，同一作用域内是唯一的
* 中括号[]表示数组元素的个数

int a[3]表示数组a有3个元素

其地址下标、索引值从0开始计算，因此3个元素分别为a[0],a[1],a[2]

* 定义数组时[]内最好是常量，使用数组时[]内即可是常量，也可以是变量

5.2.2 一维数组的初始化

在定义数组的同时进行赋值，称为初始化。全局数组若不初始化，编译器将其初始化为零。局部数组若不初始化，内容为随机值。

int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };//初始化所有成员变量，多了会报错

int a[10] = { 1, 2, 3 };//初始化前三个成员，后面所有元素都设置为0

int a[10] = { 0 };//所有的成员都设置为0

int a[10];//所有的成员都是随机数，不推荐使用！

//[]中不定义元素个数，系统自动统计个数

int a[] = { 1, 2, 3, 4, 5 };//定义了一个数组，有5个成员

int a[]; //报错

5.2.3 强化训练

1) 一维数组的最值

#include <stdio.h>

int *main*()

{

int a[] = { 1, -2, 3,- 4, 5, -6, 7, -8, -9, 10 };//定义一个数组，同时初始化所有成员变量

int i = 0;

int max = a[0];

for (i = 1; i < sizeof(a) / sizeof(a[0]); i++)

{

if (a[i] > max)

{

max = a[i];

}

}

*printf*("数组中最大值为：%d\n", max);

return 0;

}

2) 一维数组的逆置

#include <stdio.h>

int *main*()

{

int a[] = { 1, -2, 3,- 4, 5, -6, 7, -8, -9, 10 };//定义一个数组，同时初始化所有成员变量

int i = 0;

int j = sizeof(a) / sizeof(a[0]) -1;

int tmp;

while (i < j)

{

i++;

j--;

tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)

{

*printf*("%d ", a[i]);

}

*printf*("\n");

return 0;

}

3) 冒泡法排序

#include <stdio.h>

int *main*()

{

int a[] = { 1, -2, 3,- 4, 5, -6, 7, -8, -9, 10 };//定义一个数组，同时初始化所有成员变量

int i = 0;

int j = 0;

int n = sizeof(a) / sizeof(a[0]);

int tmp;

//1、流程

//2、试数

for (i = 0; i < n-1; i++)

{

for (j = 0; j < n - i -1 ; j++)//内循环的目的是比较相邻的元素，把大的放到后面

{

if (a[j] > a[j + 1])

{

tmp = a[j];

a[j] = a[j+1];

a[j+1] = tmp;

}

}

}

for (i = 0; i < n; i++)

{

*printf*("%d ", a[i]);

}

*printf*("\n");

return 0;

}

5.3二维数组

5.3.1 二维数组的定义和使用

二维数组定义的一般形式是：

类型说明符 数组名[常量表达式1][常量表达式2]

其中常量表达式1表示第一维下标的长度，常量表达式2 表示第二维下标的长度。

int a[3][4];

* 命名规则同一维数组
* 定义了一个三行四列的数组，数组名为a其元素类型为整型，该数组的元素个数为3×4个，即：



二维数组a是按行进行存放的，先存放a[0]行，再存放a[1]行、a[2]行，并且每行有四个元素，也是依次存放的。

* 二维数组在概念上是二维的：其下标在两个方向上变化，对其访问一般需要两个下标。
* 在内存中并不存在二维数组，二维数组实际的硬件存储器是连续编址的，也就是说内存中只有一维数组，即放完一行之后顺次放入第二行，和一维数组存放方式是一样的。

#include <stdio.h>

int *main*()

{

int a[3][4];

//遍历数组，并输出每个成员的值

for (i = 0; i < 3; i++)

{

for (j = 0; j < 4; j++)

{

*printf*("%d, ", a[i][j]);

}

*printf*("\n");

}

return 0;

}

5.3.2 二维数组的初始化

//分段赋值 int a[3][4] = {{ 1, 2, 3, 4 },{ 5, 6, 7, 8, },{ 9, 10, 11, 12 }};

int a[3][4] =

{

{ 1, 2, 3, 4 },

{ 5, 6, 7, 8, },

{ 9, 10, 11, 12 }

};

//连续赋值

int a[3][4] = { 1, 2, 3, 4 , 5, 6, 7, 8, 9, 10, 11, 12 };

//可以只给部分元素赋初值，未初始化则为0

int a[3][4] = { 1, 2, 3, 4 };

//所有的成员都设置为0

int a[3][4] = {0};

//[]中不定义行数，列数自动计算

int a[][4] = { 1, 2, 3, 4, 5, 6, 7, 8};

5.3.3 数组大小计算

数组名是一个地址的常量，代表数组中首元素的地址。

#include <stdio.h>

int *main*()

{

// 数据大小

int arr[3][5] = {

{2,3,4,6,7},

{5,6,7,6,7},

{8,9,0,6,7}

};

// 二维数组的大小

printf("二维数组大小为：%u\n" ,sizeof(arr));

// 二维数组多少行

printf("多少行：%u\n", sizeof(arr) / sizeof(arr[0]));

// 二维数组每行多大

printf("每行多大：%u\n",sizeof(arr[0]));

//二维数组每个元素多大

printf("每个元素多大：%u\n",sizeof(arr[0][0]));

// 二维数组列数多大

printf("列数：%u\n",sizeof(arr[0]) / sizeof(arr[0][0]));

return 0;

}

5.3.4 强化训练

#include <stdio.h>

int *main*()

{

//二维数组： 五行、三列

// 用户输入数据，生成一个5行3列的数组

int scores[5][3];

int row = sizeof(scores) / sizeof(scores[0]); // 行数

int col = sizeof(scores[0]) / sizeof(scores[0][0]); // 列

int i, j;

// 循环输入数据

for (i = 0; i < row; i++) {

for (j = 0; j < col; j++) {

scanf("%d",&scores[i][j]);

}

}

// 打印验证用户输入

//for (i = 0; i < row; i++) {

// for (j = 0; j < col; j++) {

// printf("%d ", scores[i][j]);

// }

// printf("\n"); //输出一个换行

//}

// 求每个学生的总成绩

//for (i = 0; i < row; i++) { // 每个学生

// int sum = 0; // 每个学生的总成绩,每次循环进来都设置初始值为0

// for (j = 0; j < col; j++) { // 每个学生的成绩

// sum += scores[i][j]; // sum = sum + socres[i][j]

// }

// printf("第%d个学生的总成绩为：%d\n",i+1,sum);

//

//}

// 求每门课程的总成绩

for (j = 0; j < col; j++) { // 第几门课程

int sum = 0; // 初始值要设置为0，

for (i = 0; i < row; i++) { // 每门课程对应的学生

sum += scores[i][j];

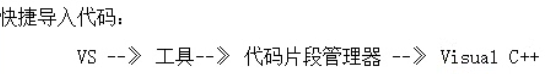
}

printf("第%d门课程的总成绩为：%d\n",j+1,sum);

}

return 0;

}



5.4多维数组(了解)

多维数组的定义与二维数组类似，其语法格式具体如下：

数组类型修饰符 数组名 [n1][n2]…[nn];

int a[3][2][4];

定义了一个三维数组，数组的名字是a，数组的长度为3，每个数组的元素又是一个二维数组，这个二维数组的长度是2，并且这个二维数组中的每个元素又是一个一维数组，这个一维数组的长度是4，元素类型是int。

{

{

{1,2,3,4},

{1,2,3,4}

},

{

{1,2,3,4},

{1,2,3,4}

},

{

{1,2,3,4},

{1,2,3,4}

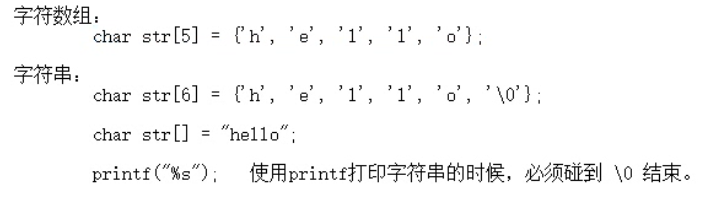
}

}

5.5 字符数组与字符串

5.5.1 字符数组与字符串区别

* C语言中没有字符串这种数据类型，可以通过char的数组来替代；
* 字符串一定是一个char的数组，但char的数组未必是字符串；
* 字符‘\0’结尾的char数组就是一个字符串，但如果char数组没有以\0结尾，就不是一个字符串，只是普通字符数组。所以字符串是一种特殊的char的数组。



5.5.2 字符串的初始化

#include <stdio.h>

// C语言没有字符串类型，通过字符数组模拟

// C语言字符串，以字符‘\0’, 数字0

int *main*()

{

//char str[5] = {'h','e','l','l','o'}; // 没有设置\0结束，打印的时候会多出一些乱码

//char str[6] = {'h','e','l','l','o','\0'}; // 有\0结束就不会多出乱码

//char str[] = "hello"; // 自动添加\0

//printf("%s",str);

return 0;

}

5.5.3 字符串的输入输出

1) gets()

#include <stdio.h>

**gets(str);**

功能：从标准输入读入字符，并保存到s指定的内存空间，直到出现换行符或读到文件结尾为止。

参数：

str：字符串首地址

返回值：

成功：读入的字符串

失败：NULL

gets(str)与scanf(“%s”,str)的区别：

* gets(str)允许输入的字符串含有空格
* scanf(“%s”,str)不允许含有空格

注意：由于scanf()和gets()无法知道字符串s大小，必须遇到换行符或读到文件结尾为止才接收输入，因此容易导致字符数组越界(缓冲区溢出)的情况。

char str[100];

*printf*("请输入str: ");

*gets*(str);

*printf*("str = %s\n", str);

2) fgets()

#include <stdio.h>

**fgets(str, int size, *FILE* \*stream);**

功能：从stream指定的文件内读入字符，保存到s所指定的内存空间，直到出现换行字符、读到文件结尾或是已读了size - 1个字符为止，最后会自动加上字符 '\0' 作为字符串结束。

参数：

str：字符串

size：指定最大读取字符串的长度size - 1

stream：文件指针，如果读键盘输入的字符串，固定写为stdin

返回值：

成功：成功读取的字符串

读到文件尾或出错： *NULL*

fgets()在读取一个用户通过键盘输入的字符串的时候，同时把用户输入的回车也做为字符串的一部分。通过scanf和gets输入一个字符串的时候，不包含结尾的“\n”，但通过fgets结尾多了“\n”。fgets()函数是安全的，不存在缓冲区溢出的问题。

char str[5];

*printf*("请输入str: ");

*fgets*(str, sizeof(str), *stdin*);

*printf*("str = %s", str);

3) puts()

#include <stdio.h>

**int puts(const char \*s);**

功能：标准设备输出s字符串，在输出完成后自动输出一个'\n'。

参数：

s：字符串首地址

返回值：

成功：非负数 0

失败：-1

#include <stdio.h>

int *main*()

{

*printf*("hello world");

*puts*("hello world");

return 0;

}

4) fputs()

#include <stdio.h>

**int fputs(const char\* str, FILE \* stream);**

功能：将str所指定的字符串写入到stream指定的文件中， 字符串结束符 '\0' 不写入文件。

参数：

str：字符串

stream：文件指针，如果把字符串输出到屏幕，固定写为stdout

返回值：

成功：0

失败：-1

fputs()是puts()的文件操作版本，但fputs()不会自动输出一个'\n'。

*printf*("hello world");

*puts*("hello world");

*fputs*("hello world", *stdout*);

5) strlen()

#include <string.h>

***size\_t* strlen(const char \*s);**

功能：计算指定指定字符串s的长度，不包含字符串结束符‘\0’

参数：

s：字符串首地址

返回值：字符串s的长度，size\_t为unsigned int类型

char str[] = "abc\0defg";

int n = *strlen*(str);

*printf*("n = %d\n", n);

5.5.4 强化训练：字符串追加

#include <stdio.h>

int *main*()

{

char str1[] = "abcdef";

char str2[] = "123456";

char dst[100];

int i = 0;

while (str1[i] != 0)

{

dst[i] = str1[i];

i++;

}

int j = 0;

while (str2[j] != 0)

{

dst[i + j] = str2[j];

j++;

}

dst[i + j] = ‘\0’; //字符串结束符

*printf*("dst = %s\n", dst);

return 0;

}

6. 函数

6.1 概述

6.1.1 函数分类

C 程序是由函数组成的，我们写的代码都是由主函数 main()开始执行的。函数是 C 程序的基本模块，是用于完成特定任务的程序代码单元。

从函数定义的角度看，函数可分为系统函数和用户定义函数两种：

* 系统函数，即库函数：这是由编译系统提供的，用户不必自己定义这些函数，可以直接使用它们，如我们常用的打印函数printf()。
* 用户定义函数：用以解决用户的专门需要。

6.1.2 函数的作用

* 函数的使用可以省去重复代码的编写，降低代码重复率

// 求两数的最大值

int *max*(int a, int b)

{

if (a > b){

return a;

}

else{

return b;

}

}

int *main*()

{

// 操作1 ……

// ……

int a1 = 10, b1 = 20;

c1 = *max*(a1, b1); // 调用max()

// 操作2 ……

// ……

int a2 = 11, b2 = 21;

c2 = *max*(a2, b2); // 调用max()

// ……

return 0;

}

* 函数可以让程序更加模块化，从而有利于程序的阅读，修改和完善

假如我们编写一个实现以下功能的程序：读入一行数字；对数字进行排序；找到它们的平均值；打印出一个柱状图。如果我们把这些操作直接写在main()里，这样可能会给用户感觉代码会有点凌乱。但，假如我们使用函数，这样可以让程序更加清晰、模块化：

#include <stdio.h>

int *main*()

{

float list[50];

// 这里只是举例，函数还没有实现

readlist(list, 50);

*sort*(list, 50);

average(list, 50);

bargraph(list, 50);

return 0;

}

这里我们可以这么理解，程序就像公司，公司是由部门组成的，这个部门就类似于C程序的函数。默认情况下，公司就是一个大部门( 只有一个部门的情况下 )，相当于C程序的main()函数。如果公司比较小( 程序比较小 )，因为任务少而简单，一个部门即可( main()函数 )胜任。但是，如果这个公司很大( 大型应用程序 )，任务多而杂，如果只是一个部门管理( 相当于没有部门，没有分工 )，我们可想而知，公司管理、运营起来会有多混乱，不是说这样不可以运营，只是这样不完美而已，如果根据公司要求分成一个个部门( 根据功能封装一个一个函数 )，招聘由行政部门负责，研发由技术部门负责等，这样就可以分工明确，结构清晰，方便管理，各部门之间还可以相互协调。

6.1.3 系统函数的调用：

当调用系统函数时，需要关心5要素：

* 头文件：包含指定的头文件
* 函数名字：函数名字必须和头文件声明的名字一样
* 功能：需要知道此函数能干嘛后才调用
* 参数：参数类型要匹配
* 返回值：根据需要接收返回值

#include <time.h>

***time\_t* time(*time\_t* \*t);**

功能：获取当前系统时间

参数：常设置为NULL

返回值：当前系统时间, *time\_t* 相当于long类型，单位为毫秒

#include <stdlib.h>

**void srand(unsigned int seed);**

功能：用来设置rand()产生随机数时的随机种子

参数：如果每次seed相等，rand()产生随机数相等

返回值：无

#include <stdlib.h>

**int rand(void);**

功能：返回一个随机数值

参数：无

返回值：随机数

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

int *main*()

{

*time\_t* tm = *time*(*NULL*);//得到系统时间

*srand*((unsigned int)tm);//随机种子只需要设置一次即可

int r = *rand*();

*printf*("r = %d\n", r);

return 0;

}

6.2 函数的定义

6.2.1 函数定义格式

函数定义的一般形式：

返回类型 函数名(形式参数列表)

{

数据定义部分;

执行语句部分;

}



6.2.2 函数名字、形参、函数体、返回值

1) 函数名

理论上是可以随意起名字，最好起的名字见名知意，应该让用户看到这个函数名字就知道这个函数的功能。注意，函数名的后面有个圆换号()，代表这个为函数，不是普通的变量名。

2) 形参列表

在定义函数时指定的形参，在未出现函数调用时，它们并不占内存中的存储单元，因此称它们是形式参数或虚拟参数，简称形参，表示它们并不是实际存在的数据，所以，形参里的变量不能赋值。

void *max*(int a = 10, int b = 20) // error, 形参不能赋值

{

}

在定义函数时指定的形参，必须是，类型+变量的形式：

//1: right, 类型+变量

void *max*(int a, int b)

{

}

//2: error, 只有类型，没有变量

void *max*(int, int)

{

}

//3: error, 只有变量，没有类型

int a, int b;

void *max*(a, b)

{

}

在定义函数时指定的形参，可有可无，根据函数的需要来设计，如果没有形参，圆括号内容为空，或写一个void关键字：

// 没形参， 圆括号内容为空

void *max*()

{

}

// 没形参， 圆括号内容为void关键字

void *max*(void)

{

}

3) 函数体

花括号{ }里的内容即为函数体的内容，这里为函数功能实现的过程，这和以前的写代码没太大区别，以前我们把代码写在main()函数里，现在只是把这些写到别的函数里。

4) 返回值

函数的返回值是通过函数中的return语句获得的，return后面的值也可以是一个表达式。

1. 尽量保证return语句中表达式的值和函数返回类型是同一类型。

int *max*() // 函数的返回值为int类型

{

int a = 10;

return a;// 返回值a为int类型，函数返回类型也是int，匹配

}

b)如果函数返回的类型和return语句中表达式的值不一致，则以函数返回类型为准，即函数返回类型决定返回值的类型。对数值型数据，可以自动进行类型转换。

double *max*() // 函数的返回值为double类型

{

int a = 10;

return a;// 返回值a为int类型，它会转为double类型再返回

}

注意：如果函数返回的类型和return语句中表达式的值不一致，而它又无法自动进行类型转换，程序则会报错。

c)return语句的另一个作用为中断return所在的执行函数，类似于break中断循环、switch语句一样。

int *max*()

{

return 1;// 执行到，函数已经被中断，所以下面的return 2无法被执行到

return 2;// 没有执行

}

d)如果函数带返回值，return后面必须跟着一个值，如果函数没有返回值，函数名字的前面必须写一个void关键字，这时候，我们写代码时也可以通过return中断函数(也可以不用)，只是这时，return后面不带内容( 分号“；”除外)。

void *max*()// 最好要有void关键字

{

return; // 中断函数，这个可有可无

}

6.3 函数的调用

定义函数后，我们需要调用此函数才能执行到这个函数里的代码段。这和main()函数不一样，main()为编译器设定好自动调用的主函数，无需人为调用，我们都是在main()函数里调用别的函数，一个 C 程序里有且只有一个main()函数。

调用语法： 函数名+小括号

6.3.1函数执行流程

#include <stdio.h>

void print\_test()

{

*printf*("test\n");

}

int main()

{

print\_test(); // print\_test函数的调用

printf(“我什么时候执行啊\n”);

return 0;

}

1. 进入main()函数
2. 调用print\_test()函数：
3. 它会在main()函数的前寻找有没有一个名字叫“print\_test”的函数定义；
4. 如果找到，接着检查函数的参数，这里调用函数时没有传参，函数定义也没有形参，参数类型匹配；
5. 开始执行print\_test()函数，这时候，main()函数里面的执行会阻塞( 停 )在print\_test()这一行代码，等待print\_test()函数的执行。
6. print\_test()函数执行完( 这里打印一句话 )，main()才会继续往下执行，执行到return 0, 程序执行完毕。

6.3.2 函数的形参和实参

* 形参出现在函数定义中，在整个函数体内都可以使用，离开该函数则不能使用。
* 实参出现在主调函数中，进入被调函数后，实参也不能使用。
* 实参变量对形参变量的数据传递是“值传递”，即单向传递，只由实参传给形参，而不能由形参传回来给实参。
* 在调用函数时，编译系统临时给形参分配存储单元。调用结束后，形参单元被释放。
* 实参单元与形参单元是不同的单元。调用结束后，形参单元被释放，函数调用结束返回主调函数后则不能再使用该形参变量。实参单元仍保留并维持原值。因此，在执行一个被调用函数时，形参的值如果发生改变，并不会改变主调函数中实参的值。

6.3.3 无参函数调用

如果是调用无参函数，则不能加上“实参”，但括号不能省略。

// 函数的定义

void test()

{

}

int main()

{

// 函数的调用

test(); // right, 圆括号()不能省略

test(250); // error, 函数定义时没有参数

return 0;

}

6.3.4有参函数调用

a)如果实参表列包含多个实参，则各参数间用逗号隔开。

// 函数的定义

void test(int a, int b)

{

}

int main()

{

int p = 10, q = 20;

test(p, q); // 函数的调用

return 0;

}

b)实参与形参的个数应相等，类型应匹配(相同或赋值兼容)。实参与形参按顺序对应，一对一地传递数据。

c)实参可以是常量、变量或表达式，无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值，以便把这些值传送给形参。所以，这里的变量是在圆括号( )外面定义好、赋好值的变量。

// 函数的定义

void test(int a, int b)

{

}

int main()

{

// 函数的调用

int p = 10, q = 20;

test(p, q); // right

test(11, 30 - 10); // right

test(int a, int b); // error, 不应该在圆括号里定义变量

return 0;

}

6.3.5 函数返回值

a）如果函数定义没有返回值，函数调用时不能写void关键字，调用函数时也不能接收函数的返回值。

// 函数的定义

void test()

{

}

int main()

{

// 函数的调用

test(); // right

void test(); // error, void关键字只能出现在定义，不可能出现在调用的地方

int a = test(); // error, 函数定义根本就没有返回值

return 0;

}

b）如果函数定义有返回值，这个返回值我们根据用户需要可用可不用，但是，假如我们需要使用这个函数返回值，我们需要定义一个匹配类型的变量来接收。

// 函数的定义, 返回值为int类型

int test()

{

}

int main()

{

// 函数的调用

int a = test(); // right, a为int类型

int b;

b = test(); // right, 和上面等级

char \*p = test(); // 虽然调用成功没有意义, p为char \*, 函数返回值为int, 类型不匹配

// error, 必须定义一个匹配类型的变量来接收返回值

// int只是类型，没有定义变量

int = test();

// error, 必须定义一个匹配类型的变量来接收返回值

// int只是类型，没有定义变量

int test();

return 0;

}

6.4 函数的声明

如果使用用户自己定义的函数，而该函数与调用它的函数（即主调函数）不在同一文件中，或者函数定义的位置在主调函数之后，则必须在调用此函数之前对被调用的函数作声明。

所谓函数声明，就是在函数尚在未定义的情况下，事先将该函数的有关信息通知编译系统，相当于告诉编译器，函数在后面定义，以便使编译能正常进行。

注意：一个函数只能被定义一次，但可以声明多次。

#include <stdio.h>

int max(int x, int y); // 函数的声明，分号不能省略

// int max(int, int); // 另一种方式

int main()

{

int a = 10, b = 25, num\_max = 0;

num\_max = max(a, b); // 函数的调用

*printf*("num\_max = %d\n", num\_max);

return 0;

}

// 函数的定义

int max(int x, int y)

{

return x > y ? x : y;

}

**函数定义和声明的区别：**

1）定义是指对函数功能的确立，包括指定函数名、函数类型、形参及其类型、函数体等，它是一个完整的、独立的函数单位。

2）声明的作用则是把函数的名字、函数类型以及形参的个数、类型和顺序(注意，不包括函数体)通知编译系统，以便在对包含函数调用的语句进行编译时，据此对其进行对照检查（例如函数名是否正确，实参与形参的类型和个数是否一致）。

6.5 exit函数

在main函数中调用exit和return结果是一样的，但在子函数中调用return只是代表子函数终止了，在子函数中调用exit，那么程序终止。

#include <stdio.h>

#include <stdlib.h>

int get\_int(int a,int b) {

printf("值1：%d\n",a);

printf("值2：%d\n",b);

// 非main函数中，return就代表退出当前这个函数

// exit 代表退出整个程序

//return a;

exit(a);

}

int main(void)

{

// 调用的时候，小括号中叫实参

int get\_num = get\_int(20,10);

printf("返回值是：%d", get\_num);

// main主函数中 exit和return没区别

return EXIT\_SUCCESS;

//exit(EXIT\_SUCCESS);

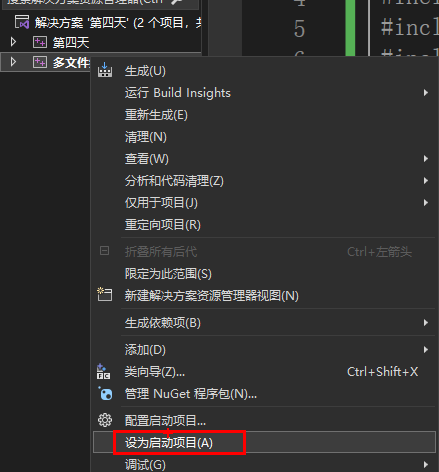
}

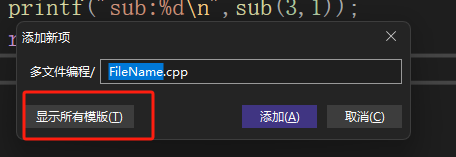
6.6 多文件(分文件)编程

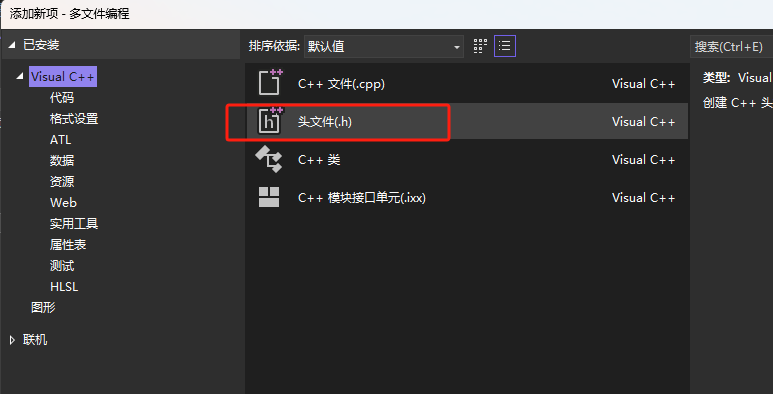
6.6.1 分文件编程

将多个不同功能的函数文件模块，编译到一起，生成一个exe文件。

* 把函数声明放在头文件xxx.h中，在主函数中包含相应头文件
* 在头文件对应的xxx.c中实现xxx.h声明的函数







#include "head.h"

6.6.2 防止头文件重复包含

当一个项目比较大时，往往都是分文件，这时候有可能不小心把同一个头文件 include 多次，或者头文件嵌套包含。

为了避免同一个文件被include多次，C/C++中有两种方式，一种是 #ifndef 方式，一种是 #pragma once 方式。

方法一：适用于Windows和其它

#ifndef \_\_HEAD\_H\_\_

#define \_\_HEAD\_H\_\_

// 声明语句

#endif

方法二：适用于Windows

#pragma once

// 声明语句

7. 指针

7.1 概述

7.1.1 内存

内存含义：

* 存储器：计算机的组成中，用来存储程序和数据，辅助CPU进行运算处理的重要部分。
* 内存：内部存贮器，暂存程序/数据——掉电丢失 SRAM、DRAM、DDR、DDR2、DDR3。
* 外存：外部存储器，长时间保存程序/数据—掉电不丢ROM、ERRROM、FLASH（NAND、NOR）、硬盘、光盘。

内存是沟通CPU与硬盘的桥梁：

* 暂存放CPU中的运算数据
* 暂存与硬盘等外部存储器交换的数据

7.1.2 物理存储器和存储地址空间

有关内存的两个概念：物理存储器和存储地址空间。

物理存储器：实际存在的具体存储器芯片。

* 主板上装插的内存条
* 显示卡上的显示RAM芯片
* 各种适配卡上的RAM芯片和ROM芯片

存储地址空间：对存储器编码的范围。我们在软件上常说的内存是指这一层含义。

* 编码：对每个物理存储单元（一个字节）分配一个号码
* 寻址：可以根据分配的号码找到相应的存储单元，完成数据的读写

7.1.3 内存地址

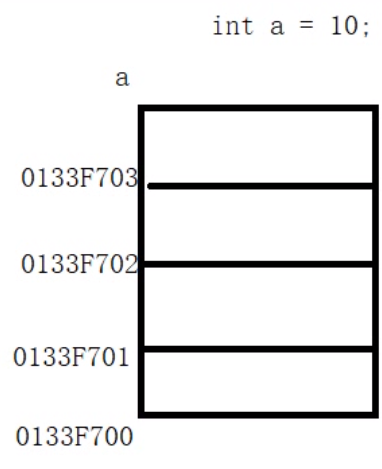
* 将内存抽象成一个很大的一维字符数组。
* 编码就是对内存的每一个字节分配一个32位或64位的编号（与32位或者64位处理器相关）。
* 这个内存编号我们称之为内存地址。

内存中的每一个数据都会分配相应的地址：

* char:占一个字节分配一个地址
* int: 占四个字节分配四个地址
* float、struct、函数、数组等

7.1.4 指针和指针变量

* 内存区的每一个字节都有一个编号，这就是“地址”。
* 如果在程序中定义了一个变量，在对程序进行编译或运行时，系统就会给这个变量分配内存单元，并确定它的内存地址(编号)
* 指针的实质就是内存“地址”。指针就是地址，地址就是指针。
* 指针是内存单元的编号，指针变量是存放地址的变量。



7.2 指针基础知识

7.2.1 指针变量的定义和使用

* 指针也是一种数据类型，指针变量也是一种变量
* 指针变量指向谁，就把谁的地址赋值给指针变量
* “\*”操作符操作的是指针变量指向的内存空间

#include <stdio.h>

int *main*()

{

int a = 0;

*printf*("%p\n", &a); //打印a的地址

//int \*代表是一种数据类型，int\*指针类型，p才是变量名

//定义了一个指针类型的变量，可以指向一个int类型变量的地址

int\* p; // int\* p---windows int \*p --linux int \* p -- 也可以

p = &a;//将a的地址赋值给变量p，p也是一个变量，值是一个内存地址编号

*printf*("%d\n", \*p);//\*p解引用、间接引用：p指向了a的地址（\*p就是a的值）

return 0;

}



7.2.2 通过指针间接修改变量的值

int a = 10;

int\* p = &a;

//\*p = 100;

//printf("a = %d, \*p = %d\n", a, \*p);

a = 20;

printf("a = %d, \*p = %d\n", a, \*p);

7.2.3 指针大小

* 使用sizeof()测量指针的大小，得到的总是：4或8
* sizeof()测的是指针变量指向存储地址的大小
* 在32位平台，所有的指针（地址）都是32位(4字节)
* 在64位平台，所有的指针（地址）都是64位(8字节)

*printf*("sizeof(int\*) = %d\n", sizeof(int\*));

*printf*("sizeof(short\*) = %d\n", sizeof(short\*));

7.2.4 野指针和空指针

指针变量也是变量，是变量就可以任意赋值，不要越界即可（32位为4字节，64位为8字节），但是，任意数值赋值给指针变量没有意义，因为这样的指针就成了野指针， (操作系统不允许操作此指针指向的内存区域)。

不要使用野指针！！！

//情况1：

int\* p;

\*p = 1000;

//情况2：

int\* p = 10; // 0-255系统占用了，

\*p = 100;

但是，野指针和有效指针变量保存的都是数值，为了标志此指针变量没有指向任何变量(空闲可用)，C语言中，可以把NULL赋值给此指针，这样就标志此指针为空指针，没有任何指针。

int \*p = *NULL*;

NULL是一个值为0的宏常量：

#define *NULL* ((void\*)0)

7.2.5万能指针、泛型指针void \*

void\*指针可以指向任意变量的内存空间：

int a = 10;

void \*p = *NULL*;

p =&a; // void万能指针可以保存任意一种变量的地址

// \*((int\*)p);

//在使用指针变量指向的内存时，转换为对应类型

printf("a = %d\n", \*((int\*)p));

7.2.6 const修饰的指针变量

// const修饰指针变量

// 结论：const右侧有\*，可以修改p，不能修改\*p；否则相反

//const int\* p = &a; 可以修改p、不可以修改\*p

//int a = 10;

//const int\* p = &a;

//\*p = 20;

//p = &a;

//int const \* p = &a; 可以修改p、不可以修改\*p

//int a = 10;

//int const \* p = &a;

//\*p = 20;

//p = &a;

//int \* const p = &a; 可以修改\*p、不可以修改p

//int a = 10;

//int \* const p = &a;

//\*p = 20;

//p = &a;

//const int \* const p = &a; 都不能改

//int a = 10;

//const int\* const p = &a;

//\*p = 20;

//p = &a;

在编辑程序时，指针作为函数参数，如果不想修改指针对应内存空间的值，需要使用const修饰指针数据类型。

7.3 指针和数组

7.3.1 数组名

数组名字是数组的首元素地址，是地址常量，不能修改！

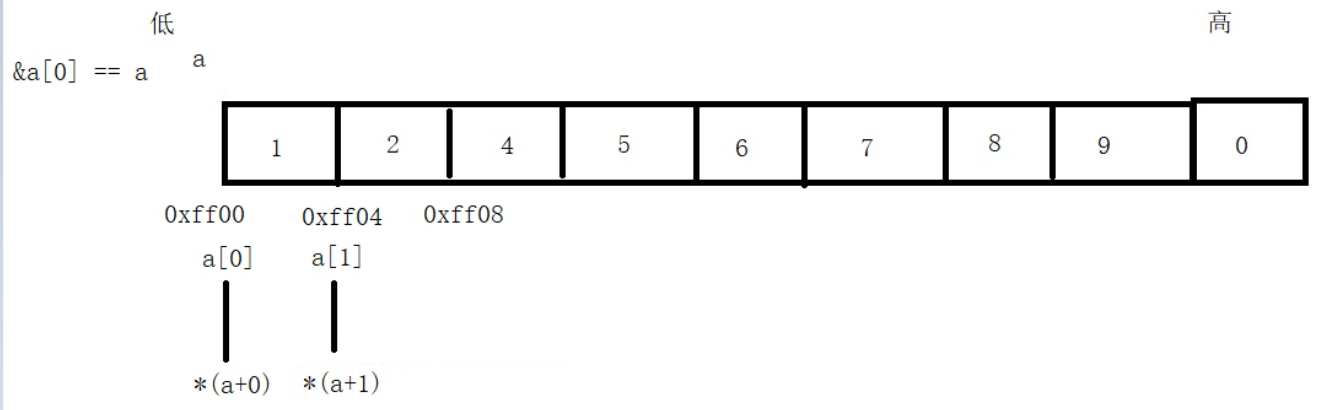
int a[] = { 1, 2, 3 };

int b[3];

// b = a; //数组名是常量，不能修改

int\* p = &b; // 指针变量可以被修改

7.3.2 指针操作数组元素



#include <stdio.h>

int *main*()

{

//取数组元素

int arr[] = {1,2,3,4,5};

int\* p = arr; // 数组地址arr前面不写&

//printf("arr[1]：%d\n",arr[1]);

//printf("\*(arr+1)：%d\n", \*(arr + 1));

//printf("p[1]：%d\n", p[1]);

//printf("\*(p+1)：%d\n", \*(p+1));

int i;

int len = sizeof(arr) / sizeof(arr[0]);

for (i = 0; i < len; i++)

{

//printf("%d " ,arr[i]);

//printf("%d " ,\*(arr+i));

//printf("%d " ,p[i]);

printf("%d ", \*(p+i));

}

return 0;

}

7.3.3 指针加减运算

1)加法运算：向右移

* 指针计算不是简单的整数相加
* 如果是一个int \*，+1的结果是移动一个int的大小
* 如果是一个char \*，+1的结果是移动一个char大小

#include <stdio.h>

int *main*()

{

int a;

int \*p = &a;

*printf*("%d\n", p);

p += 2;//移动了2个int

*printf*("%d\n", p);

char b = 0;

char \*p1 = &b;

*printf*("%d\n", p1);

p1 += 2;//移动了2个char

*printf*("%d\n", p1);

return 0;

}

通过改变指针指向操作数组元素：

#include <stdio.h>

int *main*()

{

int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int i;

int n = sizeof(a) / sizeof(a[0]);

int\* p = a;

//printf("\*p %d",\*p); // \*p 拿数组的第一个值

for (i = 0; i < n; i++)

{

// \*p==\*(p+0)==a[0]

*printf*("%d, ", \*p);

p++;

}

*printf*("\n");

return 0;

}

1. 减法运算

减法运算：向左移

#include <stdio.h>

int *main*()

{

int arr[] = {1,2,3,4,5};

int\* p = &arr[2];

printf("p%p\n",p-2);

printf("arr[0]%p\n",&arr[0]);

return 0;

}

7.3.4 指针数组

指针数组，它是数组，数组的每个元素都是指针类型。本质就是二级指针。

二维数组也是二级指针

#include <stdio.h>

int *main*()

{

// 指针数组写法1

//int a = 10;

//int b = 20;

//int c = 30;

//int\* p1 = &a;

//int\* p2 = &b;

//int\* p3 = &c;

//int\* arr[] = { p1,p2,p3 }; // 整型指针数组arr，存储的是整型地址

//printf("\*(arr[0]):%d\n", \*(arr[0]));

//// \*(arr[0]) ===== \*(\*(arr+0)) ==== \*(\*(arr)) ==== \*\*arr

//printf("\*(arr[0]):%d\n", \*\*arr); // 指针数组本质就是二级指针

// 指针数组写法2

int a[] = {10};

int b[] = {20};

int c[] = {30};

int\* arr[] = { a,b,c }; // 整型指针数组arr，存储的整型地址

printf("值：%d\n",arr[0][0]);

// 二维数组也是二级指针

// arr[0][0] == \*(arr+0)[0] === \*(\*(arr+0) + 0) ===== \*(\*arr) == \*\*arr

printf("值：%d\n", \*\*arr);

return 0;

}

7.4 多级指针

* C语言允许有多级指针存在，在实际的程序中一级指针最常用，其次是二级指针。
* 二级指针就是指向一个一级指针变量地址的指针。
* 三级指针基本用不到

int a = 10;

int\* p = &a; //一级指针

int\*\* pp = &p; // 二级指针

int\*\*\* ppp = &pp; //三级指针

printf("a：%d\n", a);

printf("\*p：%d\n", \*p);

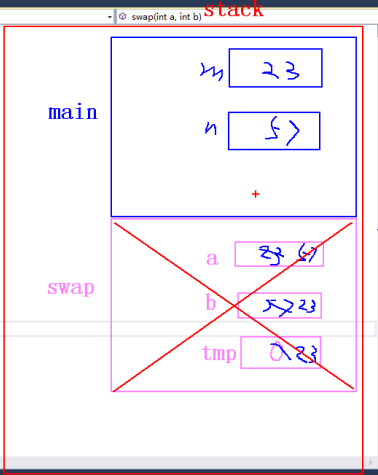
printf("\*\*pp：%d\n", \*\*pp);

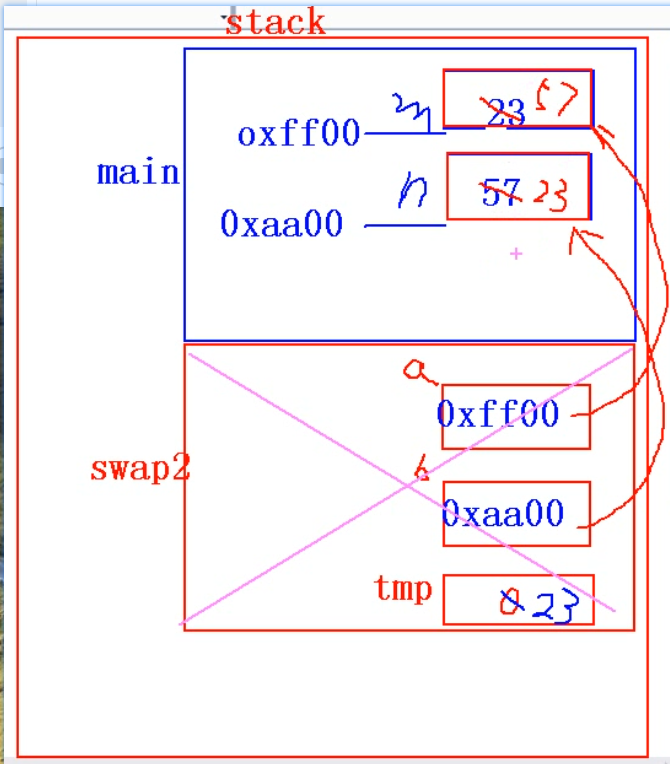
printf("\*\*\*ppp：%d\n",\*\*\*ppp);

7.5 指针和函数

7.5.1 值传递和地址传递

* 栈帧：当函数调用时，系统会在 stack 空间上申请一块内存区域，用来供函数调用，主要存放形参和局部变量(定义在函数内部)。当函数调用结束，这块内存区域自动被释放(消失)。
* 值传递：将实参的值赋给形参。传递的过程相当于在函数内部重新定义与实参同类型的变量，再把实参的值赋给该变量。就相当于拷贝了一份，本身并没有发生改变。
* 地址传递：将实参的地址传递给形参。传递的过程相当于在函数内部定义几个指针变量，然后把实参的地址赋给这些指针，指针指向的内容就是实参





#include <stdio.h>

int main(void)

{

int m = 23;

int n = 57;

printf("打印执行swap函数之前的值：m=%d,n=%d\n",m,n);

//swap(m,n); //调用函数，传值 m和n 实参

swap2(&m,&n); // 函数调用，传地址

printf("打印执行swap函数之后的值：m=%d,n=%d\n", m, n);

return EXIT\_SUCCESS;

}

int swap(int a,int b) { // a和b是形参

int temp = 0;

temp = a;

a = b;

b = temp;

return 0;

}

int swap2(int\* a, int\* b) {

int temp = 0;

temp = \*a; // 解引用 temp = 23

\*a = \*b; // 解引用 a的解引用 = b的解引用 m=57

\*b = temp; // n = 23

return 0;

}

7.6.2 数组名做函数参数

数组名做函数参数，传递的是数组首地址（一个指针）。

在整型数组作为函数参数时，需要再多封装一个元素个数。

#include <stdio.h>

void print\_arr(int arr[],int len) { // 函数定义

int i;

printf("sizeof(arr):%d\n", sizeof(arr)); // 整个数组大小 64位8；32位4

printf("sizeof(arr[0]):%d\n", sizeof(arr[0])); // 第零个数组内容的大小

// 数组名做函数参数，传递的是数组首地址（一个指针）。64位8；32位4

// 在整型数组作为函数参数时，需要再多封装一个元素个数。

// int len = sizeof(arr) / sizeof(arr[0]);

printf("print\_arr中的len：%d\n", len);

for ( i = 0; i < len; i++)

{

printf("%d\n",arr[i]);

}

}

int main(void)

{

int arr[] = { 1,2,3,4,5,6 };

int len = sizeof(arr) / sizeof(arr[0]); // 6个元素

printf("main中的len：%d\n",len);

// 需要一个打印数组内容的函数

print\_arr(arr, len); // 函数调用 .需要再多封装一个元素个数。

return EXIT\_SUCCESS;

}

7.6.3 指针为函数的返回值

#include <stdio.h>

// 定义一个相加函数

int\* my\_sum(int a,int b) {

int res;

res = a + b;

// 函数体内的值，在栈帧运行完毕后，会被标记为删除，

// 如果后续代码没有执行内存操作，此空间未被执行删除操作，

// 依然可以使用，但是！此做法不推荐！！！

return &res;

}

int main(void)

{

int a = 10;

int b = 20;

int\* res = NULL;// 初始存放空指针

res = my\_sum(a, b);

printf("res:%d\n",\*res);

return EXIT\_SUCCESS;

}

不允许把数组当成函数返回值，可以写成指针的形式。

7.7 指针和字符串

7.7.1 字符指针

* 字符串指针为只读，普通字符串为可读可写
* 字符串（字符数组）作为参数，不用提供元素个数，因为可以使用结束位置的‘\0’作为判断依据。

#include <stdio.h>

int *main*()

{

// 普通定义字符串

char str1[] = "hello";

char str3[] = "hello";

// char str2[] = { 'h','e','l','l','o'};

// 指针定义字符串

char\* str1\_zhi = "hello";

char\* str3\_zhi = "hello";

str1[0] = 'H';

// str1\_zhi[0] = 'H'; // 指针定义的字符串为常量，不允许被修改

//printf("str1：%s\n",str1);

//printf("str1\_zhi:%s\n",str1\_zhi);

// 打印普通字符串和指针字符串的地址

printf("str1地址:%p\n", str1);

printf("str3地址:%p\n", str3);

// 内容相同的指针定义的字符串，地址也一样

printf("str1\_zhi地址:%p\n", str1\_zhi);

printf("str3\_zhi地址:%p\n", str3\_zhi);

return 0;

}

7.7.2 指针数组做为main函数的形参

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

* argc传递参数总个数
* argv是字符串数组，每个成员都是char\*字符串类型，第零个是程序名



#include <stdio.h>

//argc: 传参数的个数

//argv：指针数组，指向输入的参数

int main(int argc, char\* argv[])

{

int i;

for (i = 0; i < argc; i++)

{

printf("argv[%d] = %s\n",i,argv[i]);

}

}

7.7.3 项目开发常用字符串应用模型

1) strstr中的while

利用strstr标准库函数找出一个字符串中substr出现的个数。

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int str\_times(char\* str,char\* son) {

int count = 0;

char\* p = strstr(str, son); // "lolo12lo"

while (p != NULL)

{

count++; // 匹配上了，执行++自增

p += strlen(son); //lo12lo

p = strstr(p, son); // 重新设置起点位置

}

return count;

}

int main(void)

{

// 子串在父串中出现次数

char\* str = "hellolo12lo";

char\* son = "lo";

int res = str\_times(str,son);

printf("res:%d\n", res);

return EXIT\_SUCCESS;

}

求字符串中非空格的元素个数：

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

int del\_space2(char\* str) {

int count = 0; // 计数器

//while (\*str != '\0') == while (\*str != 0) ==== while (\*str != false) while (\*str == true) === while (\*str)

while (\*str)

{

if (\*str != ' ') {

count++;

}

str++; // 指针每次循环都要向后移动一位

}

return count;

}

int main(void)

{

char\* str = "ni hao ya ha";

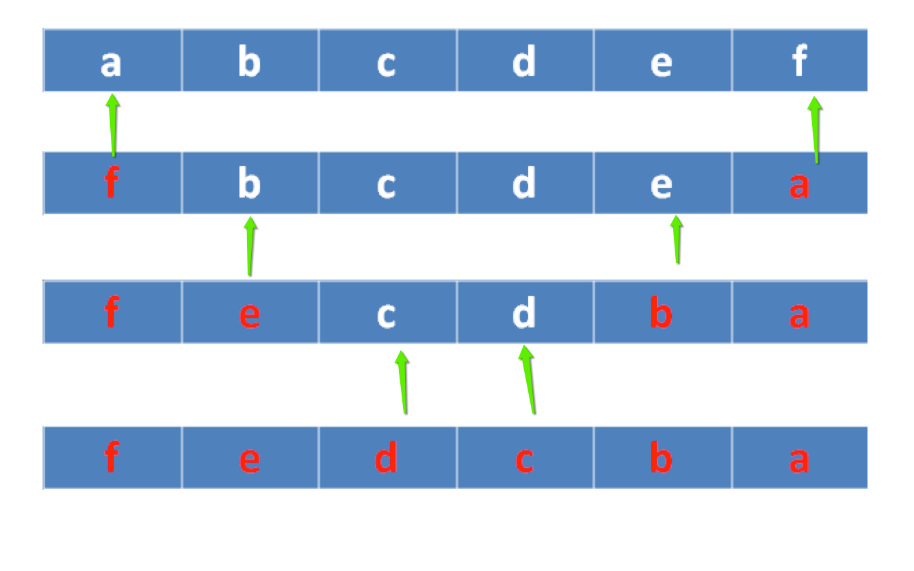
int res = del\_space2(str);

printf("res:%d\n", res);

return EXIT\_SUCCESS;

}

2) 两头堵模型：字符串反转模型(逆置)



#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void str\_inserse(char str[]) {

// hello

char\* start = str; // 保存首地址

char\* end = str + strlen(str) - 1; // 最后一个元素位置

while (start < end)

{

char temp = \*start;

\*start = \*end;

\*end = temp;

start++;

end--;

}

}

int main(void)

{

char str[] = "ni hao ya";

str\_inserse(str);

printf("str:%s", str);

return EXIT\_SUCCESS;

}

7.7.4 字符串处理函数

1) strcpy()

#include <string.h>

**char \*strcpy(char \*dest, const char \*src);**

功能：把src所指向的字符串复制到dest所指向的空间中，'\0'也会拷贝过去

参数：

dest：目的字符串首地址

src：源字符首地址

返回值：

成功：返回dest字符串的首地址

失败：NULL

注意：如果参数dest所指的内存空间不够大，可能会造成缓冲溢出的错误情况。

char src[] = "ni hao ya";

// '\0'也会拷贝过去

char dest[10] = { 0 }; // dest空间如果不够存放，就会报错

char\* res = strcpy(dest, src); // 把src的值复制给dest

// res和dest中保存的都是最终复制的结果

printf("res:%s\n", res);

printf("dest:%s\n", dest);

2) strncpy()

#include <string.h>

**char \*strncpy(char \*dest, const char \*src, *size\_t* n);**

功能：把src指向字符串的前n个字符复制到dest所指向的空间中，默认不添加'\0'。

参数：

dest：目的字符串首地址

src：源字符首地址

n：指定需要拷贝字符串个数

返回值：

成功：返回dest字符串的首地址

失败：NULL

char src[] = "ni hao ya";

// 默认不添加\0 ，一般设置dest和n的值相等（够用才行，否则会因为没有\0结束，而打印一些奇怪字符）

//char dest[9] = { 0 };

//char\* res = strncpy(dest, src,9); // 把src的值复制给dest

// n > src 只是把拷贝src的大小

//char dest[20] = { 0 };

//char\* res = strncpy(dest, src,20); // 把src的值复制给dest

// n < src 只拷贝n的大小 默认是不添加\0

char dest[5] = { 0 };

char\* res = strncpy(dest, src, 5); // 把src的值复制给dest

// res和dest中保存的都是最终复制的结果

printf("res:%s\n", res);

printf("dest:%s\n", dest);

3) strcat()

#include <string.h>

**char \*strcat(char \*dest, const char \*src);**

功能：将src字符串连接到dest的尾部，‘\0’也会追加过去

参数：

dest：目的字符串首地址

src：源字符首地址

返回值：

成功：返回dest字符串的首地址

失败：NULL

char src[] = "hehe";

char dest[8] = "gaga"; // 保证dest的空间够用

char\* res = strcat(dest, src);

printf("res:%s\n",res);

printf("dest:%s\n",dest);

4) strncat()

#include <string.h>

**char \*strncat(char \*dest, const char \*src, *size\_t* n);**

功能：将src字符串前n个字符连接到dest的尾部，‘\0’也会追加过去

参数：

dest：目的字符串首地址

src：源字符首地址

n：指定需要追加字符串个数

返回值：

成功：返回dest字符串的首地址

失败：NULL

char src[] = "hehe";

char dest[100] = "gaga"; // 保证dest空间够用

char\* res = strncat(dest, src,3); // 把src前3个添加到dest的后面

printf("res:%s\n", res);

printf("dest:%s\n", dest);

5) strcmp()

#include <string.h>

**int strcmp(const char \*s1, const char \*s2);**

功能：比较 s1 和 s2 的大小，比较的是字符ASCII码大小。

参数：

s1：字符串1首地址

s2：字符串2首地址

返回值：

相等：0

大于：>0 在不同操作系统strcmp结果会不同 返回ASCII差值

小于：<0

char\* str1 = "hehe";

char\* str2 = "aehe";

// str1 = str2 返回值就是0

// str1 > str2 返回值就是1

// str1 < str2 返回值就是-1

int res = strcmp(str1, str2);

printf("res:%d", res);

6) strncmp()

#include <string.h>

**int strncmp(const char \*s1, const char \*s2, *size\_t* n);**

功能：比较 s1 和 s2 前n个字符的大小，比较的是字符ASCII码大小。

参数：

s1：字符串1首地址

s2：字符串2首地址

n：指定比较字符串的数量

返回值：

相等：0

大于： > 0

小于： < 0

char \*str1 = "hello world";

char \*str2 = "hello mike";

if (*strncmp*(str1, str2, 5) == 0)

{

*printf*("str1==str2\n");

}

else if (*strcmp*(str1, "hello world") > 0)

{

*printf*("str1>str2\n");

}

else

{

*printf*("str1<str2\n");

}

7) sprintf()

#include <stdio.h>

**int sprintf(char \*str, const char \*format, ...);**

功能：根据参数format字符串来转换并格式化数据，然后将结果输出到str指定的空间中，直到出现字符串结束符 '\0' 为止。

参数：

str：字符串首地址

format：字符串格式，用法和printf()一样

返回值：

成功：实际格式化的字符个数

失败： - 1

char dst[100] = { 0 };

int a = 10;

char src[] = "hello world";

*printf*("a = %d, src = %s", a, src);

*printf*("\n");

int len = sprintf(dst, "a = %d, src = %s", a, src);

*printf*("dst = \" %s\"\n", dst);

*printf*("len = %d\n", len);

8) sscanf()

#include <stdio.h>

**int sscanf(const char \*str, const char \*format, ...);**

功能：从str指定的字符串读取数据，并根据参数format字符串来转换并格式化数据。

参数：

str：指定的字符串首地址

format：字符串格式，用法和scanf()一样

返回值：

成功：参数数目，成功转换的值的个数

失败： - 1

char src[] = "10 20";

int a;

int b;

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

sscanf(src, "%d %d", &a, &b);

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

9) strchr()

#include <string.h>

**char \*strchr(const char \*s, int c);**

**功能**：在字符串s中查找字母c出现的位置

**参数**：

s：字符串首地址

c：匹配字母(字符)

**返回值**：

成功：返回第一次出现的c地址

失败：NULL

char src[] = "ddda123abcd";

char \*p = *strchr*(src, 'a');

*printf*("p = %s\n", p);

10) strstr()

#include <string.h>

**char \*strstr(const char \*haystack, const char \*needle);**

功能：在字符串haystack中查找字符串needle出现的位置

参数：

haystack：源字符串首地址

needle：匹配字符串首地址

返回值：

成功：返回第一次出现的needle地址

失败：NULL

char src[] = "ddddabcd123abcd333abcd";

char \*p = *strstr*(src, "abcd");

*printf*("p = %s\n", p);

11) strtok()

#include <string.h>

**char \*strtok(char \*str, const char \*delim);**

功能：来将字符串分割成一个个片段。当strtok()在参数s的字符串中发现参数delim中包含的分割字符时, 则会将该字符改为\0 字符，当连续出现多个时只替换第一个为\0。

参数：

str：指向欲分割的字符串

delim：为分割字符串中包含的所有字符

返回值：

成功：分割后字符串首地址

失败：NULL

* 在第一次调用时：strtok()必需给予参数s字符串
* 往后的调用则将参数s设置成NULL，每次调用成功则返回指向被分割出片段的指针

char str[] = "www.baidu.com.cn"; // 不能用常量定义，因为后续要更改

// 此处务必使用双引号。结果是www\0baidu.com.cn 找到第一个匹配的位置，替换城\0

char\* res = strtok(str, ".");

//printf("res:%s\n", res); // www

while (res != NULL) // 返回值为NULL，代表没有拆分出任何内容了，代表结束，就可以跳出循环

{

printf("res:%s\n", res);

// 固定语法规则，第二次传NULL，代表把第一次拆分的结果传传递进来，再拆分

res = strtok(NULL, ".");

}

12) atoi()

#include <stdlib.h>

**int atoi(const char \*nptr);**

功能：atoi()会扫描nptr字符串，跳过前面的空格字符，直到遇到数字或正负号才开始做转换，而遇到非数字或字符串结束符('\0')才结束转换，并将结果返回返回值。

参数：

nptr：待转换的字符串

返回值：成功转换后整数

类似的函数有：

* atof()：把一个小数形式的字符串转化为一个浮点数。
* atol()：将一个字符串转化为long类型

char str1[] = "10";// 长得像int类型的字符串

char str2[] = "3.1a4F"; // 长得像float类型的字符串 3.10

char str3[] = "1234L"; // 长得像long int类型的字符串

// 遇到非数字或字符串结束符('\0')才结束转换

char str4[] = "1234ab123dc"; // 1234

int num1 = atoi(str1); // 转成整型

float num2 = atof(str2); // 转成浮点型

long int num3 = atol(str3); // 转成长整型

int num4 = atoi(str4); //

printf("num1:%d\n",num1);

printf("num2:%.2f\n",num2);

printf("num3:%ld\n",num3);

printf("num4:%d\n",num4);

8. 内存管理

8.1 作用域：生效范围；定义开始到包裹该变量的第一个大括号的结束

8.1.1 局部变量

局部变量也叫auto自动变量(auto可写可不写)，一般情况下代码块{}内部定义的变量都是自动变量，它有如下特点：

* 在一个函数内定义，只在函数范围内有效
* 在复合语句中定义，只在复合语句中有效
* 随着函数调用的结束或复合语句的结束局部变量的生命周期也结束
* 如果没有赋初值，内容为随机

#include <stdio.h>

void test()

{

//auto写不写是一样的

//auto只能出现在{}内部

auto int b = 10;

}

int *main*(void)

{

//b = 100; //err， 在main作用域中没有b

if (1)

{

//在复合语句中定义，只在复合语句中有效

int a = 10;

*printf*("a = %d\n", a);

}

//a = 10; //err离开if()的复合语句，a已经不存在

return 0;

}

8.1.2 全局变量

* 在函数外定义，可被本文件及其它文件中的函数所共用，若其它文件中的函数调用此变量,须用extern声明
* 全局变量的生命周期和程序运行周期一样
* 不同文件的全局变量不可重名

8.1.3 静态(static)全局变量

* 在函数外定义,作用范围被限制在所定义的文件中
* 不同文件静态全局变量可以重名,但作用域不冲突
* static全局变量的生命周期和程序运行周期一样，同时staitc全局变量的值只初始化一次

8.1.4 静态(static)局部变量

* static局部变量的作用域也是在定义的函数内有效
* static局部变量的生命周期和程序运行周期一样，同时staitc局部变量的值只初始化一次，但可以赋值多次
* static局部变量若未赋以初值，则由系统自动赋值：数值型变量自动赋初值0，字符型变量赋空字符

#include <stdio.h>

void fun1()

{

int i = 0;

i++;

*printf*("i = %d\n", i);

}

void fun2()

{

//静态局部变量，没有赋值，系统赋值为0，而且只会初始化一次

static int a;

a++;

*printf*("a = %d\n", a);

}

int *main*(void)

{

fun1();

fun1();

fun2();

fun2();

return 0;

}

8.1.5 extern全局变量声明

extern int a;声明一个变量，这个全局变量在别的文件中已经定义了，这里只是声明，而不是定义。

8.1.6 全局函数和静态函数

在C语言中函数默认都是全局的，使用关键字static可以将函数声明为静态，函数定义为static就意味着这个函数只能在定义这个函数的文件中使用，在其他文件中不能调用，即使在其他文件中声明这个函数都没用。

对于不同文件中的staitc函数名字可以相同。



注意：

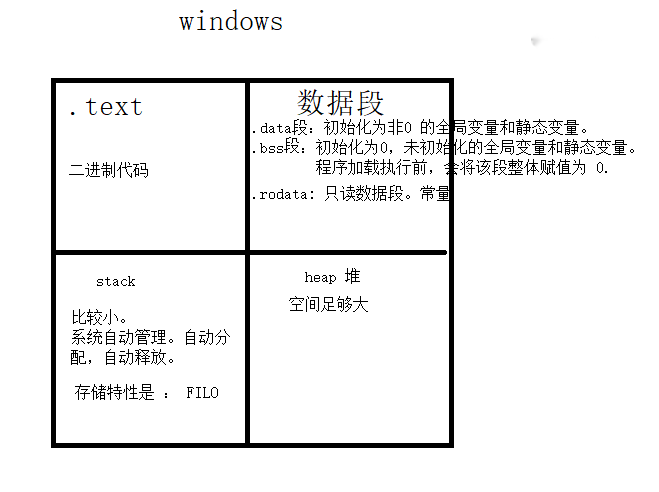
* 允许在不同的函数中使用相同的变量名，它们代表不同的对象，分配不同的单元，互不干扰。
* 同一源文件中,允许全局变量和局部变量同名，在局部变量的作用域内，全局变量不起作用。
* 所有的函数默认都是全局的，意味着所有的函数都不能重名，但如果是staitc函数，那么作用域是文件级的，所以不同的文件static函数名是可以相同的。

8.2 内存布局

8.2.1 内存分区

C代码经过预处理、编译、汇编、链接4步后生成一个可执行程序。

在 Windows 下，程序是一个普通的可执行文件。



* 代码区（text segment）

加载的是可执行文件代码段，所有的可执行代码都加载到代码区，这块内存是不可以在运行期间修改的。

* 未初始化数据区（BSS）

加载的是可执行文件BSS段，位置可以分开亦可以紧靠数据段，存储于数据段的数据（全局未初始化，静态未初始化数据）的生存周期为整个程序运行过程。

* 全局初始化数据区/静态数据区（data segment）

加载的是可执行文件数据段，存储于数据段（全局初始化，静态初始化数据，文字常量(只读)）的数据的生存周期为整个程序运行过程。

* 栈区（stack）

栈是一种先进后出的内存结构，由编译器自动分配释放，存放函数的参数值、返回值、局部变量等。在程序运行过程中实时加载和释放，因此，局部变量的生存周期为申请到释放该段栈空间。

* 堆区（heap）

堆是一个大容器，它的容量要远远大于栈，但没有栈那样先进后出的顺序。用于动态内存分配。堆在内存中位于BSS区和栈区之间。一般由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收。

8.2.2 存储类型总结

生命周期：数据在内存中存在的时间（能活多久~）

| **类型** | **作用域** | **生命周期** | **存储位置** |
| --- | --- | --- | --- |
| auto局部变量 | 一对{}内 | 函数内部 | 栈区 |
| 全局变量 | 整个程序 | 整个程序运行期 | 初始化在data段，未初始化在BSS段 |
| static局部变量 | 一对{}内 | 整个程序运行期 | 初始化在data段，未初始化在BSS段 |
| extern声明变量 | 整个程序 | 整个程序运行期 | 初始化在data段，未初始化在BSS段 |
| static全局变量 | 当前文件 | 整个程序运行期 | 初始化在data段，未初始化在BSS段 |
| 全局函数 | 整个程序 | 整个程序运行期 | 代码区 |
| static函数 | 当前文件 | 整个程序运行期 | 代码区 |
| 字符串常量 | 当前文件 | 整个程序运行期 | data段 |

8.2.3 堆区内存分配和释放（heap堆）

1)malloc()

#include <stdlib.h>

**void \**malloc*(*size\_t* size);**

功能：在内存的动态存储区(堆区)中分配一块长度为size字节的连续区域，用来存放类型说明符指定的类型。分配的内存空间内容不确定，一般使用memset初始化。

参数：

size：需要分配内存大小(单位：字节)

返回值：

成功：分配空间的起始地址

失败：NULL

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int *main*()

{

// 申请堆：参数要申请的空间大小，尽量强转为目标类型

// 当做数组来使用

//int arr[10000000] = { 10,23,54 }; // 默认申请的是栈空间，一但数据过多，就会报错

int\* p = (int \*)malloc(sizeof(int) \* 10);

int\* temp = p; // 记录p的初始地址

if (p == NULL) {

printf("申请失败%d", p);

return -1; //失败返回值设置为-1

}

else {

for (size\_t i = 0; i < 10; i++)

{

p[i] = i;

//\*(p + i) = i;

}

for (size\_t i = 0; i < 10; i++)

{

//printf("%d\n",\*(p+i));

//printf("%d\n",p[i]);

printf("%d\n",\*p);

p++; // p = p + 1; // 因为free的参数需要首地址，所以如果p发生偏移，free就会报错

}

}

// malloc和free对应，

free(temp);// 释放堆空间 ，可以删除temp里面的空间

p = NULL; // free后，内存空间不会立即消失，需要设置为NULL，才不可以再访问

for (size\_t i = 0; i < 10; i++)

{

//printf("%d\n",\*(p+i));

printf("%d\n",p[i]);

}

return 0;

}

2)free()

#include <stdlib.h>

**void *free*(void \*ptr);**

功能：释放ptr所指向的一块内存空间，ptr是一个任意类型的指针变量，指向被释放区域的首地址。对同一内存空间多次释放会出错。

参数：

ptr：需要释放空间的首地址，被释放区应是由malloc函数所分配的区域。

返回值：无

8.2.4内存操作函数

1) memset()

#include <string.h>

**void \*memset(void \*s, int c, *size\_t* n);**

功能：将s的内存区域的前n个字节以参数c填入

参数：

s：需要操作内存s的首地址

c：填充的字符，c虽然参数为int，但必须是unsigned char , 范围为0~255

n：指定需要设置的大小

返回值：s的首地址

int a[10];

//int\* p = (int \*)malloc(sizeof(int) \* 10);

////printf("sizeof\_p:%d\n",sizeof(p));// 默认占用空间就是64位8字节、32位4字节

//// 参数1：要操作的元素首地址、参数2：要设置的初始值 参数3：要设置的空间大小

//// 要注意默认malloc申请的空间就是8字节

//memset(p, 0, sizeof(int) \* 10); // 初始值为0

//for (size\_t i = 0; i < 10; i++)

//{

// printf("p:%d\n", p[i]);

//}

char\* p = (char \*)malloc(sizeof(char) \* 100);

//printf("sizeof\_p:%d\n", sizeof(p));

memset(p, 0, sizeof(char) \* 100); // 对于字符数组，设置第二个参数，表示ask码

for (size\_t i = 0; i < 10; i++)

{

printf("p:%d\n",p[i]);

}

free(p);

p = NULL;

2) memcpy()

#include <string.h>

**void \*memcpy(void \*dest, const void \*src, *size\_t* n);**

功能：拷贝src所指的内存内容的前n个字节到dest所值的内存地址上。

参数：

dest：目的内存首地址

src：源内存首地址，注意：dest和src所指的内存空间不可重叠，可能会导致程序报错

n：需要拷贝的字节数

返回值：dest的首地址

int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// strcpy和memcpy区别：

// strcpy可以用在字符串；memcpy用在内存复制

int arr[10] = {1,2,3,4,5,6};

int\* p = (int \*)malloc(sizeof(int) \* 6);

memcpy(p, arr, sizeof(int) \* 6);

// dest和src所指的内存空间不可重叠，可能会导致程序报错

//memcpy(&arr[2], arr, sizeof(int) \* 6);

for (size\_t i = 0; i < 6; i++)

{

printf("%d\n",p[i]);

}

free(p);

p = NULL;

3) memmove()

memmove()功能用法和memcpy()一样，区别在于：dest和src所指的内存空间重叠时，memmove()仍然能处理，不过执行效率比memcpy()低些。

4) memcmp()

#include <string.h>

**int memcmp(const void \*s1, const void \*s2, *size\_t* n);**

功能：比较s1和s2所指向内存区域的前n个字节。直到遇到第一个不相同的为止

参数：

s1：内存首地址1

s2：内存首地址2

n：需比较的前n个字节

返回值：

相等：0

大于：1

小于：-1

//数值

//int arr1[] = { 1,2,1,9,12 };

//int arr2[] = { 1,2,3,4,5 };

//int res = memcmp(arr1, arr2, sizeof(int) \* 5);

//字符 a:97 A:65

char\* ch1 = "hAazz";

char\* ch2 = "h";

int res = memcmp(ch1, ch2, sizeof(char) \* 5);

// 直到遇到第一个不相同的值为止，返回结果，a>b返回1 a<b返回-1 全部相等，返回0

printf("res:%d\n", res);

9. 复合类型(自定义类型)

9.1 结构体

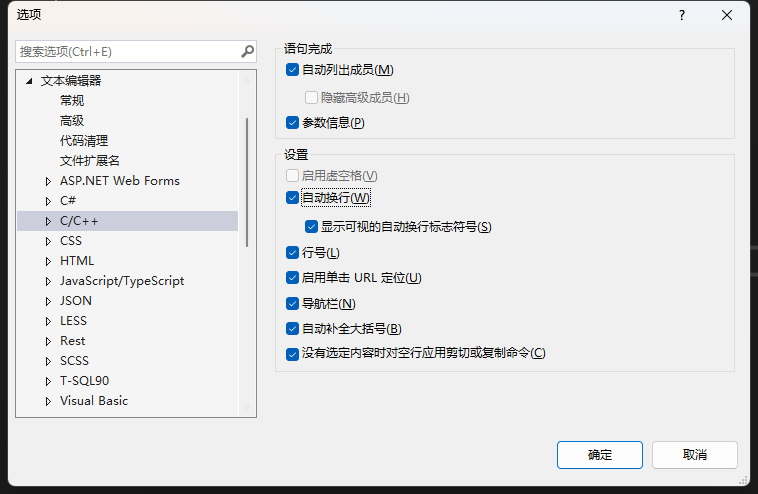
9.1.1 概述

数组：描述一组具有相同类型数据的有序集合，用于处理大量相同类型的数据运算。

有时我们需要将不同类型的数据组合成一个有机的整体，如：一个学生有学号/姓名/性别/年龄/地址等属性。显然单独定义以上变量比较繁琐，数据不便于管理。

Ｃ语言中给出了另一种构造数据类型——结构体。





9.1.2 结构体变量的定义和初始化

定义结构体：

struct 结构体名字

{

结构体成员列表

};

定义结构体变量：

struct 结构体名字 结构体变量名

赋值：

结构体名变量名.结构体成员列表 = 值

注意：方法3--赋值字符串，要使用strcpy实现

结构体类型和结构体变量关系：

* 结构体类型：指定了一个结构体类型，它相当于一个模型，但其中并无具体数据，系统对之也不分配实际内存单元。
* 结构体变量：系统根据结构体类型（内部成员状况）为之分配空间。

// 定义结构体：

//struct 结构体名字

//{

// 结构体成员列表

//};

//定义结构体变量：

//struct 结构体名字 结构体变量名

//

//赋值：

//结构体名变量名.结构体成员列表 = 值

//注意：方法3--赋值字符串，要使用strcpy实现

// 结构体最后要有分号结束

//struct students // 定义结构体

//{

// // 成员列表

// char name[20];

// unsigned int age;

// char tel[12];

// float scores[3];

// char gender;

//}stu1; // 方法4:定义结构体的同时，把结构体的变量名定义好

//方法5 定义和赋值一起

//struct students // 定义结构体

//{

// // 成员列表

// char name[20];

// unsigned int age;

// char tel[12];

// float scores[3];

// char gender;

//}stu1 = { "张三",18,"13311111111",98.5,60,78,'w' };

// 方法6：

struct students // 定义结构体

{

// 成员列表

char name[20];

unsigned int age;

char tel[12];

float scores[3];

char gender;

}stu1 = { .gender = 'w',.scores[0] = 99,.scores[1] = 98.5,.scores[2] = 86,.tel = "13311122222",.age = 18,.name = "赵四" };

int main(void)

{

// 定义结构体变量

// 方法1、按顺序赋值

//struct students stu1 = { "张三",18,"13311111111",98.5,60,78,'w' };

// 方法2、不按顺序赋值 .成员名 = 值

//struct students stu1 = { .gender = 'w',.scores[0] = 99,.scores[1] = 98.5,.scores[2] = 86,.tel = "13311122222",.age = 18,.name = "赵四" };

// 方法3、先定义出结构体的变量;再使用 变量名.成员们 = 值

//struct students stu1;

//stu1.name = "张三"; // 方法3给字符串数组赋值的时候要使用strcpy

//strcpy(stu1.name, "王五");

//stu1.age = 19;

//strcpy(stu1.tel, "13333333333");

//stu1.scores[0] = 98.5;

//stu1.scores[1] = 78.5;

//stu1.scores[2] = 68.5;

//stu1.gender = 'w';

// 方法4：

//strcpy(stu1.name, "王五");

//stu1.age = 19;

//strcpy(stu1.tel, "13333333333");

//stu1.scores[0] = 98.5;

//stu1.scores[1] = 78.5;

//stu1.scores[2] = 68.5;

//stu1.gender = 'w';

// 打印看结果

printf("姓名：%s\n", stu1.name);

printf("年龄：%d\n", stu1.age);

printf("电话：%s\n", stu1.tel);

printf("成绩1：%.1f 成绩2：%.1f 成绩3：%.1f\n", stu1.scores[0], stu1.scores[1], stu1.scores[2]);

// 一个中文占2字节，需要使用%s去匹配

printf("性别：%s\n", stu1.gender == 'm' ? "男":"女");

9.1.3 结构体空间大小

1、结构体定义的时候，变量成员的顺序会影响结构体的大小

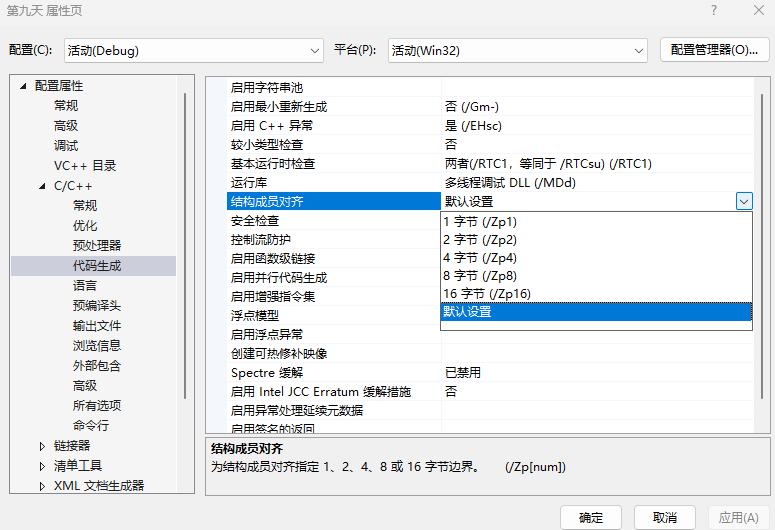
2、对齐:成员变量以什么样的方式，排列;紧密排列、还是松散中间是不是有间隔编译器对于对齐方式。vs默认是8字节对齐。

3、int类型:4字节，如果编译设置的对齐方式，大于4字节，那么就必须以字节的倍数分配地址

4、对齐方式>=4字节的时候，如果结构体里面有int，那么结构体的大小，就是4的整数倍。

5、如果有相同类型的变量，要放在一起，会减小结构体的空间

结论：建议结构体定义时变量类型从小到大的顺序，可以节省空间。但是数据量不大的时候，还是定义为人类更容易理解的顺序





9.1.4 结构体数组

#include <stdio.h>

struct students3

{

char name[20];

unsigned int age;

char tel[12];

float scores[3];

char gender;

};

int main(void)

{

struct students3 stu[2];

for (size\_t i = 0; i < 2; i++)

{

printf("请您输入 姓名 年龄 电话 成绩\*3 性别:\n");

scanf("%s %d %s %f %f %f %c",stu[i].name,&stu[i].age,stu[i].tel,&stu[i].scores[0], &stu[i].scores[1], &stu[i].scores[2],&stu[i].gender);

}

for (size\_t i = 0; i < 2; i++)

{

// 打印看结果

printf("姓名：%s\n", stu[i].name);

printf("年龄：%d\n", stu[i].age);

printf("电话：%s\n", stu[i].tel);

printf("成绩1：%.1f 成绩2：%.1f 成绩3：%.1f\n", stu[i].scores[0], stu[i].scores[1], stu[i].scores[2]);

// 一个中文占2字节，需要使用%s去匹配

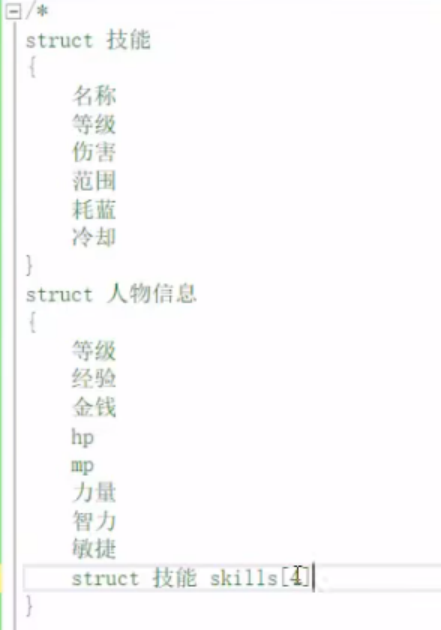
printf("性别：%s\n", stu[i].gender == 'm' ? "男" : "女");

}

return EXIT\_SUCCESS;

}

9.1.5 结构体套结构体



#include <stdio.h>

struct person

{

char name[20];

char sex;

};

struct stu

{

int id;

struct person info;

};

int *main*()

{

struct stu s[2] = { 1, "lily", 'F', 2, "yuri", 'M' };

int i = 0;

for (i = 0; i < 2; i++)

{

*printf*("id = %d\tinfo.name=%s\tinfo.sex=%c\n", s[i].id, s[i].info.name, s[i].info.sex);

}

return 0;

}

9.1.6 结构体和指针

1)结构体成员为指针

#include<stdio.h>

struct students8

{

char\* name; // 成员为指针

int age;

};

int main8(void)

{

struct students8 stu;

// 申请堆空间

stu.name = (char\*)malloc(sizeof(char) \* 20);

strcpy(stu.name, "张三");

stu.age = 18;

printf("姓名：%s 年龄：%d\n", stu.name, stu.age);

free(stu.name);

stu.name = NULL;

return EXIT\_SUCCESS;

2)堆空间创建结构体

#include<stdio.h>

#include <string.h>

#include <stdlib.h>

struct students10 {

char\* name;

int age;

};

int main(void)

{

struct students10 stu;

struct students10\* p = (struct students10 \*)malloc(sizeof(stu));

p->name = (char\*)malloc(sizeof(char) \* 20);

strcpy(p->name, "李四");

p->age = 20;

printf("姓名：%s 年龄：%d\n", p->name, p->age);

// 先清空内层的p->name 再清空外层的p

if (p->name) {

free(p->name);

p->name = NULL;

}

if (p) {

free(p);

p = NULL;

}

3)结构体为指针

#include<stdio.h>

#include <string.h>

#include <stdlib.h>

struct students9

{

char\* name;

int age;

};

int main10(void)

{

struct students9 stu;

struct students9\* p = &stu; // 结构体指针

// 写法1：结构体指针，找成员，使用 ->

//p->name = (char \*)malloc(sizeof(char) \* 20);

//strcpy(p->name, "张三");

//p->age = 19;

//printf("姓名：%s 年龄：%d\n",p->name,p->age);

//free(p->name);

//p->name = NULL;

//写法2： 但是写法1更简单，推荐使用写法1

(\*p).name = (char\*)malloc(sizeof(char) \* 20);

strcpy((\*p).name, "张三");

(\*p).age = 19;

printf("姓名：%s 年龄：%d\n", (\*p).name, (\*p).age);

free((\*p).name);

(\*p).name = NULL;

return EXIT\_SUCCESS;

}

9.1.7 结构体赋值

#include<stdio.h>

#include<string.h>

//结构体类型的定义

struct stu

{

char name[50];

int age;

};

int *main*()

{

struct stu s1;

//如果是普通变量，通过点运算符操作结构体成员

*strcpy*(s1.name, "abc");

s1.age = 18;

*printf*("s1.name = %s, s1.age = %d\n", s1.name, s1.age);

//相同类型的两个结构体变量，可以相互赋值

//把s1成员变量的值拷贝给s2成员变量的内存

//s1和s2只是成员变量的值一样而已，它们还是没有关系的两个变量

struct stu s2 = s1;

//memcpy(&s2, &s1, sizeof(s1));

*printf*("s2.name = %s, s2.age = %d\n", s2.name, s2.age);

return 0;

}

9.1.8 结构体做函数参数

1)结构体普通变量做函数参数

#include<stdio.h>

#include <string.h>

struct students14 {

char name[20];

int age;

};

void set\_stu(struct students14 stu) {

strcpy(stu.name, "李四");

stu.age = 20;

printf("姓名：%s 年龄：%d\n", stu.name, stu.age);

}

int main(void)

{

struct students14 stu;

strcpy(stu.name, "张三");

stu.age = 18;

set\_stu(stu); // 值传递，形参不影响实参的值

printf("姓名：%s 年龄：%d\n", stu.name, stu.age);

return EXIT\_SUCCESS;

}

2)结构体指针变量做函数参数

#include<stdio.h>

#include <string.h>

struct students15 {

char name[20];

int age;

};

void set\_stu15(struct students15\* stu) {

strcpy(stu->name, "李四");

stu->age = 20;

printf("姓名：%s 年龄：%d\n", stu->name, stu->age);

}

int main(void)

{

struct students15 stu;

strcpy(stu.name, "张三");

stu.age = 18;

set\_stu15(&stu); // 地址传递，修改同一块内存空间，内容会改变

printf("姓名：%s 年龄：%d\n", stu.name, stu.age);

return EXIT\_SUCCESS;

}

3)结构体数组名做函数参数

#include<stdio.h>

//结构体类型的定义

struct stu

{

char name[50];

int age;

};

//void set\_stu\_pro(struct stu tmp[100], int n)

//void set\_stu\_pro(struct stu tmp[], int n)

void set\_stu\_pro(struct stu \*tmp, int n)

{

int i = 0;

for (i = 0; i < n; i++)

{

sprintf(tmp->name, "name%d%d%d", i, i, i);

tmp->age = 20 + i;

tmp++;

}

}

int *main*()

{

struct stu s[3] = { 0 };

int i = 0;

int n = sizeof(s) / sizeof(s[0]);

set\_stu\_pro(s, n); //数组名传递

for (i = 0; i < n; i++)

{

*printf*("%s, %d\n", s[i].name, s[i].age);

}

return 0;

}

4)const修饰结构体指针形参变量

//结构体类型的定义

struct stu

{

char name[50];

int age;

};

void fun1(struct stu \* const p)

{

//p = NULL; //err

p->age = 10; //ok

}

//void fun2(struct stu const\* p)

void fun2(const struct stu \* p)

{

p = *NULL*; //ok

//p->age = 10; //err

}

void fun3(const struct stu \* const p)

{

//p = NULL; //err

//p->age = 10; //err

}

9.2 共用体(联合体)

* 联合union是一个能在同一个存储空间存储不同类型数据的类型；
* 联合体所占的内存长度等于其最长成员；
* 共用体变量的地址和它的各成员的地址都是同一地址。

#include <stdio.h>

union union1

{

char ch1;

int int1;

short sh1;

};

int main(void)

{

union union1 uni;

// 所有成员和联合体公用一个地址

printf("uni地址：%p\n",&uni);

printf("uni.ch1地址：%p\n",&uni.ch1);

printf("uni.int1地址：%p\n",&uni.int1);

printf("uni.sh1地址：%p\n",&uni.sh1);

// 按照成员最大的大小来定义联合体的大小

printf("sizof：%d\n",sizeof(uni));

// 一个成员赋值，会影响其它成员

uni.int1 = 0x12345678;

printf("uni.int1:%x\n", uni.int1);

printf("uni.sh1:%x\n", uni.sh1);

printf("uni.ch1:%x\n", uni.ch1);

uni.sh1 = 0x9999;

printf("uni.int1:%x\n", uni.int1);

printf("uni.sh1:%x\n", uni.sh1);

printf("uni.ch1:%x\n", uni.ch1);

return EXIT\_SUCCESS;

}

9.3 枚举

枚举：将变量的值一一列举出来，变量的值只限于列举出来的值的范围内。

枚举类型定义：

enum 枚举名

{

枚举值表

};

* 在枚举值表中应列出所有可用值，也称为枚举元素。
* 枚举值是常量，不能在程序中用赋值语句再对它赋值。
* 举元素本身由系统定义了一个表示序号的数值从0开始顺序定义为0，1，2 …

#include <stdio.h>

enum color {

red,

green = 6,

blue,

yellow = 2,

black

};

int main(void)

{

printf("red:%d\n",red);

printf("green:%d\n", green);

printf("blue:%d\n", blue);

printf("yellow:%d\n", yellow);

printf("black:%d\n", black);

return 0;

}

9.4 typedef

typedef为C语言的关键字，作用是为一种数据类型(基本类型或自定义数据类型)定义一个新名字，不能创建新类型。

* 与#define不同，typedef仅限于数据类型，而不是能是表达式或具体的值

10. 文件操作

10.1 概述

10.1.1设备文件和磁盘文件

* 设备文件

在操作系统中把每一个与主机相连的输入、输出设备看作是一个文件，把它们的输入、输出等同于对磁盘文件的读和写。（屏幕、键盘、硬盘、显卡。。）

* 磁盘文件

指一组相关数据的有序集合,通常存储在外部介质(如磁盘)上，使用时才调入内存。

10.1.2 磁盘文件的分类

计算机的存储在物理上是二进制的，所以物理上所有的磁盘文件本质上都是一样的：以字节为单位进行顺序存储。

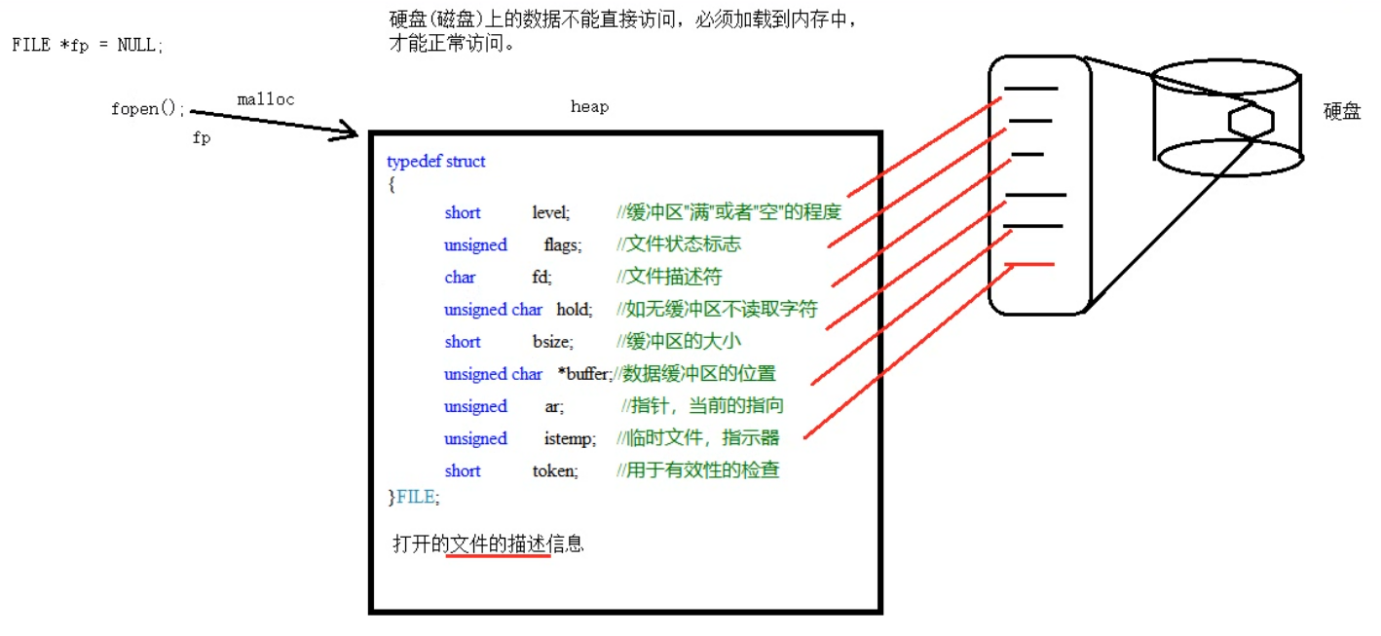
从用户或者操作系统使用的角度（逻辑上）把文件分为：

* 文本文件：基于字符编码的文件
* 二进制文件：基于值编码的文件（视频、音频。。）

10.2 文件的打开和关闭

10.2.1 文件指针

在C语言中用一个指针变量指向一个文件，这个指针称为文件指针。



typedef struct

{

short level; //缓冲区"满"或者"空"的程度

unsigned flags; //文件状态标志

char fd; //文件描述符

unsigned char hold; //如无缓冲区不读取字符

short bsize; //缓冲区的大小

unsigned char \*buffer;//数据缓冲区的位置

unsigned ar; //指针，当前的指向

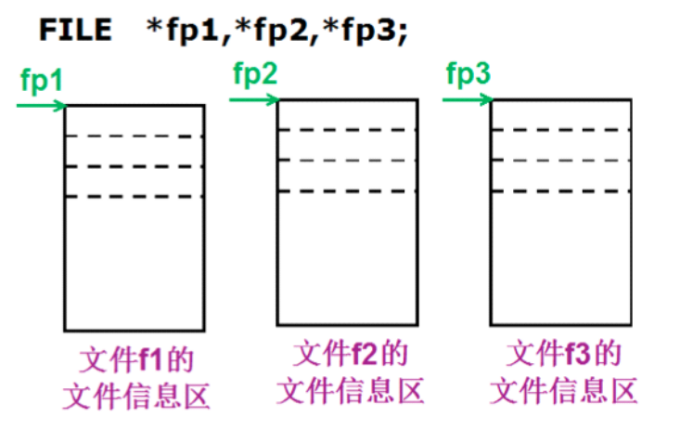
unsigned istemp; //临时文件，指示器

short token; //用于有效性的检查

}FILE;

FILE是系统使用typedef定义出来的有关文件信息的一种结构体类型，结构中含有文件名、文件状态和文件当前位置等信息。

声明FILE结构体类型的信息包含在头文件“stdio.h”中，一般设置一个指向FILE类型变量的指针变量，然后通过它来引用这些FILE类型变量。通过文件指针就可对它所指的文件进行各种操作。



C语言中有三个特殊的文件指针由系统默认打开，用户无需定义即可直接使用:

* stdin： 标准输入，默认为当前终端（键盘），我们使用的scanf等函数默认从此终端获得数据。
* stdout：标准输出，默认为当前终端（屏幕），我们使用的printf、puts函数默认输出信息到此终端。
* stderr：标准出错，默认为当前终端（屏幕），我们使用的perror函数默认输出信息到此终端。

10.2.2 文件的打开

任何文件使用之前必须打开：

#include <stdio.h>

**FILE \* fopen(const char \* filename, const char \* mode);**

功能：打开文件

参数：

filename：需要打开的文件名，根据需要加上路径

mode：打开文件的模式设置

返回值：

成功：文件指针

失败：NULL

第一个参数的几种形式:

*FILE* \*fp\_passwd = *NULL*;

//相对路径：

//打开当前目录passdw文件：源文件(源程序)所在目录

*FILE* \*fp\_passwd = *fopen*("passwd.txt", "r");

//打开当前目录(test)下passwd.txt文件

fp\_passwd = *fopen*(". / test / passwd.txt", "r");

//打开当前目录上一级目录（相对当前目录）passwd.txt文件

fp\_passwd = *fopen*(".. / passwd.txt", "r");

//绝对路径：

//打开C盘test目录下一个叫passwd.txt文件

fp\_passwd = *fopen*("c:/test/passwd.txt","r");

第二个参数的几种形式(打开文件的方式)：

| **打开模式** | **含义** |
| --- | --- |
| r或rb | 以只读方式打开一个文本文件（不创建文件，若文件不存在则报错） |
| w或wb | 以写方式打开文件(如果文件存在则清空文件，文件不存在则创建一个文件) |
| a或ab | 以追加方式打开文件，在末尾添加内容，若文件不存在则创建文件 |
| r+或rb+ | 以可读、可写的方式打开文件(不创建新文件、若文件不存在则报错) |
| w+或wb+ | 以可读、可写的方式打开文件(如果文件存在则清空文件，文件不存在则创建一个文件) |
| a+或ab+ | 以添加方式打开可读、可写的文件。若文件不存在则创建文件；如果文件存在，则写入的数据会被加到文件尾后，即文件原先的内容会被保留。 |

int main(void)

{

FILE \*fp = NULL;

// "\\"这样的路径形式，只能在windows使用

// "/"这样的路径形式，windows和linux平台下都可用，建议使用这种

// 路径可以是相对路径，也可是绝对路径

fp = fopen("../test", "w");

//fp = fopen("..\\test", "w");

if (fp == NULL) //返回空，说明打开失败

{

//perror()是标准出错打印函数，能打印调用库函数出错原因

perror("open");

return -1;

}

return 0;

}

10.2.3 文件的关闭

任何文件在使用后应该关闭：

* 打开的文件会占用内存资源，如果总是打开不关闭，会消耗很多内存
* 一个进程同时打开的文件数是有限制的，超过最大同时打开文件数，再次调用fopen打开文件会失败
* 如果没有明确的调用fclose关闭打开的文件，那么程序在退出的时候，操作系统会统一关闭。

#include <stdio.h>

**int fclose(FILE \* stream);**

功能：关闭先前fopen()打开的文件。此动作让缓冲区的数据写入文件中，并释放系统所提供的文件资源。

参数：

stream：文件指针

返回值：

成功：0

失败：-1

*FILE* \* fp = *NULL*;

fp = *fopen*("abc.txt", "r");

*fclose*(fp);

10.3 文件的顺序读写

10.3.1 按照字符读写文件fgetc、fputc

#### 1）写文件

#include <stdio.h>

**int fputc(int ch, FILE \* stream);**

功能：将ch转换为unsigned char后写入stream指定的文件中

参数：

ch：需要写入文件的字符

stream：文件指针

返回值：

成功：成功写入文件的字符对应的ask码

失败：返回-1

// 两种写法都可以

//char\* filename = "text05.txt";

char filename[] = "text05.txt";

FILE\* fp = fopen(filename, "w");

if (fp == NULL) {

perror("错误是");

return -1;

}

int res = fputc('a', fp); // 返回值为添加内容对应的ask码

printf("res: %d\n",res);

fclose(fp);

printf("成功");

#### 2）文件结尾

在C语言中，EOF表示文件结束符(end of file)。在while循环中以EOF作为文件结束标志，这种以EOF作为文件结束标志的文件，必须是文本文件。在文本文件中，数据都是以字符的ASCII代码值的形式存放。我们知道，ASCII代码值的范围是0~127，不可能出现-1，因此可以用EOF作为文件结束标志。

#define *EOF* (-1)

当把数据以二进制形式存放到文件中时，就会有-1值的出现，因此不能采用EOF作为二进制文件的结束标志。为解决这一个问题，ANSI C提供一个feof函数，用来判断文件是否结束。feof函数既可用以判断二进制文件又可用以判断文本文件。

#include <stdio.h>

**int feof(FILE \* stream);**

功能：检测是否读取到了文件结尾。

参数：

stream：文件指针

返回值：

非0值：已经到文件结尾

0：没有到文件结尾

#### 3）读文件

#include <stdio.h>

**int fgetc(FILE \* stream);**

功能：从stream指定的文件中读取一个字符

参数：

stream：文件指针

返回值：

成功：返回读取到的字符对应ask码

失败：-1

char ch;

char ch = 0;

FILE\* fp = fopen("text05.txt", "r");

if (fp == NULL) {

printf("错误是");

return -1;

}

while (1)

{

ch = fgetc(fp);

if (ch == EOF) { // EOF 代表文件的结束

break;

}

printf("%d\n",ch);

}

fclose(fp);

10.3.2按照行读写文件fgets、fputs

#### 1）写文件

#include <stdio.h>

**int fputs(const char \* str, FILE \* stream);**

功能：将str所指定的字符串写入到stream指定的文件中，字符串结束符 '\0' 不写入文件。

参数：

str：字符串

stream：文件指针

返回值：

成功：0

失败：-1

char \*buf[] = { "123456\n", "bbbbbbbbbb\n", "ccccccccccc\n" };

for (size\_t i = 0; i < 3; i++)

{

fputs(buf[i], fp);

}

#### 2）读文件

#include <stdio.h>

**char \* fgets(char \* str, int size, FILE \* stream);**

功能：从stream指定的文件内读入字符，保存到str所指定的内存空间，直到出现换行字符、读到文件结尾或是已读了size - 1个字符为止，最后会自动加上字符 '\0' 作为字符串结束。

参数：

str：字符串

size：指定最大读取字符串的长度（size - 1）

stream：文件指针

返回值：

成功：成功读取的字符串

读到文件尾或出错： NULL

// 按行读取

FILE\* fp = fopen("text05.txt", "r");

if (fp == NULL) {

printf("错误是");

return -1;

}

char arr[100] = { 0 };

while (!feof(fp)) // 判断文件没有到达结束位置，就一直读取

{

char\* p = fgets(arr, sizeof(arr), fp);

if (p != NULL) { // 只要读取内容不是NULL, 没有读取完毕

printf("arr:%s",arr);

}

}

10.3.3按照格式化文件fprintf、fscanf

#### 1）写文件

#include <stdio.h>

**int fprintf(FILE \* stream, const char \* format, ...);**

功能：根据参数format字符串来转换并格式化数据，然后将结果输出到stream指定的文件中，指定出现字符串结束符 '\0' 为止。

参数：

stream：已经打开的文件

format：字符串格式，用法和printf()一样

返回值：

成功：实际写入文件的字符个数

失败：-1

int main8(void)

{

FILE\* fp = fopen("text05.txt", "w");

// fp ==NULL

if (!fp) { // fp==false fp != true !fp == true !fp

perror("错误是");

return -1;

}

fprintf(fp,"%d %c %d = %d",2,'\*',5,10);

return EXIT\_SUCCESS;

}

#### 读文件

#include <stdio.h>

**int fscanf(FILE \* stream, const char \* format, ...);**

功能：从stream指定的文件读取字符串，并根据参数format字符串来转换并格式化数据。

参数：

stream：已经打开的文件

format：字符串格式，用法和scanf()一样

返回值：

成功：参数数目，成功转换的值的个数

失败： - 1

int a, b, c;

char ch;

FILE\* fp = fopen("text05.txt", "r");

if (!fp) {

perror("错误是");

return -1;

}

fscanf(fp, "%d %c %d = %d", &a, &ch, &b, &c);

printf("%d %c%d = %d",a,ch,b,c);

fclose(fp);

10.3.4按照块读写文件fread、fwrite（二进制）

#### 1）写文件

#include <stdio.h>

**size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);**

功能：以数据块的方式给文件写入内容

参数：

ptr：准备写入文件数据的地址

size： size\_t 为 unsigned int类型，此参数指定写入文件内容的块数据大小。一般写 1

nmemb：写入文件的块数，一般写sizeof(xx)。写入文件数据总大小为：size \* nmemb

stream：已经打开的文件指针

返回值：

成功：实际成功写入文件数据的块数目（nmemb 的值）

失败：0

struct students10

{

int id;

char name[20];

int age;

};

int main10(void)

{

struct students10 stu[4] = {

{1,"张三",18},

{2,"张四",28},

{3,"张五",38},

{4,"张六",48 }

};

FILE\* fp = fopen("test06.txt", "w");

if (!fp) {

perror("错误是");

return -1;

}

int res = fwrite(stu, 1, sizeof(stu), fp);

if (res == 0) {

perror("错误是");

return -1;

}

fclose(fp);

return EXIT\_SUCCESS;

}

#### 2）读文件

#include <stdio.h>

**size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);**

功能：以数据块的方式从文件中读取内容

参数：

ptr：存放读取出来数据的内存空间

size： size\_t 为 unsigned int类型，此参数指定读取文件内容的块数据大小

nmemb：读取文件的块数，读取文件数据总大小为：size \* nmemb

stream：已经打开的文件指针

返回值：

成功：实际成功读取到内容的块数

失败：0

struct students11

{

int id;

char name[20];

int age;

};

int main11(void)

{

struct students11 stu[4];

FILE\* fp = fopen("test06.txt", "r");

if (!fp) {

perror("错误是");

return -1;

}

int res = fread(stu,1,sizeof(stu),fp);

if (res == 0) {

perror("错误是");

return -1;

}

printf("res:%d\n");

for (size\_t i = 0; i < 4; i++)

{

printf("id:%d name:%s age:%d\n",stu[i].id, stu[i].name, stu[i].age);

}

fclose(fp);

return EXIT\_SUCCESS;

}

10.4 文件的随机读写

#include <stdio.h>

**int fseek(FILE \*stream, long offset, int whence);**

功能：移动文件流（文件光标）的读写位置。

参数：

stream：已经打开的文件指针

offset：根据whence来移动的位移数（偏移量：矢量），可以是正数，也可以负数，如果正数，则相对于whence往右移动，如果是负数，则相对于whence往左移动。

whence：其取值如下：

SEEK\_SET：从文件开头移动offset个字节

SEEK\_CUR：从当前位置移动offset个字节

SEEK\_END：从文件末尾移动offset个字节

返回值：

成功：0

失败：-1

#include <stdio.h>

**long ftell(FILE \*stream);**

功能：获取文件流（文件光标）的读写位置到起始位置的偏移量

参数：

stream：已经打开的文件指针

返回值：

成功：当前文件流（文件光标）的读写位置的偏移量

失败：-1

#include <stdio.h>

**void rewind(FILE \*stream);**

功能：把文件流（文件光标）的读写位置移动到文件开头。

参数：

stream：已经打开的文件指针

返回值：

无返回值

struct students13

{

int id;

char name[20];

int age;

};

int main(void)

{

struct students13 stu[4] = {

{1,"张三",18},

{2,"张四",28},

{3,"张五",38},

{4,"张六",48}

};

printf("结构体一个元素大小：%d\n",sizeof(struct students13));

printf("结构体总大小：%d\n",sizeof(stu));

FILE\* fp = fopen("test06.txt", "r");

if (!fp) {

perror("错误是");

return -1;

}

// 改变读取光标的位置

// 从头开始读

fseek(fp, sizeof(struct students13) \* 3, SEEK\_SET);

// 从结尾开始读，别忘了矢量值变复数，向左读取

//fseek(fp, sizeof(struct students13) \* -3, SEEK\_END);

// 当前光标位置读取

//fseek(fp, sizeof(struct students13), SEEK\_CUR);

// 让光标返回起始位置

rewind(fp);

// 光标位置到起始位置的偏移量

int offset\_len = ftell(fp);

printf("offset\_len:%d\n", offset\_len);

// 读取

int res = fread(stu, 1, sizeof(stu), fp);

if (res == 0) {

perror("错误是");

return -1;

}

printf("id:%d name:%s age:%d\n", stu[0].id, stu[0].name, stu[0].age);

return EXIT\_SUCCESS;

}

10.5 Windows和Linux文本文件区别

* b是二进制模式的意思，b只是在Windows有效，在Linux用r和rb的结果是一样的
* Unix和Linux下所有的文本文件行都是\n结尾，而Windows所有的文本文件行都是\r\n结尾
* 在Windows平台下，以“文本”方式打开文件，不加b：
  + 当读取文件的时候，系统会将所有的 "\r\n" 转换成 "\n"
  + 当写入文件的时候，系统会将 "\n" 转换成 "\r\n" 写入
  + 以"二进制"方式打开文件，则读\写都不会进行这样的转换
* 在Unix/Linux平台下，“文本”与“二进制”模式没有区别，"\r\n" 作为两个字符原样输入输出

10.6 获取文件状态

#include <sys/types.h>

#include <sys/stat.h>

**int *stat*(const char \*path, struct *stat* \*buf);**

功能：获取文件状态信息

参数：

path：文件名

buf：保存文件信息的结构体

返回值：

成功：0

失败-1

struct stat {

*dev\_t* *st\_dev*; //文件的设备编号

*ino\_t* *st\_ino*; //节点

mode\_t *st\_mode*; //文件的类型和存取的权限

nlink\_t *st\_nlink*; //连到该文件的硬连接数目，刚建立的文件值为1

uid\_t *st\_uid*; //用户ID

gid\_t *st\_gid*; //组ID

*dev\_t* *st\_rdev*; //(设备类型)若此文件为设备文件，则为其设备编号

*off\_t* *st\_size*; //文件字节数(文件大小)

unsigned long st\_blksize; //块大小(文件系统的I/O 缓冲区大小)

unsigned long st\_blocks; //块数

*time\_t* *st\_atime*; //最后一次访问时间

*time\_t* *st\_mtime*; //最后一次修改时间

*time\_t* *st\_ctime*; //最后一次改变时间(指属性)

};

// 想使用stat 需要引入以下两个包

#include <sys/types.h>

#include <sys/stat.h>

int main(void)

{

// 不用打开文件，运行效率高

struct stat buf; // stat 空结构体

int res = stat("test06.txt", &buf);

printf("文件大小：%d\n", buf.st\_size);

return EXIT\_SUCCESS;

}

10.7 删除文件、重命名文件名

#include <stdio.h>

**int remove(const char \*pathname);**

功能：删除文件

参数：

pathname：文件名

返回值：

成功：0

失败：-1

#include <stdio.h>

**int rename(const char \*oldpath, const char \*newpath);**

功能：把oldpath的文件名改为newpath

参数：

oldpath：旧文件名

newpath：新文件名

返回值：

成功：0

失败： - 1

10.8 文件缓冲区

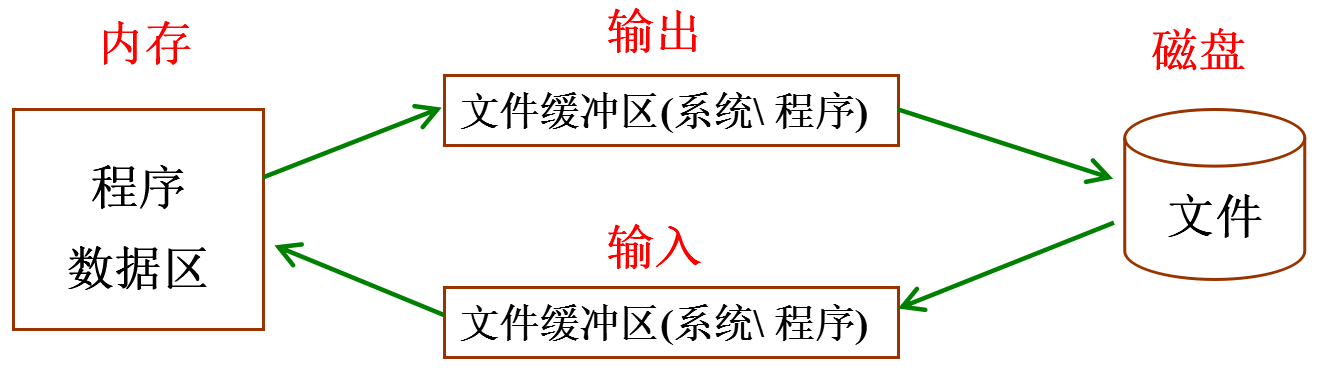
10.8.1文件缓冲区

ANSI C标准采用“缓冲文件系统”处理数据文件。

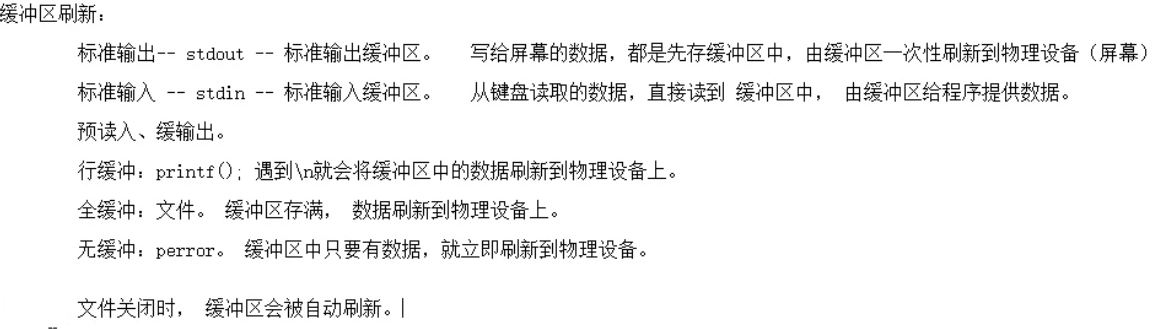
所谓缓冲文件系统是指系统自动地在内存区为程序中每一个正在使用的文件开辟一个文件缓冲区从内存向磁盘输出数据必须先送到内存中的缓冲区，装满缓冲区后才一起送到磁盘去。

如果从磁盘向计算机读入数据，则一次从磁盘文件将一批数据输入到内存缓冲区(充满缓冲区)，然后再从缓冲区逐个地将数据送到程序数据区(给程序变量) 。

10.8.2磁盘文件的存取



* 磁盘文件，一般保存在硬盘、U盘等掉电不丢失的磁盘设备中，在需要时调入内存
* 在内存中对文件进行编辑处理后，保存到磁盘中
* 程序与磁盘之间交互，不是立即完成，系统或程序可根据需要设置缓冲区，以提高存取效率



10.8.3更新缓冲区

#include <stdio.h>

**int *fflush*(*FILE* \*stream);**

功能：更新缓冲区，让缓冲区的数据立马写到文件中。

参数：

stream：文件指针

返回值：

成功：0

失败：-1