

目录

第 1 章 Linux 基础之命令.....	6
1.1 命令的格式.....	6
1.2 命令帮助信息.....	6
1.3 man 命令.....	6
1.4 命令常用的技巧.....	7
1.4.1 自动补全.....	7
1.4.2 历史命令.....	7
1.4.3 重定向.....	7
1.4.4 管道.....	7
1.5 ls tree clear.....	7
1.5.1 ls 查看目录下的信息.....	7
1.5.2 tree 以树状结构显示目录信息.....	7
1.5.3 clear 清屏.....	8
1.6 cd pwd.....	8
1.6.1 cd : change directory 改变目录的意思.....	8
1.6.2 pwd.....	8
1.7 cat rm cp mv.....	8
1.7.1 cat.....	8
1.7.2 rm.....	8
1.7.3 cp 是 copy 复制的意思.....	9
1.7.4 mv 是 move 是移动的意思, 即剪切的意思.....	9
1.8 mkdir touch.....	9
1.8.1 mkdir 创建文件夹.....	9
1.8.2 touch 创建文件.....	9
1.9 find: 在指定路径下查找指定文件.....	9
1.10 grep: 在指定文件中查找指定内容.....	10
1.11 ln: 创建链接文件.....	10
1.12 tar: 压缩与解压缩命令.....	10
1.12.1 gzip 格式:.....	10
1.12.2 bz2 格式:.....	11
1.13 编辑器之神 ---vi.....	11
1.13.1 安装 vim (vi 升级版).....	11

1.13.2 vi 的使用	11
1.13.3 vi 的三种模式	12
1.14 编译器 gcc	13
第 2 章 c 数据类型及语句	14
2.1 关键字	14
2.1.1 数据类型相关的关键字	14
2.1.2 存储相关关键字	17
2.1.3 控制语句相关的关键字	18
2.1.4 其他关键字	18
2.2 数据类型	20
2.2.1 基本类型	20
2.2.2 构造类型	20
2.2.3 类型转换	22
2.2.4 指针	25
2.3 运算符	25
2.3.1 运算符	25
2.3.2 运算符的分类:	25
2.3.3 算数运算符	26
2.3.4 关系运算符	26
2.3.5 逻辑运算符	26
2.3.6 位运算符	26
2.3.7 条件运算符	28
2.3.8 逗号运算符 ,	28
2.4 控制语句相关关键字讲解	31
2.4.1 选择控制语句相关的关键字	31
2.4.2 循环控制语句相关的关键字	35
第 3 章 数组	38
3.1 数组的概念	38
3.2 数组的分类	39
3.2.1 按元素的类型分类	39
3.2.2 按维数分类	39
3.3 数组的定义	39
3.3.1 一维数组的定义	39

3.3.2 二维数组的定义.....	40
3.4 定义并初始化.....	41
3.4.1 一维数组的初始化.....	41
3.4.2 二维数组的定义并初始化.....	41
3.5 数组元素的引用方法.....	42
3.5.1 字符数组的定义.....	43
第4章 函数.....	43
4.1 函数的概念.....	43
4.2 函数的分类.....	44
4.3 函数的定义.....	46
4.4 函数的声明.....	47
4.5 函数的调用.....	49
4.6 函数总结.....	50
4.7 变量的存储类别.....	50
4.7.1 内存的分区：.....	50
4.7.2 普通的全局变量.....	51
4.7.3 静态全局变量 <code>static</code>	51
4.7.4 普通的局部变量.....	52
4.7.5 静态的局部变量.....	52
4.7.6 外部函数.....	53
4.7.7 内部函数.....	53
第5章 预处理、动态库、静态库.....	54
5.1 c 语言编译过程.....	54
5.2 <code>include</code>	54
5.3 <code>define</code>	55
5.4 选择性编译.....	57
5.5 静态库.....	58
5.6 动态库.....	60
第6章 指针.....	61
6.1 指针的概念.....	61
6.1.1 关于内存那点事.....	61
6.1.2 指针的相关概念.....	62
6.1.3 指针的定义方法.....	63
6.1.4 指针的分类.....	65
6.1.5 指针和变量的关系.....	66
6.1.6 指针和数组元素之间的关系.....	68

6.1.7 指针数组.....	71
6.1.8 指针的指针.....	72
6.1.9 字符串和指针.....	73
6.1.10 数组指针.....	75
6.1.11 指针和函数的关系.....	80
6.1.12 经常容易混淆的指针.....	88
6.1.13 特殊指针.....	89
第7章 动态内存申请.....	90
7.1 动态分配内存的概述.....	90
7.2 静态分配、动态分配.....	90
7.3 动态分配函数.....	91
7.4 内存泄露.....	94
第8章 字符串处理函数.....	96
8.1 测字符串长度函数.....	96
8.2 字符串拷贝函数.....	97
8.3 格式化字符串操作函数.....	103
8.4 const:.....	105
第9章 结构体.....	106
9.1 结构体类型的概念及定义.....	106
9.2 结构体变量的定义初始化及使用.....	108
9.3 结构体数组.....	111
9.4 结构体指针.....	112
9.5 结构体内存分配.....	115
9.6 位段.....	121
第10章 链表.....	128
10.1 链表的概念.....	128
10.2 链表的创建.....	130
10.3 链表的遍历.....	130
10.4 链表的释放.....	130
10.5 链表节点的查找.....	130
10.6 链表节点的删除.....	130
10.7 链表中插入一个节点.....	130
10.8 链表排序.....	130
10.9 链表逆序.....	130
10.10 双向链表.....	130
10.10.1 双向链表的创建.....	130
10.10.2 双向链表节点的删除.....	130
10.10.3 双向链表插入节点.....	130
第11章 文件.....	130

11.1 文件的概念	130
11.1.1 文件的定义:	131
11.1.2 磁盘文件的分类:	133
11.2 文件指针.....	135
11.3 打开文件 fopen.....	136
11.4 关闭文件 fclose.....	138
11.5 一次读写一个字符.....	139
11.6 一次读写一个字符串	141
11.7 读文件 fread	142
11.8 写文件 fwrite	142
11.9 格式化读写文件函数	143
11.10 随机读写.....	144
第 12 章 Makefile	146
12.1 make 概述.....	146
12.2 makefile 语法及其执行	147
12.2.1 makefile 语法规则.....	147
12.2.2 make 命令格式.....	147
12.2.3 假想目标:.....	148
12.3 makefile 变量	149
12.3.1 makefile 变量概述.....	149
12.3.2 makefile 的变量分类:	149
12.3.3 自定义变量语法.....	149
12.3.4 系统环境变量	150
12.3.5 预定义变量.....	151

第1章 Linux 基础之命令

1、目录结构及文件

2、Linux 命令

平常通过图形界面能够实现的功能，通过命令基本上都可以实现

1.1 命令的格式

命令 选项 参数

```
ls -a -l -h /
```

```
rm -r -f a.txt b.txt
```

1.2 命令帮助信息

命令 --help

注意：并不是所有的命令都有帮助信息 pwd 就没有

1.3 man 命令

man 是 Linux 提供的一个手册，这个手册有很多个章节

第一章中放的是 命令 的帮助信息

第二章中放的是系统调用（操作系统提供的函数 open）

第三章中放的是库函数（c 库提供的函数 printf）的帮助信息

用法：

man 章节 查找信息

```
man 1 ls
```

```
man 2 open
```

```
man 3 printf
```

注意：如果有命令和系统调用及库函数同名的话，在各个章节中找就行

注意：命令中间的章节可以没有，如果没有的话，就从第 1 章开始往后找，直到找到或者找完。 man ls

1.4 命令常用的技巧

1.4.1 自动补全

输入命令的一部分，然后按 `tab` 键，就自动补全命令，如果有多个相同开头的命令多按一次 `tab` 键，他会将所有的以你输入开头的命令提示出来

例：`ifconfig`

输入 `if` 按两下 `tab` 键

输入 `ifc` 按 `tab` 键

注意：目录或文件的名字 也可以自动补全

在当前目录下有个文件夹叫 `zhaoxiutao`

我要删除文件夹得话

```
rm zhao
```

然后按 `tab` 键就可以补全目录

1.4.2 历史命令

可以按 上下键 将以前输入的命令调出来，敲回车执行或者修改之后敲回车执行

1.4.3 重定向

命令 > 目的文件

本来输出到屏幕上的数据，重定向输出 到了 目的文件中

1.4.4 管道

一个程序的输出 作为另外一个程序的输入

```
ls --help | more
```

```
ls --help | less
```

1.5 ls tree clear

1.5.1 ls 查看目录下的信息

```
ls -a -l -h /
```

1.5.2 tree 以树状结构显示目录信息

系统默认没有这个命令，需要安装 `sudo apt-get install tree`

用法：`tree -L` 层数

```
tree -L 3
```

1.5.3 clear 清屏

1.6 cd pwd

1.6.1 cd : change directory 改变目录的意思

绝对路径:

从根目录（最顶层目录开始的路径）

```
/home/teacher/xiutao/test
```

```
/home/teacher/test.txt
```

相对路径:

相对当前路径下的那个路径

```
./xiutao/test
```

```
./xiutao/test.c
```

当前目录 : .

上一级路径 : ..

```
cd /home/teacher/xiutao
```

如果我的工作路径在 teacher

cd ./xiutao 或者 cd xiutao 也是可以的

cd - 回到刚才所在的路径（相当于 windows 下的后退）

cd .. 切换到上一级路径（相当于 windows 下的返回上一级）

cd ~ 回到家目录下

1.6.2 pwd

显示当前工作目录的绝对路径

1.7 cat rm cp mv

1.7.1 cat

cat : 显示文本文件内容

使用方法 : cat 文件路径或名称

```
cat test.txt
```

```
cat /home/teacher/test.txt
```

1.7.2 rm

rm 是 remove 的意思，移除 即删除的意思

rm 选项 文件夹或文件的路径或名称

```
rm test.c
```

```
rm -rf xiutao
```

```
rm -rf /home/teacher/xiutao
```

注意：选项 `-rf` 可以没有，但是删除文件夹的时候必须带`-r`

`-f` 的意思是强制删除，即使要删除的文件或文件夹不存在也不报错

1.7.3 cp 是 copy 复制的意思

`cp` 选项 源文件或文件夹路径或者名称 目的路径

```
cp -r -v test.txt /home/teacher/xiutao
```

将当前目录下的 `test.txt` 文件拷贝到 `/home/teacher/xiutao` 目录下

注意：`cp` 后面可以加选项

拷贝文件夹的时候必须加`-r`，

`-v` 显示拷贝进度

1.7.4 mv 是 move 是移动的意思，即剪切的意思

使用方法：

`mv` 选项 要剪切的文件或目录 目的路径

```
mv -v test.txt /home/teacher/xiutao
```

注意：

如果源文件和目的路径在同一个路径下，相当于重命名

```
mv test.txt test2.txt
```

相当于给 `test.txt` 重命名为 `test2.txt`

1.8 mkdir touch

1.8.1 mkdir 创建文件夹

例：`mkdir xiutao`

注意：如过递归创建的话，需要加`-p`

```
mkdir -p xiutao/yunfeng/xuegang/lize
```

所在的目录必须可写，即允许创建文件夹

1.8.2 touch 创建文件

```
touch test.txt
```

1.9 find: 在指定路径下查找指定文件

用法：`find` 路径 `-name` 文件名

```
find /home/teacher -name xuegang.txt
```

在 `/home/teacher` 下及其子目录下找 `xuegang.txt`

1.10 grep: 在指定文件中查找指定内容

用法: `grep` 查找信息 文件名 参数 `-n` 显示行号

```
grep hello /home/teacher/test.txt -n
```

意思是在 `/home/teacher/test.txt` 文件中找 `hello` 字符串, 显示出来并带行号

例: 在某个路径及其子目录下的所有文件中查找字符串

```
grep hello /home/teacher/* -R -n
```

在 `/home/teacher/` 下的文件及其子目录下文件中查找 `hello` 字符串

注意: 必须加 `-R` 递归查找子目录, `-n` 是用来控制显示行号

1.11 ln: 创建链接文件

`ln` 源文件名字 连接文件 (快捷方式) 的名字 `-s`

```
ln test.txt test2.txt -s
```

注: `test2.txt` 就是 `test.txt` 连接文件 (快捷方式)

1.12 tar: 压缩与解压缩命令

1.12.1 gzip 格式:

1、压包 (打包)

压缩用法:

`tar zcvf` 压缩包包名 文件 1 文件 2 ...

```
tar zcvf bk.tar.gz a.c b.txt xiutao test.txt
```

将 `a.c b.txt xiutao test.txt` 文件或文件夹压缩到 `bk.tar.gz` 中

2、解压

解压方法 1: 解压到当前目录

`tar zxvf` 压缩包名称

```
tar zxvf bk.tar.gz
```

解压方法 2: 解压到指定路径下

`tar zxvf` 压缩包名称 `-C` 目的路径

```
tar zxvf bk.tar.gz -C /home/teacher/share
```

将压缩包里的文件或文件夹 解压到 `/home/teacher/share` 目录下

注意: `gzip` 格式的压缩包后缀名 `.tar.gz` 结尾

1.12.2 bz2 格式:

1、 压包

压缩用法:

```
tar jcvf 压缩包包名 文件 1 文件 2 ...
```

```
tar jcvf bk.tar.bz2 a.c b.txt xiutao test.txt
```

将 a.c b.txt xiutao test.txt 文件或文件夹压缩到 bk.tar.gz 中

2、 解压

解压方法 1: 解压到当前目录

```
tar jxvf 压缩包的名称
```

```
tar jxvf bk.tar.bz2
```

解压方法 2: 解压到指定路径下

```
tar jxvf 压缩包的名称 -C 目的路径
```

```
tar jxvf bk.tar.bz2 -C /home/teacher/share
```

将压缩包里的文件或文件夹 解压到/home/teacher/share 目录下

注意: bz2 格式的压缩包后缀名.tar.bz2 结尾

1.13 编辑器之神 ---vi

1.13.1 安装 vim (vi 升级版)

1、 安装 vim

```
sudo apt-get install vim
```

2、 安装 ctags

```
sudo apt-get install ctags
```

3、 配置 vim

step1: 将 vim_configure 拷入当前用户的目录下

使用 samba 或共享文件夹完成

step2: 打开终端, 执行以下以下命令:

```
cd vim_configure
```

```
./copy_con.sh
```

1.13.2 vi 的使用

使用 vi 打开文件

```
vi filename:
```

打开或新建文件, 并将光标置于第一行行首

```
vi test.txt
```

```
vi +n filename:
```

打开存在文件, 并将光标置于第 n 行行首

vi +8 test.txt

1.13.3 vi 的三种模式

1、插入模式

这种模式可以直接编辑文档.....

注意：vi 创建一个不存在文件时（.c 文件可以，txt 不可以），默认进入插入模式

2、编辑模式

在编辑模式下可以敲一些命令，执行例如 复制 n 行、剪切 n 行、粘贴等功能

注意：vi 在打开一个已经存在的文件的时候，默认进入的是编辑模式

3、命令模式（最后一行模式）

在此模式下可以保存文件，退出 vi 等功能

4、三种模式之间的切换：

1) 由编辑模式进插入模式 按 a i o 这几个字母

区别是插入的位置不一样

2) 由插入模式进编辑模式

按 ESC 键

3) 只能由编辑模式才能进最后一行模式

在编辑模式下按 shift : 进入 最后一行模式

在最后一行模式下的命令

:w 保存文件

:wq 保存文件并退出 vi

:x 保存文件并退出 vi

:q! 不保存文件，强制退出 vi

:w filename 另存为 filename

4) 由命令模式（最后一行模式）也可以进编辑模式

按 ESC

总结：

a) 由 插入模式或者命令模式（最后一行模式） 进编辑模式 方法是按 ESC 键

b) 由编辑模式 进 命令模式（最后一行模式） 按 shift :

c) 由编辑模式进 插入模式 按 a i o

d) 插入模式和命令模式（最后一行模式）不能直接切换

5、编辑模式下的操作：

a) u 撤消前面多次修改。ctl r 反撤销

b) [n]x 删除光标后 n 个字符。

3x

c) [n]X 删除光标前 n 个字符。

6X

d) [n]dd 删除从当前行开始的 n 行。

4dd

e) [n]yy 复制从当前行开始的 n 行。

4yy

- f) p 把粘贴板上的内容插入到当前行。
- g) . 执行上一次操作
- h) shift+zz(按住 shift 按两下 z 键) 保存退出当前文件

6、编辑模式下移动光标

[n]G: 将光标定位到第 n 行开始处

5G 将光标移动到第 5 行开始处

G: 将光标定位到文件结束处

gg:将光标定位到文件开始处

编辑模式下的查找

/字符串: 从光标开始处向文件尾查找字符串。

n: 同一方向重复上一次查找命令。

N: 反方向重复上一次查找命令

:nohl 取消高亮

:set hls 设置高亮 vi 配置文件的作用:

7、vi_configure

- 1、自动添加创建时间、名称等注释信息
- 2、自动添加行号
- 3、支持鼠标点击定位
- 4、支持函数列表功能(F9 打开或关闭)
- 5、支持多文件打开功能 (F5)
- 6、可使用 ctrl+N 补全函数以及结构体成员
- 7、自动整理代码格式:

使用鼠标选中代码, 按下“=”号键可自动规整代码

- 8、支持 vim 快捷键.pdf 中的快捷键。

1.14 编译器 gcc

在 windows 操作系统下, 编译工具用集成开发环境 vc6.0

在 Linux 操作系统下没有很好的集成环境让我们用, 用的编译器是 gcc

程序的编译分为四个阶段: 由.c 到可执行程序

- 1、预编译
- 2、编译
- 3、汇编
- 4、链接

编译程序:

- 1、一步到位

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

运行程序:./hello

2、分步骤完成

例：分步进行

```
gcc -E hello.c -o hello.i    1、预处理
gcc -S hello.i -o hello.s    2、编译
gcc -c hello.s -o hello.o    3、汇编
gcc hello.o -o hello_elf    4、链接
```

第2章 c 数据类型及语句

a. c 语言特点

我的第一个 c 语言程序

```
#include<stdio.h>
int main()//这个我的第一个 c 语言程序
{
    printf("hello world\n"); //printf 是输出打印的函数
    return 0;
}
```

1.#include<stdio.h> 头文件包含，一定要有

2.每一个 c 语言的程序有且只有一个 main 函数，这是整个程序的开始位置

3.C 语言中()、[]、{}、“”、”、都必须成对出现,必须是英文符号

4.C 语言中语句要以分号结束。

5.//为注释

```
/*
```

有志者，事竟成，破釜沉舟，百二秦关终属楚；

苦心人，天不负，卧薪尝胆，三千越甲可吞吴

```
*/
```

2.1 关键字

2.1.1 数据类型相关的关键字

用于定义变量或者类型

类型 变量名；

char 、 short、 int 、 long 、 float、 double、

struct、 union、 enum 、 signed、 unsigned、 void

1、 char 字符型，用 char 定义的变量是字符型变量，占 1 个字节

```
char ch='a'; =为赋值号
```

```
char ch1= '1' ; 正确
char ch2 = '1234' ; 错误的
```

2、short 短整型 ,使用 short 定义的变量是短整型变量, 占 2 个字节

```
short int a=11; -32768 - ---32767
```

3、int 整型 , 用 int 定义的变量是整型变量, 在 32 位系统下占 4 个字节, 在 16 平台下占 2 个字节

```
int a=44; -20 亿---20 亿
```

4、long 长整型 用 long 定义的变量是长整型的, 在 32 位系统下占 4 个字节

```
long int a=66;
```

5、float 单浮点型 (实数), 用 float 定义的变量是单浮点型的实数, 占 4 个字节

```
float b=3.8f;
```

6、double 双浮点型 (实数), 用 double 定义的变量是双浮点型的实数, 占 8 个字节

```
double b=3.8;
```

7、struct 这个关键字是与结构体类型相关的关键字, 可以用它来定义结构体类型, 以后讲结构体的时候再讲

8、union 这个关键字是与共用体 (联合体) 相关的关键字, 以后再讲

9、enum 与枚举类型相关的关键字 以后再讲

10、signed 有符号(正负)的意思

在定义 char 、整型 (short 、int、long) 数据的时候用 signed 修饰, 代表咱们定义的数据是有符号的, 可以保存正数, 也可以保存负数

例 : signed int a=10;

```
signed int b=-6;
```

注意: 默认情况下 signed 可以省略 即 int a=-10;//默认 a 就是有符号类型的数据

11、unsigned 无符号的意思

在定义 char 、整型 (short 、int、long) 数据的时候用 unsigned 修饰, 代表咱们定义的数据是无符号类型的数据

只能保存正数和 0。

```
unsigned int a=101;
```

```
unsigned int a=-101; //错误
```

扩展: 内存存储

```
char ch='a'; //占 1 个字节, 存储的是 97
```

```
0110 0001
```

字节: 内存的基本单位, 8 位为 1 个字节

计算机存储时, 只能存储 1 和 0 的二进制组合, 1 和 0 都分别占 1 位

字符型数据在内存中存储的不是字符本身, 而是存储其 Ascii 码

整型变量存储的是其值的二进制

```
unsigned int a = 97;
```

扩展：正数和负数在内存中到底是怎么存的

原码、反码、补码

规定：

正数的原码反码和补码相同 5

0000 0101

负数：-5

0000 0101

最高位为符号位 最高位为 1 代表是个负数

原码：-5

1000 0101

反码：除了符号位 其他位取反

1111 1010

补码：反码 +1

1111 1011

注意：负数在内存中是以补码形式存放的

例 1：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a=-5;
    printf("%d\n",a);
    printf("%x\n",a);
    return 0;
}
```

12、void 空类型的关键字

char、int、float 都可以定义变量

void 不能定义变量，没有 void 类型的变量

void 是用来修饰函数的参数或者返回值，代表函数没有参数或没有返回值

例：

```
void fun(void)
{
```

}

代表 fun 函数没有返回值，fun 函数没有参数

例 2 :

```
#include <stdio.h>
int main()
{
    char a = 'a';
    short int b = 10;
    int c;
    long int d;
    float e;
    double f;
    printf("%d\n",sizeof(a));
    printf("%d\n",sizeof(b));
    printf("%d\n",sizeof(c));
    printf("%d\n",sizeof(d));
    printf("%d\n",sizeof(e));
    printf("%d\n",sizeof(f));
    return 0;
}
```

2.1.2 存储相关关键字

register、static、const、auto、extern

1、**register** 是寄存器的意思，用 register 修饰的变量是寄存器变量，

即：在编译的时候告诉编译器这个变量是寄存器变量，**尽量**将其存储空间分配在寄存器中。

注意：

- (1):定义的变量不一定真的存放在寄存器中。
- (2): cpu 取数据的时候去寄存器中拿数据比去内存中拿数据要快
- (3): 因为寄存器比较宝贵，所以不能定义寄存器数组
- (4): **register** 只能修饰 字符型及整型的，不能修饰浮点型

```
register char ch;
register short int b;
register int c;
register float d;//错误的
```

- (5): 因为 **register** 修饰的变量可能存放在寄存器中不存放在内存中，所以不能对寄存器变量取地址。因为只有存放在内存中的数据才有地址

```
register int a;
int *p;
p=&a;//错误的，a 可能没有地址
```

2、static 是静态的意思

static 可以修饰全局变量、局部变量、函数
这个以后的课程中重点讲解

3、const

const 是常量的意思
用 const 修饰的变量是只读的，不能修改它的值

```
const int a=101;//在定义 a 的时候用 const 修饰，并赋初值为 101  
从此以后，就不能再给 a 赋值了  
a=111;//错误的
```

const 可以修饰指针，这个在以后课程中重点讲解

4、auto int a;和 int a 是等价的，auto 关键字现在基本不用

5、extern 是外部的意思，一般用于函数和全局变量的声明，这个在后面的课程中，会用到

2.1.3 控制语句相关的关键字

if、else、break、continue、for、while、do、switch case
goto、default

2.1.4 其他关键字

sizeof、typedef、volatile

1、sizeof

使用来测变量、数组的占用存储空间的大小（字节数）

```
例 3：  
int a=10;  
int num;  
num=sizeof(a);
```

2、typedef 重命名相关的关键字

```
unsigned short int a = 10;
```

U16

关键字，作用是给一个已有的类型，重新起个类型名，并没有创造一个新的类型
以前大家看程序的时候见过类似的变量定义方法

```
INT16 a;
```

```
U8 ch;
```

```
INT32 b;
```

大家知道，在 c 语言中没有 INT16 U8 这些关键字

INT16 U8 是用 typedef 定义出来的新的类型名，其实就是 short int 及 unsigned char 的别名

typedef 起别名的方法：

1、用想起名的类型定义一个变量

```
short int a;
```

2、用新的类型名替代变量名

```
short int INT16;
```

3、在最前面加 `typedef`

```
typedef short int INT16;
```

4: 就可以用新的类型名定义变量了

```
INT16 b;和 short int b;//是一个效果
```

例 4 :

```
#include <stdio.h>
//short int b;
//short int INT16;
typedef short int INT16;
int main(int argc, char *argv[])
{
    short int a=101;
    INT16 c=111;
    printf("a=%d\n",a);
    printf("c=%d\n",c);
    return 0;
}
```

3、volatile 易改变的意思

用 `volatile` 定义的变量，是易改变的，即告诉 `cpu` 每次用 `volatile` 变量的时候，重新去内存中取保证用的是最新的值,而不是寄存器中的备份。

`volatile` 关键字现在较少适用

```
volatile int a=10;
```

扩展知识:

命名规则:

在 `c` 语言中给变量和函数起名的时候，由字母、数字、下划线构成
必须以字母或者下划线开头

例 5 :

```
int a2;//正确的
int a_2;//正确的
int _b;//正确的
int 2b;// 错误的
int $a2;//错误的
```

注意：起名的时候要求见名知意

Linux 风格

```
stu_num
```

驼峰风格

```
StuNum
```

大小写敏感

```
int Num;
```

```
int num;
```

C 语言的程序结构

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` 函数是程序的入口，即 程序从 `main` 函数开始执行

2.2 数据类型

2.2.1 基本类型

`char`、`short int`、`int`、`long int`、`float`、`double`

2.2.2 构造类型

概念：由若干个相同或不同类型数据构成的集合，这种数据类型被称为构造类型

例：`int a[10];`

数组、结构体、共用体、枚举

扩展：常量和变量

常量：在程序运行过程中，其值不可以改变的量

例：`100`、`'a'`、`"hello"`

- 整型 `100`, `125`, `-100`, `0`
- 实型 `3.14`, `0.125f`, `-3.789`
- 字符型 `'a'`, `'b'`, `'2'`
- 字符串 `"a"`, `"ab"`, `"1232"`

ASCII 码表

例 6：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    unsigned int i;
    for(i=0;i<=255;i++)
    {
        printf("%d %c ",i,i);
        if(i%10==0)
```

```

        printf("\n");
    }
    return 0;
}

```

变量：其值可以改变的量被称为变量

```

int a=100;
a=101;

```

整型数据

➤ 整型常量：(按进制分)：

十进制：以正常数字 1-9 开头，如 457 789

八进制：以数字 0 开头，如 0123

十六进制：以 0x 开头，如 0x1e

a=10, b=11,c=12, d=13,e=14,f=15

➤ 整型变量：

➤ 有/无符号短整型(un/signed) short(int) 2 个字节

➤ 有/无符号基本整型(un/signed) int 4 个字节

➤ 有/无符号长整型(un/signed) long (int) 4 个字节 (32 位处理器)

实型数据(浮点型)

➤ 实型常量

➤ 实型常量也称为实数或者浮点数

十进制形式：由数字和小数点组成:0.0、0.12、5.0

指数形式：123e3 代表 123*10 的三次方

123e-3

➤ 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

➤ 实型变量

单精度(float)和双精度(double)3.1415926753456

float 型：占 4 字节，7 位有效数字,指数-37 到 38

3333.333 33

double 型：占 8 字节，16 位有效数字,指数-307 到 308

字符数据

➤ 字符常量：

直接常量：用单引号括起来，如：'a'、'b'、'0'等。

转义字符：以反斜杠“\”开头，后跟一个或几个字符、如'\n','\t'等，分别代表换行、横向跳格。

'\\'表示的是\ '%%' '\'

➤ 字符变量：

用 char 定义，每个字符变量被分配一个字节的内存空间

字符值以 ASCII 码的形式存放在变量的内存单元中；

注：char a;

```
a = 'x';
```

a 变量中存放的是字符'x'的 ASCII :120

即 a=120 跟 a='x'在本质上是一致的.

例 7 :

```
#include <stdio.h>    \p<
int main(int argc, char *argv[])    \im
{
    char a='x';
    char b=120;
    printf("a=%c\n",a);        \ip
    printf("b=%c\n",b);
    return 0;
}
```

字符串常量

是由双引号括起来的字符序列，如“CHINA”、“哈哈”

“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

‘a’为字符常量，”a”为字符串常量

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’

格式化输出字符：

%d 十进制有符号整数 %u 十进制无符号整数
%x, 以十六进制表示的整数 %o 以八进制表示的整数
%f float 型浮点数 %lf double 型浮点数
%e 指数形式的浮点数
%s 字符串 %c 单个字符
%p 指针的值

特殊应用：

%3d %03d %-3d %5.2f

%3d:要求宽度为 3 位，如果不足 3 位，前面空格补齐;如果足够 3 位，此语句无效

%03d: 要求宽度为 3 位，如果不足 3 位，前面 0 补齐;如果足够 3 位，此语句无效

%-3d: 要求宽度为 3 位，如果不足 3 位，后面空格补齐;如果足够 3 位，此语句无效

%.2f: 小数点后只保留 2 位

2.2.3 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：

⑩ 自动转换：

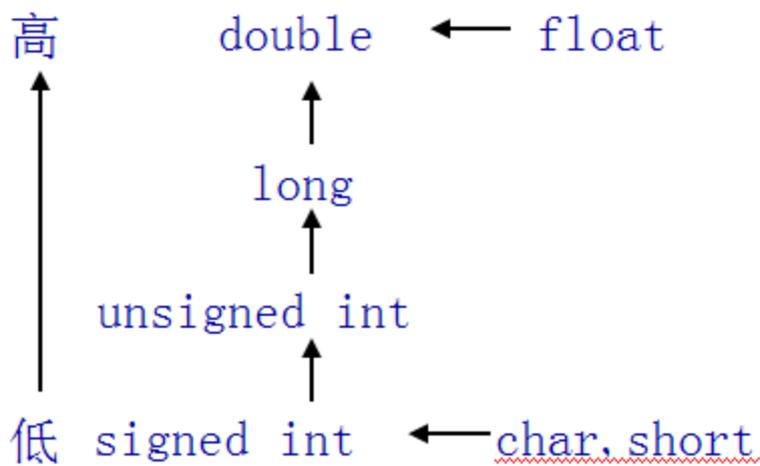
遵循一定的规则,由编译系统自动完成.

⑩ 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

➤ 自动转换的原则:

- 1、 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.
- 2、 转换方向:



- 1) 当表达式中出现了 char、short int、int 类型中的一种或者多种, 没有其他类型了 参加运算的成员全部变成 int 类型的参加运算, 结果也是 int 类型的 ‘

例 8 :

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("%d\n",5/2);
    return 0;
}
```

- 2) 当表达式中出现了带小数的实数, 参加运算的成员全部变成 double 类型的参加运算, 结果也是 double 型。

例 9 :

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("%lf\n",5.0/2);
    return 0;
}
```

- 3) 当表达式中有有符号数 也有无符号数, 参加运算的成员变成无符号数参加运算结果也是无符号数.(表达式中无实数)

例 10:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a=-8;
    unsigned int b=7;
    if(a+b>0)
    {
        printf("a+b>0\n");
    }
    else
    {
        printf("a+b<=0\n");
    }
    printf("%x\n",(a+b));
    printf("%d\n",(a+b));
    return 0;
}
```

- 4) 在赋值语句中等号右边的类型自动转换为等号左边的类型

例 11:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a;
    float b=5.8f;//5.8 后面加 f 代表 5.8 是 float 类型，不加的话，认为是 double 类型
    a=b;
    printf("a=%d\n",a);
    return 0;
}
```

- 5) 注意自动类型转换都是在运算的过程中进行临时性的转换，并不会影响自动类型转换的变量的值和其类型

例 12 :

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a;
    float b=5.8f;//5.8 后面加 f 代表 5.8 是 float 类型，不加的话，认为是 double 类型
    a=b;
    printf("a=%d\n",a);
    printf("b=%f\n",b);//b 的类型依然是 float 类型的，它的值依然是 5.8
}
```

```
return 0;  
}
```

强制转换:通过类型转换运算来实现

(类型说明符)(表达式)

功能:

把表达式的运算结果强制转换成类型说明符所表示的类型

例如:

(float)a; // 把 a 的值转换为实型

(int)(x+y); // 把 x+y 的结果值转换为整型

注意:

类型说明符必须加括号

例 13 :

```
#include <stdio.h>  
int main(int argc, char *argv[])  
{  
    float x=0;  
    int i=0;  
    x=3.6f;  
    i = x;  
    i = (int)x;  
    printf("x=%f,i=%d\n",x,i);  
    return 0;  
}
```

说明:

无论是强制转换或是自动转换,都只是为了本次运算的需要,而对变量的数据长度进行的临时性转换,而不改变数据定义的类型以及它的值

2.2.4 指针

2.3 运算符

2.3.1 运算符

用算术运算符将运算对象(也称操作数)连接起来的、符合 C 语法规则的式子,称为 C 算术表达式
运算对象包括常量、变量、函数等

例如: $a * b / c - 1.5 + 'a'$

2.3.2 运算符的分类:

1、双目运算符:即参加运算的操作数有两个

例: +

a+b

2、单目运算符：参加运算的操作数只有一个

++自增运算符 给变量值+1

--自减运算符

```
int a=10;
```

```
a++;
```

3、三目运算符:即参加运算的操作数有 3 个

()?():()

2.3.3 算数运算符

+ - * / % += -= *= /= %=

10%3 表达式的结果为 1

复合运算符：

a += 3 相当于 a=a+3

a*=6+8 相当于 a=a*(6+8)

2.3.4 关系运算符

(>、<、==、>=、<=、!=)

!=为不等于

一般用于判断条件是否满足或者循环语句

2.3.5 逻辑运算符

1、&& 逻辑与

两个条件都为真，则结果为真

```
if((a>b) && (a<c))
```

if(b<a<c)//这种表达方式是错误的

2、|| 逻辑或

两个条件至少有一个为真，则结果为真

```
if((a>b) || (a<c))
```

3、! 逻辑非

```
if(!(a>b))
```

```
{
```

```
}
```

2.3.6 位运算符

1、&按位 与

任何值与 0 得 0，与 1 保持不变

使某位清 0

```
0101 1011&
```

```
1011 0100
```

```
-----
```

```
0001 0000
```

2、| 按位或

任何值或 1 得 1，或 0 保持不变

0101 0011 |

1011 0100

1111 0111

3、~ 按位取反

1 变 0，0 变 1

0101 1101 ~

1010 0010

4、^ 按位异或

相异得 1，相同得 0

1001 1100 ^

0101 1010

1100 0110

5、位移

>> 右移

<< 左移

注意右移分：逻辑右移、算数右移

(1)、右移

逻辑右移 高位补 0，低位溢出

算数右移 高位补符号位，低位溢出 (有符号数)

-15

1000 1111

1111 0000

1111 11 00 -4

A)、逻辑右移

低位溢出、高位补 0

0101 1010 >>3

0000 1011

B)、算数右移：

对有符号数来说

低位溢出、高位补符号位。

1010 1101 >> 3

1111 010 1

0101 0011 >>3

0000 101 0

总结 右移：

1、逻辑右移 高位补 0，低位溢出

注：无论是有符号数还是无符号数都是高位补 0，低位溢出

2、算数右移 高位补符号位，低位溢出 (有符号数)

注：对无符号数来说，高位补 0，低位溢出

对有符号数来说，高位补符号位，低位溢出

在一个编译系统中到底是逻辑右移动，还是算数右移，取决于编译器

判断右移是逻辑右移还是算数右移

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("%d\n", -1 >> 3);
    return 0;
}
```

如果结果还是-1 证明是算数右移

(2)、左移<< 高位溢出，低位补0

5<<1

2.3.7 条件运算符

():0

A?B:C;

如果? 前边的表达式成立，整个表达式的值，是? 和: 之间的表达式的结果

否则是: 之后的表达式的结果

例 14:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a;
    a=(3<5)?(8):(9);
    printf("a=%d\n",a);
    return 0;
}
```

2.3.8 逗号运算符，

(),0

例 15:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int num;
    num=(5,6);
    printf("%d\n",num);
    return 0;
}
```

注意逗号运算符的结果是,后边表达式的结果

注意:逗号运算符的优先级最低

运算符总结:

运算符不仅有优先级还有结合型

2+3+9+10 左结合性

a=b=c=d=100 右结合性

优先级高的先于优先级低的先运算。

3+4*7

优先级一样的按结合性来运算

左结合性 从左往右运算

右结合性 从右往左运算

优先级别	运算符	运算形式	结合方向	名称或含义
1	()	(e)	自左至右	圆括号
	[]	a[e]		数组下标
	.	x.y		成员运算符
	->	p->x		用指针访问成员的指向运算符
2	- +	-e	自右至左	负号和正号
	++ --	++x 或 x++		自增运算和自减运算
	!	!e		逻辑非
	~	~e		按位取反
	(t)	(t)e		类型转换
	*	*p		指针运算, 由地址求内容
	&	&x		求变量的地址
sizeof	sizeof(t)	求某类型变量的长度		

3	* / %	e1 * e2	自左至右	乘、除和求余
4	+ -	e1 + e2	自左至右	加和减
5	<< >>	e1 << e2	自左至右	左移和右移
6	< <= > >=	e1 < e2	自左至右	关系运算(比较)
7	== !=	e1 == e2	自左至右	等于和不等比较
8	&	e1 & e2	自左至右	按位与
9	^	e1 ^ e2	自左至右	按位异或
10		e1 e2	自左至右	按位或
11	&&	e1 && e2	自左至右	逻辑与(并且)
12		e1 e2	自左至右	逻辑或(或者)
13	?:	e1 ? e2 : e3	自右至左	条件运算
14	=	x = e	自右至左	赋值运算
	+= -= *=	x += e		复合赋值运算
	/= %= >>=			
	<<= &= ^=			
15	,	e1, e2	自左至右	顺序求值运算

自增自减运算符

i++ i--

运算符在变量的后面, 在当前表达式中先用 i 的值, 下条语句的时候 i 的值改变

例 16:

```
#include <stdio.h>
int main()
```

```
{
    int i=3;
    int num;
    num=i++;
    printf("num=%d,i=%d\n",num,i);//num=3 ,i=4
    return 0;
}
```

++i 先加，后用

例 17:

```
#include <stdio.h>
int main()
{
    int i=3;
    int num;
    num=++i;
    printf("num=%d,i=%d\n",num,i);//num=4,i=4
    return 0;
}
```

例 18 :

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i=3;
    int num;
    num = (i++)+(i++)+(i++);
    printf("num=%d\n",num);

    return 0;
}
```

例 19 :

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i=3;
    int num;
```

```
num = (++i)+(++i)+(++i);  
printf("num=%d\n",num);  
return 0;  
}
```

2.4 控制语句相关关键字讲解

2.4.1 选择控制语句相关的关键字

1、 if 语句

形式:

1) if(条件表达式)

{//复合语句, 若干条语句的集合

语句 1;

语句 2;

}

如果条件成立执行大括号里的所有语句, 不成立的话大括号里的语句不执行

例 20 :

```
#include<stdio.h>  
int main()  
{  
    int a=10;  
    if(a>5)  
    {  
        printf("a>5\n");  
    }  
    return 0;  
}
```

2) if(条件表达式)

{

}

else

{

}

if else 语句的作用是, 如果 if 的条件成立, 执行 if 后面{}内的语句, 否则执行 else 后的语句

例 21 :

```
#include<stdio.h>  
int main()
```

```
{
    int a=10;
    if(a>5)
    {
        printf("a>5\n");
    }
    else
    {
        printf("a<=5\n");
    }
    return 0;
}
```

注意 if 和 else 之间只能有一条语句，或者有一个复合语句，否则编译会出错

例 22：

```
if()
    语句 1;
    语句 2;
else
    语句 3;
    语句 4;
```

错误：if 和 else 之间只能有一条语句,如果有多条语句的话加大括号

例 23：

```
if()
{
    语句 1;
    语句 2;
}
else
{
    语句 3;
    语句 4;
}
```

正确

3) if(条件表达式)

```
{
}
else if(条件表达式)
{
```

```
}  
else if(条件表达式)  
{  
}  
else  
{  
}
```

在判断的时候，从上往下判断，一旦有成立的表达式，执行对应的复合语句，下边的就不再判断了，各个条件判断是互斥的

例 24 :

```
#include <stdio.h>  
int main(void)  
{  
    char ch;  
    float score = 0;  
    printf("请输入学生分数:\n");  
    scanf("%f",&score);  
    if(score<0 || score >100)  
    {  
        printf("你所输入的信息有错\n");  
        return 0;  
    }  
    else if( score<60)  
    {  
        ch = 'E';  
    }  
    else if ( score < 70 )  
    {  
        ch = 'D';  
    }  
    else if ( score < 80 )  
    {  
        ch = 'C';  
    }  
    else if ( score < 90 )  
    {  
        ch = 'B';  
    }  
    else
```

```
{
    ch = 'A';
}

printf("成绩评定为 : %c\n",ch);
return 0;
}
```

2、 switch 语句

switch (表达式) //表达式只能是字符型或整型的(short int int long int)

```
{
    case 常量表达式1:
        语句1;
        break;
    case 常量表达式2:
        语句2;
        break;
    default: 语句3; break;
}
```

注意: break 的使用

例 25:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int n;
    printf("请输入一个 1~7 的数\n");
    scanf("%d",&n);
    switch(n)
    {
        case 1:
            printf("星期一\n");
            break;
        case 2:
            printf("星期二\n");
            break;
        case 3:
            printf("星期三\n");
            break;
        case 4:
```

```
        printf("星期四\n");
        break;
    case 5:
        printf("星期五\n");
        break;
    case 6:
        printf("星期六\n");
        break;
    case 7:
        printf("星期天\n");
        break;
    default:
        printf("您的输入有误，请输入 1~7 的数\n");
        break;
}
return 0;
}
```

2.4.2 循环控制语句相关的关键字

1、 for 循环

```
for(表达式 1;表达式 2;表达式 3)
{//复合语句，循环体
```

```
}
```

第一次进入循环的时候执行表达式 1，表达式 1 只干一次，

表达式 2，是循环的条件，只有表达式 2 为真了，才执行循环体，也就是说每次进入循环体之前要判断表达式 2 是否为真。

每次执行完循环体后，首先执行表达式 3

例 25 : for 循环求 0~100 的和

```
#include <stdio.h>
int main(void)
{
    int i;
    int sum=0;
    for(i=1;i<=100;i++)
    {
        sum = sum+i;
```

```
    }  
    printf("sum=%d\n",sum);  
    return 0;  
}
```

例 26 :

```
#include <stdio.h>  
int main(int argc, char *argv[])  
{  
    int i,j;  
    for(i=1;i<=9;i++)  
    {  
        for(j=1;j<=i;j++)  
        {  
            printf("%d*%d=%d  ",i,j,i*j);  
        }  
        printf("\n");  
    }  
    return 0;  
}
```

2、 while 循环

1) 形式 1:

```
while(条件表达式)  
{//循环体, 复合语句  
  
}
```

进入 **while** 循环的时候, 首先会判断条件表达式是否为真, 为真进入循环体, 否则退出循环

例 27 :

```
#include <stdio.h>  
int main(void)  
{  
    int i=1;  
    int sum=0;  
    while(i<=100)  
    {  
        sum = sum+i;
```

```
        i++;  
    }  
    printf("sum=%d\n",sum);  
    return 0;  
}
```

2) 形式 2 : do

```
do{//循环体
```

```
}while(条件表达式);
```

先执行循环体里的代码，然后去判断条件表达式是否为真，为真再次执行循环体，否则退出循环

例 28 :

```
#include <stdio.h>  
int main(void)  
{  
    int i=1;  
    int sum=0;  
    do  
    {  
        sum = sum+i;  
        i++;  
    }while(i<=100);  
    printf("sum=%d\n",sum);  
    return 0;  
}
```

形式 1 和形式 2 的区别是，形式 1 先判断在执行循环体，形式 2 先执行循环体，再判断
break 跳出循环

continue 结束本次循环，进入下一次循环

例 29 :

```
#include <stdio.h>  
int main(void)  
{  
    int i;  
    int sum=0;  
    for(i=1;i<=100;i++)  
    {  
        if(i==10)
```

```
break;//将 break 修改成 continue 看效果
sum = sum+i;
}
printf("sum=%d\n",sum);
return 0;
}
```

return 返回函数的意思。结束 **return** 所在的函数，
在普通函数中，返回到被调用处，在 **main** 函数中的话，结束程序

3、 goto

例 30 :

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("test00000000000000000000\n");
    printf("test1111111111111111\n");
    goto xiutao;
    printf("test2222222222222222\n");
    printf("test3333333333333333\n");
    printf("test44444444444444444444\n");
    printf("test5555555555555555\n");
xiutao:
    printf("test6666666666666666\n");
    return 0;
}
```

第3章 数组

3.1 数组的概念

数组是若干个相同类型的变量在内存中有序存储的集合。

`int a[10];` //定义了一个整型的数组 **a**，**a** 是数组的名字，数组中有 10 个元素，每个元素的类型都是 `int` 类型，而且在内存中连续存储。

这十个元素分别是 `a[0]` `a[1]` `a[9]`

`a[0]~a[9]`在内存中连续的顺序存储

3.2 数组的分类

3.2.1 按元素的类型分类

1) 字符数组

即若干个字符变量的集合，数组中的每个元素都是字符型的变量

```
char s[10]; s[0],s[1]...s[9];
```

2) 短整型的数组

```
short int a[10]; a[0],a[9]; a[0]=4;a[9]=8;
```

3) 整型的数组

```
int a[10]; a[0] a[9]; a[0]=3;a[9]=6;
```

4) 长整型的数组

```
long int a[5];
```

5) 浮点型的数组（单、双）

```
float a[6]; a[4]=3.14f;
```

```
double a[8]; a[7]=3.115926;
```

6) 指针数组

```
char *a[10]
```

```
int *a[10];
```

7) 结构体数组

```
struct stu boy[10];
```

3.2.2 按维数分类

一维数组

```
int a[30];
```

类似于一排平房

二维数组

```
int a[2][30];
```

可以看成一栋楼房 有多层，每层有多个房间，也类似于数学中的矩阵

二维数组可以看成由多个一维数组构成的。

有行，有列，

多维数组

```
int a[4][2][10];
```

三维数组是由多个相同的二维数组构成的

```
int a[5][4][2][10];
```

3.3 数组的定义

定义一个数组，在内存里分配空间

3.3.1 一维数组的定义

格式：

数据类型 数组名 [数组元素个数];
int a [10];
char b [5];定义了 5 个 char 类型变量的数组 b
5 个变量分别为 b[0],b[1],b[2], b[3],b[4];

在数组定义的时候可以不给数组元素的个数，根据初始化的个数来定数组的大小

例 1:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a[]={1,2,3,4,5};
    printf("%d\n",sizeof(a));
    return 0;
}
```

3.3.2 二维数组的定义

格式:

数据类型 数组名 [行的个数][列的个数];
int a [4][5];
定义了 20 个 int 类型的变量 分别是
a[0][0],a[0][1],a[0][2],a[0][3],a[0][4];
a[1][0],a[1][1],a[1][2],a[1][3],a[1][4];
a[2][0],a[2][1],a[2][2],a[2][3],a[2][4];
a[3][0],a[3][1],a[3][2],a[3][3],a[3][4];

多维数组定义:

```
int a[3][4][5]
int a[8][3][4][5];
```

扩展:

二维数组在定义的时候，可以不给出行数，但必须给出列数，二维数组的大小根据初始化的行数来定

例 2:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a[][3]={
        {1,2,3},
        {4,5,6},
        {7,8,9},
        {10,11,12}
    };
}
```

```
printf("%d\n",sizeof(a));  
return 0;  
}
```

3.4 定义并初始化

开辟空间的同时并且给变量赋值

3.4.1 一维数组的初始化

a、全部初始化

```
int a[5]={2,4,7,8,5};
```

代表的意思: a[0]=2; a[1]=4;a[2]=7;a[3]=8;a[4]=5;

b、部分初始化

```
int a[5]={2,4,3};初始化赋值不够后面补 0
```

```
a[0]=2; a[1]=4;a[2]=3;a[3]=0;a[4]=0;
```

注意: 只能省略后面元素, 可以不初始化, 不能中间的不初始化

例 3 :

```
#include <stdio.h>  
int main(int argc, char *argv[])  
{  
    int a[5]={2,3,5};  
    int i;  
    for(i=0;i<5;i++)  
    {  
        printf("a[%d]=%d\n",i,a[i]);  
    }  
    return 0;  
}
```

3.4.2 二维数组的定义并初始化

按行初始化:

a、全部初始化

```
int a[2][2]={{1,2},{4,5}};
```

```
a[0][0]=1; a[0][1]=2; a[1][0]=4; a[1][1]=5;
```

b、部分初始化

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

```
a[0][0]=1; a[0][2]=0;
```

逐个初始化:

全部初始化:

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

部分初始化:

```
int a[2][3]={3,5,6,8};
```

3.5 数组元素的引用方法

一维数组元素的引用方法

数组名 [下标]; //下标代表数组元素在数组中的位置

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

```
a[2];
```

二维数组元素的引用方法

数组名[行下标][列下标];

```
int a[3][4];
```

```
a[1][2]
```

例 4 :

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a[3][4]={{1,2,3,4},{5,6},{5}};
    int b[3][4]={11,12,13,14,15,16,17,18,19};
    int ij;
    for(i=0;i<3;i++)//遍历所有行
    {
        for(j=0;j<4;j++)//遍历一行的所有列
        {
            printf("a[%d][%d]=%d    ",i,j,a[i][j]);
        }
        printf("\n");
    }

    for(i=0;i<3;i++)//遍历所有行
    {
        for(j=0;j<4;j++)//遍历一行的所有列
        {
            printf("b[%d][%d]=%d    ",i,j,b[i][j]);
        }
        printf("\n");
    }
}
```

```
return 0;
}
```

3.5.1 字符数组的定义

```
char c1[]={ 'c' , ' ' , 'p' , 'r' , 'o' , 'g' };
char c2[] = "c prog" ;
char a[][5] = {
    { 'B' , 'A' , 'S' , 'I' , 'C' },
    { 'd' , 'B' , 'A' , 'S' , 'E' }
};
char a[][6] = { "hello" , "world" };
```

➤ 字符数组的引用

- 1.用字符串方式赋值比用字符逐个赋值要多占 1 个字节,用于存放字符串结束标志 '\0' ;
- 2.上面的数组 c2 在内存中的实际存放情况为:

'c'	' '	'p'	'r'	'o'	'g'	'\0'
-----	-----	-----	-----	-----	-----	------

注: '\0'是由 C 编译系统自动加上的

- 3.由于采用了'\0'标志, 字符数组的输入输出将变得简单方便.

例 5 :

```
int main( )
{
    char str[15];
    printf("input string:\n");
    scanf("%s",str);//hello
    printf("output:%s\n",str);
    return 0;
}
```

第4章 函数

4.1 函数的概念

函数是 c 语言的功能单位, 实现一个功能可以封装一个函数来实现。
定义函数的时候一切以功能为目的, 根据功能去定函数的参数和返回值。

4.2 函数的分类

1、从定义角度分类（即函数是谁实现的）

- 1.库函数（c 库实现的）
- 2.自定义函数（程序员自己实现的函数）
- 3.系统调用（操作系统实现的函数）

2、从参数角度分类

1.有参函数

函数有形参，可以是一个，或者多个，参数的类型随便完全取决于函数的功能

```
int fun(int a,float b,double c)
{
}
```

```
int max(int x,int y)
{
}
```

2.无参函数

函数没有参数,在形参列表的位置写个 void 或什么都不写

```
int fun(void)
{
}
```

```
int fun()
{
}
```

3、从返回值角度分类

(1).带返回值的函数

在定义函数的时候，必须带着返回值类型，在函数体里，必须有 return 如果没有返回值类型，默认返回整型。

例 1：

char fun()//定义了一个返回字符数据的函数

```
{
    char b='a';
    return b;
}
```

例 2：

```
fun()
{
```

```
        return 1;  
    }  
}
```

如果把函数的返回值类型省略了，默认返回整型

注：在定义函数的时候，函数的返回值类型，到底是什么类型的，取决于函数的功能。

(2).没返回值的函数

在定义函数的时候，函数名字前面加 void

void fun(形参表)

```
{  
    ;  
    ;  
    return ;  
    ;  
}
```

在函数里不需要 return

如果想结束函数，返回到被调用的地方， return ;什么都不返回就可以了

例 3:

```
#include <stdio.h>  
int max(int x,int y)  
{  
    int z;  
    if(x>y)  
        z=x;  
    else  
        z=y;  
    return z;  
}  
void help(void)  
{  
    printf("*****\n");  
    printf("*****帮助信息*****\n");  
    printf("*****\n");  
}  
int main(int argc, char *argv[])  
{  
    int num;  
    help();  
    num = max(10,10+5);  
}
```

```
printf("num=%d\n",num);  
return 0;  
}
```

4.3 函数的定义

什么叫做函数的定义呢？即函数的实现

1、函数的定义方法

返回值类型 函数名字(形参列表)

{//函数体，函数的功能在函数体里实现

}

例 4：

```
int max(int x, int y)  
{  
    int z;  
    if(x>y)  
        z=x;  
    else  
        z=y;  
    return z;  
}
```

注：形参必须带类型，而且以逗号分隔

函数的定义不能嵌套，即不能在一个函数体内定义另外一个函数，所有的函数的定义是平行的。

例 5：

```
void fun(void)  
{  
    ;  
    ;  
    ;  
}  
void fun2(void)  
{  
    ;  
}
```

这个程序是错误的，不能再 fun 的函数体中，定义 fun2 函数。

例 6：

```
void fun(void)
{
    ;
    ;
    ;
}
void fun2(void)
{
    ;
}
```

这个程序是正确的，fun 和 fun2 是平行结构

注：在一个程序中，函数只能定义一次

给函数起名字的时候，尽量见的名知意，符合 c 语言的命名规则

4.4 函数的声明

1、概念

对已经定义的函数，进行说明
函数的声明可以声明多次。

2、为什么要声明

有些情况下，如果不对函数进行声明，编译器在编译的时候，可能不认识这个函数，
因为编译器在编译 c 程序的时候，从上往下编译的。

3、声明的方法

什么时候需要声明

1) 主调函数和被调函数在同一个.c 文件中的时候

1] 被调函数在上，主调函数在下

```
例 7 :
void fun(void)
{
    printf("hello world\n");
}
int main()
{
    fun();
}
```

这种情况下不需要声明

2] 被调函数在下，主调函数在上

```
例 8 :  
int main()  
{  
    fun();  
}  
void fun(void)  
{  
    printf("hello world\n");  
}
```

编译器从上往下编译，在 `main` 函数（主调函数），不认识 `fun`，需要声明

怎么声明 呢？

1) 直接声明法

将被调用的函数的第一行拷贝过去，后面加分号

```
例 9 :  
void fun(void);  
int main()  
{  
    fun();  
}  
void fun(void)  
{  
    printf("hello world\n");  
}
```

2) 间接声明法

将函数的声明放在头文件中，.c 程序包含头文件即可

```
例 10 :  
a.c  
#include " a.h"  
int main()  
{  
    fun();  
}  
void fun(void)  
{  
    printf("hello world\n");  
}  
  
a.h  
extern void fun(void);
```

2) 主调函数和被调函数不在同一个.c 文件中的时候

一定要声明

声明的方法:

直接声明法

将被调用的函数的第一行拷贝过去, 后面加分号, 前面加 `extern`

间接声明法

将函数的声明放在头文件中, .c 程序包含头文件即可

4.5 函数的调用

函数的调用方法

变量= 函数名(实参列表);//带返回值的

函数名(实参列表);//不带返回值的

1、有无返回值

1).有返回值的, 根据返回值的类型, 需要在主调函数中定义一个对应类型的变量, 接返回值

例 11 :

```
int max(int x,int y)// x、y 形参, 是个变量
{

}

int main()
{
    int num;//需要定义一个 num 接收 max 函数的返回值
    num=max(4,8);//4 和 8 就是实参
}
```

2).没有返回值的函数, 不需要接收返回值。

例 12 :

```
void fun(void)
{
    printf("hello world\n");
}

int main()
{
    fun();
}
```

2、有无形参

函数名(实参列表);//带形参的

函数名();//没有形参的

注意：实参，可以常量，可以是变量，或者是表达式
形参是变量，是被调函数的局部变量。

4.6 函数总结

在定义函数的时候，关于函数的参数和返回值是什么情况，完全取决于函数的功能。

使用函数的好处？

- 1、定义一次，可以多次调用，减少代码的冗余度。
- 2、使咱们代码，模块化更好，方便调试程序，而且阅读方便

4.7 变量的存储类别

4.7.1 内存的分区：

- 1、内存：物理内存、虚拟内存

物理内存：实实在在存在的存储设备

虚拟内存：操作系统虚拟出来的内存。

操作系统会在物理内存和虚拟内存之间做映射。

在 32 位系统下，每个进程的寻址范围是 4G, 0x00 00 00 00 ~ 0xff ff ff ff

在写应用程序的，咱们看到的都是虚拟地址。

- 2、在运行程序的时候，操作系统会将 虚拟内存进行分区。

- 1).堆

在动态申请内存的时候，在堆里开辟内存。

- 2).栈

主要存放局部变量。

- 3).静态全局区

- 1: 未初始化的静态全局区

静态变量（定义变量的时候，前面加 `static` 修饰），或全局变量，没有初始化的，存在此区

- 2: 初始化的静态全局区

全局变量、静态变量，赋过初值的，存放在此区

4).代码区

存放咱们的程序代码

5).文字常量区

存放常量的。

4.7.2 普通的全局变量

概念:

在函数外部定义的变量

`int num=100;//num` 就是一个全局变量

```
int main()
```

```
{
```

```
    return 0;
```

```
}
```

作用范围:

全局变量的作用范围，是程序的所有地方。

只不过用之前需要声明。声明方法 `extern int num;`

注意声明的时候，不要赋值。

生命周期:

程序运行的整个过程，一直存在，直到程序结束。

注意：定义普通的全局变量的时候，如果不赋初值，它的值默认为 0

4.7.3 静态全局变量 `static`

概念:

定义全局变量的时候，前面用 `static` 修饰。

`static int num=100;//num` 就是一个静态全局变量

```
int main()
```

```
{
```

```
    return 0;
```

```
}
```

作用范围:

`static` 限定了静态全局变量的，作用范围

只能在它定义的.c（源文件）中有效

生命周期:

在程序的整个运行过程中，一直存在。

注意：定义静态全局变量的时候，如果不赋初值，它的值默认为 0

4.7.4 普通的局部变量

概念:

在函数内部定义的，或者复合语句中定义的变量

```
int main()
{
    int num;//局部变量

    {
        int a;//局部变量
    }
}
```

作用范围:

在函数中定义的变量，在函数中有效

在复合语句中定义的，在复合语句中有效。

生命周期:

在函数调用之前，局部变量不占用空间，调用函数的时候，才为局部变量开辟空间，函数结束了，局部变量就释放了。

在复合语句中定义的亦如此。

```
#include <stdio.h>
void fun()
{
    int num=3;
    num++;
    printf("num=%d\n",num);
}
int main()
{
    fun();
    fun();
    fun();
    return 0;
}
```

4.7.5 静态的局部变量

概念:

定义局部变量的时候，前面加 `static` 修饰

作用范围：

在它定义的函数或复合语句中有效。

生命周期：

第一次调用函数的时候，开辟空间赋值，函数结束后，不释放，以后再调用函数的时候，就不再为其开辟空间，也不赋初值，用的是以前的那个变量。

```
void fun()
{
    static int num=3;
    num++;
    printf("num=%d\n",num);
}
int main()
{
    fun();
    fun();
    fun();
}
```

注意：

1:

定义普通局部变量，如果不赋初值，它的值是随机的。

定义静态局部变量，如果不赋初值，它的值是 0

2: 普通全局变量，和静态全局变量如果不赋初值，它的值为 0

4.7.6 外部函数

咱们定义的普通函数，都是外部函数。

即函数可以在程序的任何一个文件中调用。

4.7.7 内部函数

在定义函数的时候，返回值前面加 `static` 修饰。这样的函数被称为内部函数。

`static` 限定了函数的作用范围，在定义的.c 中有效。

内部函数，和外部函数的区别：

外部函数，在所有地方都可以调用，

内部函数，只能在所定义的.c 中的函数调用。

扩展：

在同一作用范围内，不允许变量重名。

作用范围不同的可以重名。

局部范围内，重名的全局变量不起作用。（就近原则）

第5章 预处理、动态库、静态库

5.1 c 语言编译过程

```
gcc -E hello.c -o hello.i    1、预处理
gcc -S hello.i -o hello.s    2、编译
gcc -c hello.s -o hello.o    3、汇编
gcc hello.o -o hello_elf     4、链接
```

1: 预编译

将.c 中的头文件展开、宏展开
生成的文件是.i 文件

2: 编译

将预处理之后的.i 文件生成 .s 汇编文件

3、汇编

将.s 汇编文件生成.o 目标文件

4、链接

将.o 文件链接成目标文件

预处理有几种啊？

5.2 include

`#include <>` //用尖括号包含头文件，在系统指定的路径下找头文件

`#include ""` //用双引号包含头文件，先在当前目录下找头文件，找不到，再到系统指定的路径下找。

注意：`include` 经常用来包含头文件，可以包含 .c 文件，但是大家不要包含.c

因为 `include` 包含的文件会在预编译被展开，如果一个.c 被包含多次，展开多次，会导致函数重复定义。所以不要包含.c 文件。

注意：预处理只是对 `include` 等预处理操作进行处理并不会进行语法检查

这个阶段有语法错误也不会报错，第二个阶段即编译阶段才进行语法检查。

例 1:

main.c :

```
#include "fun.h"
```

```
int main(int argc, char *argv[])
{
    int num;
    num=max(10,20);
    return 0;
}
fun.h
int max(int x,int y);
```

编译: `gcc -E main.c -o main.i`

5.3 define

定义宏用 `define` 去定义

宏是在预编译的时候进行替换。

1、不带参宏

```
#define PI 3.14
```

在预编译的时候如果代码中出现了 `PI` 就用 `3.14` 去替换。

宏的好处: 只要修改宏定义, 其他地方在预编译的时候就会重新替换。

注意: 宏定义后边不要加分号。

例 2 :

```
#define PI 3.1415926
int main()
{
    double f;
    printf("%lf\n",PI);

    f=PI;
    return 0;
}
```

宏定义的作用范围, 从定义的地方到本文件末尾。

如果想在中间终止宏的定义范围

```
#undef PI //终止 PI 的作用
```

例 3 :

```
#define PI 3.1415926
int main()
{
    double f;
    printf("%lf\n",PI);
```

```
#undef PI

#define PI 3.14

    f=PI;
    return 0;
}
```

2、带参宏

```
#define S(a,b) a*b
```

注意带参宏的形参 a 和 b 没有类型名，

S(2,4) 将来在预处理的时候替换成 实参替代字符串的形参，其他字符保留，2 * 4

例 4：

```
#define S(a,b) a*b

int main(int argc, char *argv[])
{
    int num;
    num=S(2,4);

    return 0;
}
```

S(2+4,3)被替换成 2+4 * 3

注意：带参宏，是在预处理的时候进行替换

解决歧义方法

例 5：

```
#define S(a,b) (a)*(b)

int main(int argc, char *argv[])
{
    int num;
    num=S(2+3,5);//(2+3 ) *(5)

    return 0;
}
```

3、带参宏和带参函数的区别

带参宏被调用多少次就会展开多少次，执行代码的时候没有函数调用的过程，不需要压栈弹栈。所以带参宏，

是浪费了空间，因为被展开多次，节省时间。

带参函数，代码只有一份，存在代码段，调用的时候去代码段取指令，调用的时候要，压栈弹栈。有个调用的过程。

所以说，带参函数是浪费了时间，节省了空间。

带参函数的形参是有类型的，带参宏的形参没有类型名。

5.4 选择性编译

1、

```
#ifdef AAA
    代码段一
#else
    代码段二
#endif
```

如果在当前.c `ifdef` 上边定义过 `AAA` ，就编译代码段一，否则编译代码段二

注意和 `if else` 语句的区别，`if else` 语句都会被编译，通过条件选择性执行代码而 选择性编译，只有一块代码被编译

例:6 :

```
#define AAA

int main(int argc, char *argv[])
{
    #ifdef AAA
        printf("hello kitty!!\n");
    #else
        printf("hello 千锋 edu\n");
    #endif

    return 0;
}
```

2、

```
#ifndef AAA
    代码段一
#else
```

代码段二

```
#endif
```

和第一种互补。

这种方法，经常用在防止头文件重复包含。

防止头文件重复包含：

3、

```
#if 表达式
```

```
    程序段一
```

```
#else
```

```
    程序段二
```

```
#endif
```

如果表达式为真，编译第一段代码，否则编译第二段代码

选择性编译都是在预编译阶段干的事情。

5.5 静态库

一：动态编译

动态编译使用的是动态库文件进行编译

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

默认的咱们使用的是动态编译方法

二：静态编译

静态编译使用的静态库文件进行编译

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

三：静态编译和动态编译区别

1：使用的库文件的格式不一样

动态编译使用动态库，静态编译使用静态库

注意：

1：静态编译要把静态库文件打包编译到可执行程序中。

2：动态编译不会把动态库文件打包编译到可执行程序中，

它只是编译链接关系

例 7：

```
mytest.c
```

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

int main(int argc, char *argv[])
{
    int a=10,b=20,max_num,min_num;
    max_num=max(a,b);
    min_num=min(a,b);
    printf("max_num=%d\n",max_num);
    printf("min_num=%d\n",min_num);
    return 0;
}

mylib.c
int max(int x,int y)
{
    return (x>y)?x:y;
}
int min(int x,int y)
{
    return (x<y)?x:y;
}

mylib.h
extern int max(int x,int y);
extern int min(int x,int y);
```

制作静态库:

```
gcc -c mylib.c -o mylib.o
```

```
ar rc libtestlib.a mylib.o
```

注意：静态库起名的时候必须以 **lib** 开头以 **.a** 结尾

编译程序:

方法 1:

```
gcc -static mytest.c libtestlib.a -o mytest
```

方法 2:

 可以指定头文件及库文件的路径

比如咱们讲 libtestlib.a mylib.h 移动到/home/teacher 下

```
mv libtestlib.a mylib.h /home/teacher
```

编译程序命令:

```
gcc -static mytest.c -o mytest -L/home/teacher -ltestlib -I/home/teacher
```

注意: `-L` 是指定库文件的路径

`-l` 指定找哪个库, 指定的只要库文件名 `lib` 后面 `.a` 前面的部分

`-I` 指定头文件的路径

方法 3:

咱们可以将库文件及头文件存放到系统默认指定的路径下

库文件默认路径是 `/lib` 或者是 `/usr/lib`

头文件默认路径是 `/usr/include`

```
sudo mv libtestlib.a /usr/lib
```

```
sudo mv mylib.h /usr/include
```

编译程序的命令

```
gcc -static mytest.c -o mytest -ltestlib
```

5.6 动态库

制作动态链接库:

```
gcc -shared mylib.c -o libtestlib.so
```

//使用 `gcc` 编译、制作动态链接库

动态链接库的使用:

方法 1: 库函数、头文件均在当前目录下

```
gcc mytest.c libtestlib.so -o mytest
```

```
export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH
```

```
./mytest
```

方法 2: 库函数、头文件假设在 `/opt` 目录

```
gcc mytest.c -o mytest -L/home/teacher -ltestlib -I/home/teacher
```

编译通过, 运行时出错, 编译时找到了库函数, 但链接时找不到库, 执行以下操作, 把当前目录加入搜索路径

```
export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH
```

```
#!/mytest 可找到动态链接库
```

方法 3: 库函数、头文件均在系统路径下

```
cp libtestlib.so /usr/lib
```

```
cp mylib.h /usr/include
```

```
gcc mytest.c -o mytest -ltestlib
```

```
#!/mytest
```

问题: 有个问题出现了?

我们前面的静态库也是放在 `/usr/lib` 下, 那么连接的到底是动态库还是静态库呢?

当静态库与动态库重名时, 系统会优先连接动态库, 或者我们可以加入 `-static` 指定使用静态库

第6章 指针

6.1 指针的概念

6.1.1 关于内存那点事

存储器：存储数据器件

外存

外存又叫外部存储器，长期存放数据，掉电不丢失数据

常见的外存设备：硬盘、flash、rom、u 盘、光盘、磁带

内存

内存又叫内部存储器，暂时存放数据，掉电数据丢失

常见的内存设备：ram、DDR



物理内存：实实在在存在的存储设备

虚拟内存：操作系统虚拟出来的内存。

32bit 32 根

0x00 00 00 00

0xff ff ff ff

操作系统会在物理内存和虚拟内存之间做映射。

在 32 位系统下，每个进程（运行着的程序）的寻址范围是 4G,0x00 00 00 00 ~0xff ff ff ff

在写应用程序的，咱们看到的都是虚拟地址。

在运行程序的时候，操作系统会将 虚拟内存进行分区。

1.堆

在动态申请内存的时候，在堆里开辟内存。

2.栈

主要存放局部变量（在函数内部，或复合语句内部定义的变量）。

3.静态全局区

1): 未初始化的静态全局区

静态变量（定义的时候，前面加 static 修饰），或全局变量，没有初始化的，存在此区

2): 初始化的静态全局区

全局变量、静态变量，赋过初值的，存放在此区

4.代码区

存放咱们的程序代码

5.文字常量区

存放常量的。

内存以字节为单位来存储数据的，咱们可以将程序中的虚拟寻址空间，看成一个很大的一维的字符数组

6.1.2 指针的相关概念

操作系统给每个存储单元分配了一个编号，从 0x00 00 00 00 ~0xff ff ff ff

这个编号咱们称之为地址

指针就是地址

0xffff_ffff	
0xffff_ffffe	
0xffff_ffffd	
	...
	...
	...
0x0000_0003	
0x0000_0002	'\n'
0x0000_0001	'a'
0x0000_0000	100

指针变量：是个变量，是个指针变量，即这个变量用来存放一个地址编号

在 32 位平台下，地址总线是 32 位的，所以地址是 32 位编号，所以指针变量是 32 位的即 4 个字节。

注意：1:

无论什么类型的地址，都是存储单元的编号，在 32 位平台下都是 4 个字节，
即任何类型的指针变量都是 4 个字节大小

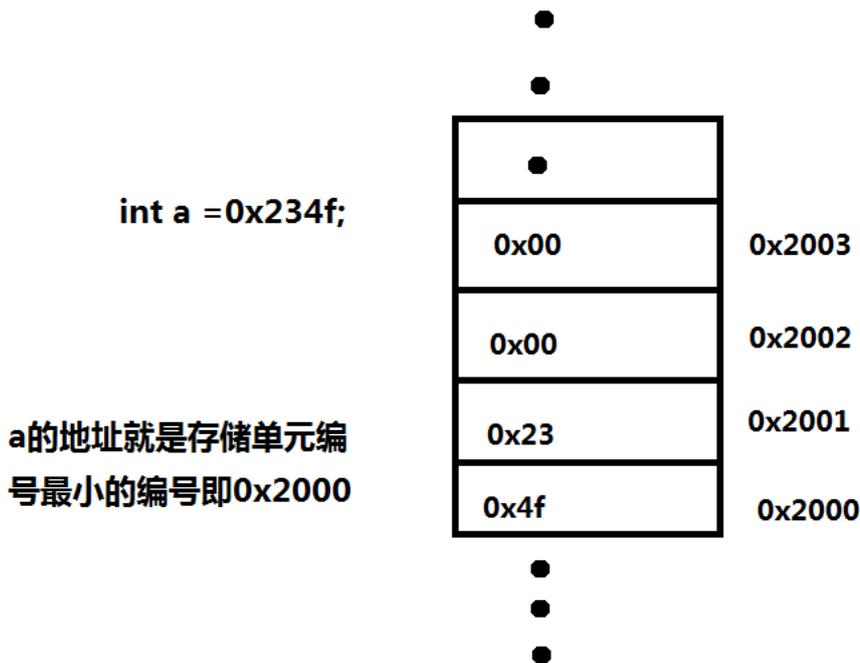
2: 对应类型的指针变量，只能存放对应类型的变量的地址

举例：整型的指针变量，只能存放整型变量的地址

扩展:

字符变量 char ch; ch 占 1 个字节，它有一个地址编号，这个地址编号就是 ch 的地址

整型变量 int a; a 占 4 个字节，它占有 4 个字节的存储单元，有 4 个地址编号。



Int a=0x00 00 23 4f

6.1.3 指针的定义方法

1.简单的指针

数据类型 * 指针变量名;

int * p;//定义了一个指针变量 p

在 定义指针变量的时候 * 是用来修饰变量的，说明变量 p 是个指针变量。

变量名是 p

2.关于指针的运算符

& 取地址 、 *取值

例 1 :

```
int a=0x0000234f;
```

```
int *p;//在定义指针变量的时候*代表修饰的意思，修饰 p 是个指针变量。
```

```
p=&a;//把 a 的地址给 p 赋值，&是取地址符。
```

变量 p 里存放的就是 0x00002000

p 保存了 a 的地址，也可以说 p 指向了 a

```
int num;
```

```
num=*p;//注意：在调用的时候 *代表取值得意思，*p 就相当于 p 指向的变量，即 a
```

所以说 num 的值为 0x234f;

扩展：如果在一行中定义多个指针变量，每个指针变量前面都需要加*来修饰

```
int *p,*q;//定义了两个整型的指针变量 p 和 q
```

```
int * p,q;//定义了一个整型指针变量 p，和整型的变量 q
```

例 2 :

```
int main()
{
    int a= 100, b = 200;
    int *p_1, *p_2 = &b; //表示该变量的类型是一个指针变量·指针变量名是 p_1 而不是*p_1.
    //p_1 在定义的时候没有赋初值·p_2 赋了初值
    p_1= &a ; //p_1 先定义后赋值
    printf("%d\n", a);
    printf("%d\n", *p_1);
    printf("%d\n", b);
    printf("%d\n", *p_2);
    return 0;
}
```

注意:

在定义 `p_1` 的时候, 因为是个局部变量, 局部变量没有赋初值, 它的值是随机的, `p_1` 指向哪里不一定, 所以 `p_1` 就是个野指针。

3. 指针大小

例 3 : 在 32 位系统下·所有类型的指针都是 4 个字节

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    char *p1;
    short int *p2;
    int *p3;
    long int *p4;
    float *p5;
    double *p6;

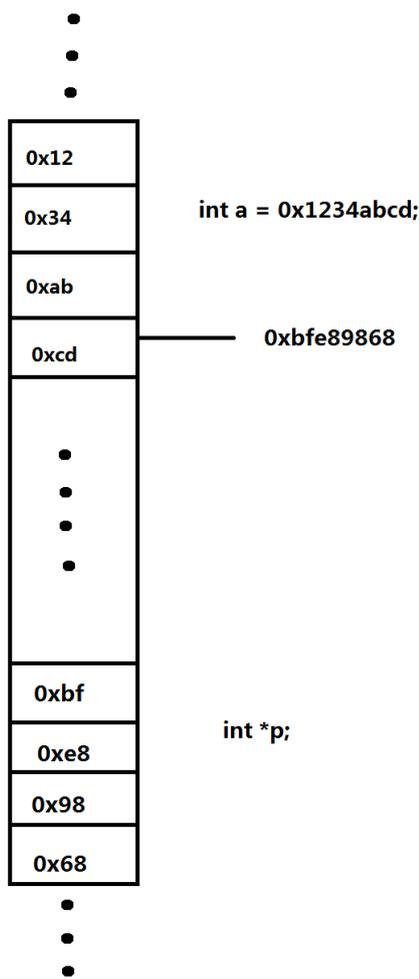
    printf("%d\n", sizeof(p1));
    printf("%d\n", sizeof(p2));
    printf("%d\n", sizeof(p3));
    printf("%d\n", sizeof(p4));
    printf("%d\n", sizeof(p5));
    printf("%d\n", sizeof(p6));
    return 0;
}
```

例 4 :

```
#include <stdio.h>
int main(int argc, char *argv[])
```

```
{
int a=0x1234abcd;
int *p;
p=&a;

printf("&a=%p\n",&a);
printf("p=%p\n",p);
return 0;
}
```



6.1.4 指针的分类

按指针指向的数据的类型来分

1:字符指针

字符型数据的地址

char *p;//定义了一个字符指针变量，只能存放字符型数据的地址编号

```
char ch;  
p= &ch;
```

2: 短整型指针

```
short int *p;//定义了一个短整型的指针变量 p，只能存放短整型变量的地址  
short int a;  
p =&a;
```

3: 整型指针

```
int *p;//定义了一个整型的指针变量 p，只能存放整型变量的地址  
int a;  
p =&a;  
注：多字节变量，占多个存储单元，每个存储单元都有地址编号，  
c 语言规定，存储单元编号最小的那个编号，是多字节变量的地址编号。
```

4: 长整型指针

```
long int *p;//定义了一个长整型的指针变量 p，只能存放长整型变量的地址  
long int a;  
p =&a;
```

5: float 型的指针

```
float *p;//定义了一个 float 型的指针变量 p，只能存放 float 型变量的地址  
float a;  
p =&a;
```

6: double 型的指针

```
double *p;//定义了一个 double 型的指针变量 p，只能存放 double 型变量的地址  
double a;  
p =&a;
```

7: 函数指针

8: 结构体指针

9: 指针的指针

10: 数组指针

总结:无论什么类型的指针变量，在 32 位系统下，都是 4 个字节，只能存放对应类型的变量的地址编号。

6.1.5 指针和变量的关系

指针可以存放变量的地址编号

在程序中，引用变量的方法

1:直接通过变量的名称

```
int a;  
a=100;
```

2:可以通过指针变量来引用变量

```
int *p;//在定义的时候，*不是取值的意思，而是修饰的意思，修饰 p 是个指针变量  
p=&a;//取 a 的地址给 p 赋值，p 保存了 a 的地址，也可以说 p 指向了 a  
*p= 100;//在调用的时候*是取值的意思，*指针变量 等价于指针指向的变量
```

注：指针变量在定义的时候可以初始化

```
int a;
```

```
int *p=&a;//用 a 的地址，给 p 赋值，因为 p 是指针变量
```

指针就是用来存放变量的地址的。

***+指针变量 就相当于指针指向的变量**

例 5 :

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

```
int main()
```

```
{
```

```
int *p1,*p2,temp,a,b;
```

```
p1=&a;
```

```
p2=&b;
```

```
printf("请输入:a b 的值:\n");
```

```
scanf("%d %d",p1,p2);//给 p1 和 p2 指向的变量赋值
```

```
temp = *p1; //用 p1 指向的变量 (a) 给 temp 赋值
```

```
*p1 = *p2; //用 p2 指向的变量 (b) 给 p1 指向的变量 (a) 赋值
```

```
*p2 = temp;//temp 给 p2 指向的变量 (b) 赋值
```

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

```
printf("*p1=%d *p2=%d\n",*p1,*p2);
```

```
return 0;
```

```
}
```

运行结果：

输入 100 200

运行结果为：

a=200 b=100

a=200 b=100

扩展：

对应类型的指针，只能保存对应类型数据的地址，

如果想让不同类型的指针相互赋值的时候，需要强制类型转换

```
void * p;
```

例 6 :

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

```
int main()
```

```
{
```

```
int a=0x1234,b=0x5678;
```

```
char *p1,*p2;
```

```
printf("%0x %0x\n",a,b);
```

```
p1=(char *)&a;
```

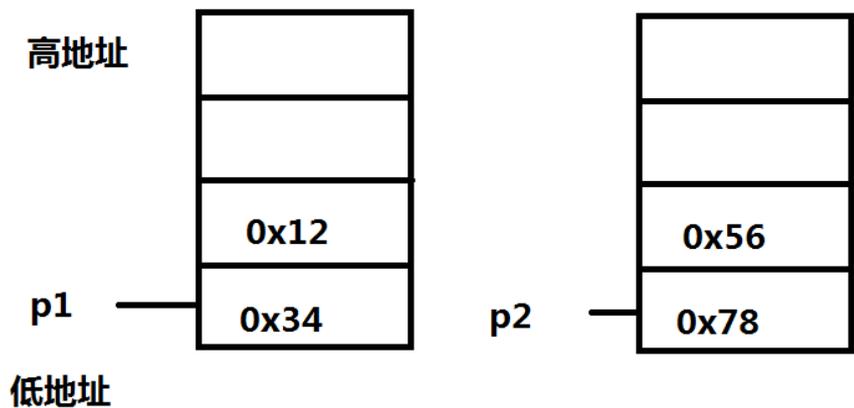
```

p2=(char *)&b;
printf("%0x %0x\n",*p1,*p2);
p1++;
p2++;
printf("%0x %0x\n",*p1,*p2);
return 0;
}

```

int a = 0x1234;

int b = 0x5678;



结果为:

0x 34 0x78

0x12 0x56

注意:

- 1: *+指针 取值，取几个字节，由指针类型决定的指针为字符指针则取一个字节，指针为整型指针则取 4 个字节，指针为 double 型指针则取 8 个字节。
- 2: 指针++ 指向下个对应类型的数据
字符指针++，指向下个字符数据，指针存放的地址编号加 1
整型指针++，指向下个整型数据，指针存放的地址编号加 4

6.1.6 指针和数组元素之间的关系

1、

变量存放在内存中，有地址编号，咱们定义的数组，是多个相同类型的变量的集合，每个变量都占内存空间，都有地址编号
指针变量当然可以存放数组元素的地址。

例 7 :

```

int a[10];
//int *p = &a[0];
int *p;
p=&a[0];//

```

指针变量 p 保存了数组 a 中第 0 个元素的地址，即 a[0]的地址

2、数组元素的引用方法

方法 1: 数组名[下标]

```
int a[10];  
a[2]=100;
```

方法 2: 指针名加下标

```
int a[10];  
int *p;  
p=a;  
p[2]=100;//因为 p 和 a 等价
```

补充: c 语言规定: 数组的名字就是数组的首地址, 即第 0 个元素的地址, 是个常量。

注意: p 和 a 的不同, p 是指针变量, 而 a 是个常量。所以可以用等号给 p 赋值, 但不能给 a 赋值。

方法 3: 通过指针运算加取值的方法来引用数组的元素

```
int a[10];  
int *p;  
p=a;  
*(p+2)=100;//也是可以的, 相当于 a[2]=100  
解释: p 是第 0 个元素的地址, p+2 是 a[2]这个元素的地址。  
对第二个元素的地址取值, 即 a[2]
```

例 8:

```
#include <stdio.h>  
int main(int argc, char *argv[])  
{  
    int a[5]={0,1,2,3,4};  
    int *p;  
    p=a;  
    printf("a[2]=%d\n",a[2]);  
    printf("p[2]=%d\n",p[2]);  
    printf("*(p+2)%d\n",*(p+2));  
    printf("*(a+2)%d\n",*(a+2));  
    printf("p=%p\n",p);  
    printf("p+2=%p\n",p+2);  
    return 0;  
}
```

3、指针的运算

1: 指针可以加一个整数,往下指几个它指向的变量, 结果还是个地址

前提: 指针指向数组的时候, 加一个整数才有意义

```
例 9 :  
int a[10];
```

```
int *p;  
p=a;  
p+2;//p 是 a[0]的地址 · p+2 是&a[2]
```

假如 p 保存的地址编号是 2000 的话，p+2 代表的地址编号是 2008

例 10 :

```
char buf[10] ;  
char *q;  
q=buf;  
q+2 //相当于&buf [2]
```

假如：q 中存放的地址编号是 2000 的话，q+2 代表的地址编号是 2002

2: 两个相同类型指针可以比较大小

前提：只有两个**相同类型的指针指向同一个数组的元素**的时候，比较大小才有意义
指向前面元素的指针 小于 指向后面 元素的指针

例 11 :

```
#include <stdio.h>  
int main(int argc, char *argv[])  
{  
    int a[10];  
    int *p,*q,n;//如果在一行上定义多个指针变量的 · 每个变量名前面加*  
    //上边一行定义了两个指针 p 和 q · 定义了一个整型的变量 n  
    p=&a[1];  
    q=&a[6];  
    if(p<q)  
    {  
        printf("p<q\n");  
    }  
    else if(p>q)  
    {  
        printf("p>q\n");  
    }  
    else  
    {  
        printf("p == q\n");  
    }  
    return 0;  
}
```

结果是 p<q

3.两个相同类型的指针可以做减法

前提：必须是两个相同类型的指针指向同一个数组的元素的时候，做减法才有意义
做减法的结果是，两个指针指向的中间有多少个元素

例 12:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a[10];
    int *p,*q;
    p=&a[0];
    q=&a[3];
    printf("%d\n",q-p);
    return 0;
}
```

结果是 3

4: 两个相同类型的指针可以相互赋值

注意:只有相同类型的指针才可以相互赋值 (void *类型的除外)

```
int *p;
int *q;
int a;
p=&a;//p 保存 a 的地址, p 指向了变量 a
q=p;//用 p 给 q 赋值, q 也保存了 a 的地址, 指向 a
```

注意：如果类型不相同的指针要想相互赋值，必须进行强制类型转换

注意: c 语言规定数组的名字，就是数组的首地址，就是数组第 0 个元素的地址

```
int *p;
int a[10];
p=a; p=&a[0];这两种赋值方法是等价的
```

6.1.7 指针数组

1、指针和数组的关系

1: 指针可以保存数组元素的地址

2: 可以定义一个数组，数组中有若干个相同类型指针变量，这个数组被称为指针数组

指针数组的概念：

指针数组本身是个数组，是个指针数组，是若干个相同类型的指针变量构成的集合

2、指针数组的定义方法：

类型说明符 * 数组名 [元素个数];

int * p[10];//定义了一个整型的指针数组 p，有 10 个元素 p[0]~p[9].

每个元素都是 int *类型的变量

```
int a;
p[1]=&a;
```

```
int b[10];
```

```
p[2]=&b[3];
```

p[2]、*(p+2)是等价的，都是指针数组中的第 2 个元素。

3、指针数组的分类

字符指针数组 char *p[10]、短整型指针数组、整型的指针数组、长整型的指针数组

float 型的指针数组、double 型的指针数组

结构体指针数组、函数指针数组

例 13 :

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

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

```
{
```

```
    char *name[5] = {"Follw me","BASIC","Greatwall","FORTRAN","Computer"};
```

```
    int i;
```

```
    for(i=0;i<5;i++)
```

```
    {
```

```
        printf("%s\n",name[i]);
```

```
    }
```

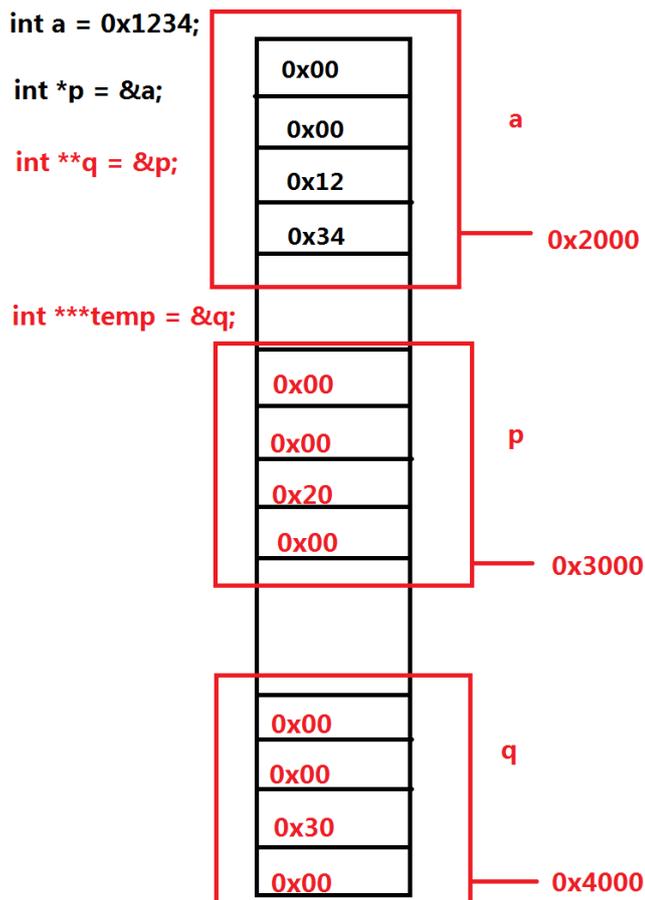
```
    return 0;
```

```
}
```

6.1.8 指针的指针

指针的指针，即指针的地址，

咱们定义一个指针变量本身指针变量占 4 个字节，指针变量也有地址编号



```

int a;
int *p;
p=&a;
*p == a
int **q;
q=&p;
*q == p
**q == *p == a
int ***m;
m=&q;
*(*m) == a

```

p q m 都是指针变量，都占 4 个字节，都存放地址编号，只不过类型不一样而已

6.1.9 字符串和指针

字符串的概念：

字符串就是以'\0'结尾的若干的字符的集合

字符串的存储形式：数组、字符串指针、堆

1、char string[100] = "I love C!"

定义了一个字符数组 string,用来存放多个字符，并且用"I love C!"给 string 数组初始化，字符串"I love C!"存放在 string 中

2、char *str = "I love C!"

定义了一个指针变量 str,只能存放字符地址编号，所以说 I love C! 这个字符串中的字符不能存放在 str 指针变量中。str 只是存放了字符 I 的地址编号，“I love C!”存放在文字常量区

3、char *str=(char*)malloc(10*sizeof(char));//动态申请了 10 个字节的存储空间，首地址给 str 赋值。

strcpy(str,"I love C"); //将字符串 "I love C!" 拷贝到 str 指向的内存里

➤ 字符数组：

在内存（栈、静态全局区）中开辟了一段空间存放字符串

➤ 字符串指针：

在文字常量区开辟了一段空间存放字符串，将字符串的首地址付给 str

➤ 堆：

使用 malloc 函数在堆区申请空间，将字符串拷贝到堆区

注意：

可修改性：

1. 栈和全局区内存中的内容是可修改的

```
char str[100]="I love C!";  
str[0]='y';//正确可以修改的
```

2. 文字常量区里的内容是不可修改的

```
char *str="I love C!";  
*str='y';//错误，I 存放在文字常量区，不可修改
```

3. 堆区的内容是可以修改的

```
char *str=(char*)malloc(10*sizeof(char));  
strcpy(str,"I love C");  
*str='y';//正确，可以，因为堆区内容是可修改的
```

注意：str 指针指向的内存能不能被修改，要看 str 指向哪里。

str 指向文字常量区的时候，内存里的内容不可修改

str 指向栈、堆、静态全局区的时候，内存的内容是可以修改

初始化：

字符数组、指针指向的字符串：定义时直接初始化

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

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

堆中存放的字符串

不能初始化、只能使用 strcpy、scanf 赋值

```
char *buf_heap;
```

```
buf_heap=(char *)malloc(15);
```

```
strcpy(buf_heap,"hello world");
```

```
scanf("%s",buf_heap);
```

使用时赋值

字符数组：使用 scanf 或者 strcpy

buf_aver="hello kitty"; 错误,因为字符数组的名字是个常量

strcpy(buf_aver,"hello kitty"); 正确

scanf("%s",buf_aver); 正确

指向字符串的指针：

buf_point="hello kitty"; 正确,buf_point 指向另一个字符串

strcpy(buf_point,"hello kitty"); 错误，只读,能不能复制字符串到 buf_piont 指向的内存里
取决于 buf_point 指向哪里。

6.1.10 数组指针

1、二维数组

二维数组，有行，有列。二维数组可以看成有多个一维数组构成的，是多个一维数组的集合，可以认为二维数组的每一个元素是个一维数组。

例：

```
int a[3][5];
```

定义了一个 3 行 5 列的一个二维数组。

可以认为二维数组 a 由 3 个一维数组构成，每个元素是一个一维数组。

回顾：

数组的名字是数组的首地址，是第 0 个元素的地址，是个常量，数组名字加 1 指向下个元素

二维数组 a 中，a+1 指向下个元素，即下一个一维数组，即下一行。

例 14：

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

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

```
{
```

```
    int a[3][5];
```

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

```
    printf("a+1=%p\n",a+1);
```

```
    return 0;
```

```
}
```

2、数组指针的概念：

本身是个指针，指向一个数组，加 1 跳一个数组，即指向下个数组。

3、数组指针的定义方法：

指向的数组的类型 (*指针变量名) [指向的数组的元素个数]

```
int (*p)[5];//定义了一个数组指针变量 p，p 指向的是整型的有 5 个元素的数组
```

p+1 往下指 5 个整型，跳过一个有 5 个整型元素的数组。

例 15 :

```
#include <stdio.h>
int main()
{
    int a[3][5]; // 定义了一个 3 行 5 列的一个二维数组
    int (*p)[5]; // 定义一个数组指针变量 p, p+1 跳一个有 5 个元素的整型数组
    printf("a=%p\n", a); // 第 0 行的行地址
    printf("a+1=%p\n", a+1); // 第 1 行的行地址, a 和 a +1 差 20 个字节
    p=a;
    printf("p=%p\n", p);
    printf("p+1=%p\n", p+1); // p+1 跳一个有 5 个整型元素的一维数组
    return 0;
}
```

例 16 : 数组指针的用法 1

```
#include <stdio.h>
void fun(int (*p)[5], int x, int y)
{
    p[0][1]=101;
}
int main()
{
    int i, j;
    int a[3][5];
    fun(a, 3, 5);
    for(i=0; i<3; i++)
    {
        for(j=0; j<5; j++)
        {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}
```

4、各种数组指针的定义:

(1)、一维数组指针, 加 1 后指向下一个一维数组

```
int (*p)[5];
```

配合每行有 5 个 int 型元素的二维数组来用

```
int a[3][5]
int b[4][5]
int c[5][5]
int d[6][5]
```

.....

```
p=a;
p=b;
p=c;
p=d;
```

都是可以的~~~~

(2)、二维数组指针，加 1 后指向下个二维数组

```
int(*p)[4][5];
```

配合三维数组来用，三维数组中由若干个 4 行 5 列二维数组构成

```
int a[3][4][5];
int b[4][4][5];
int c[5][4][5];
int d[6][4][5];
```

这些三维数组，有个共同的特点，都是有若干个 4 行 5 的二维数组构成。

```
p=a;
p=b;
p=c;
p=d;
```

例 17 :

```
#include <stdio.h>
int main()
{
    int a[3][4][5];
    printf("a=%p\n",a);
    printf("a+1=%p\n",a+1);//a 和 a+1 地址编号相差 80 个字节
    //验证了 a+1 跳一个 4 行 5 列的一个二维数组

    int(*p)[4][5];
    p=a;
    printf("p=%p\n",p);
    printf("p+1=%p\n",p+1);//p 和 p+1 地址编号相差也 80 个字节
    return 0;
}
```

5、三维数组指针，加 1 后指向下个三维数组

```
int(*p)[4][5][6];
```

p+1 跳一个三维数组；

什么样的三维数组啊？

由 4 个 5 行 6 列的二维数组构成的三维数组

配合：

```
int a[7][4][5][6];
```

6、四维数组指针，加 1 后指向下个四维数组，以此类推。。。

7、注意：

容易混淆的内容：

指针数组：是个数组，有若干个相同类型的指针构成的集合

```
int *p[10];
```

数组 p 有 10 个 int *类型的指针变量构成，分别是 p[0] ~ p[9]

数组指针：本身是个指针，指向一个数组，加 1 跳一个数组

```
int (*p)[10];
```

P 是个指针，p 是个数组指针，p 加 1 指向下个数组，跳 10 个整形。

指针的指针：

```
int **p;//p 是指针的指针
```

```
int *q;
```

```
p=&q;
```

8、数组名字取地址：变成 数组指针

一维数组名字取地址，变成一维数组指针，即加 1 跳一个一维数组

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

a+1 跳一个整型元素，是 a[1]的地址

a 和 a+1 相差一个元素，4 个字节

&a 就变成了一个一维数组指针，是 int(*p)[10]类型的。

(&a)+1 和 &a 相差一个数组即 10 个元素即 40 个字节。

例 18：

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

```
int main()
```

```
{
```

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

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

```
printf("a+1=%p\n",a+1);
```

```
printf("&a=%p\n",&a);
```

```
printf("&a +1=%p\n",&a+1);
```

```
return 0;
```

```
}

```

a 是个 int *类型的指针，是 a[0]的地址。

&a 变成了数组指针，加 1 跳一个 10 个元素的整型一维数组

在运行程序时，大家会发现 a 和&a 所代表的地址编号是一样的，即他们指向同一个存储单元，但是 a 和&a 的指针类型不同。

```
例 19 :
int a[4][5];
a+1 跳 5 个整型
(&a)+1 跳 4 行 5 列 ( 80 个字节)。
```

总结：c 语言规定，数组名字取地址，变成了数组指针。加 1 跳一个数组。

9、数组名字和指针变量的区别：

```
int a[10];
int *p;
p=a;
```

相同点：

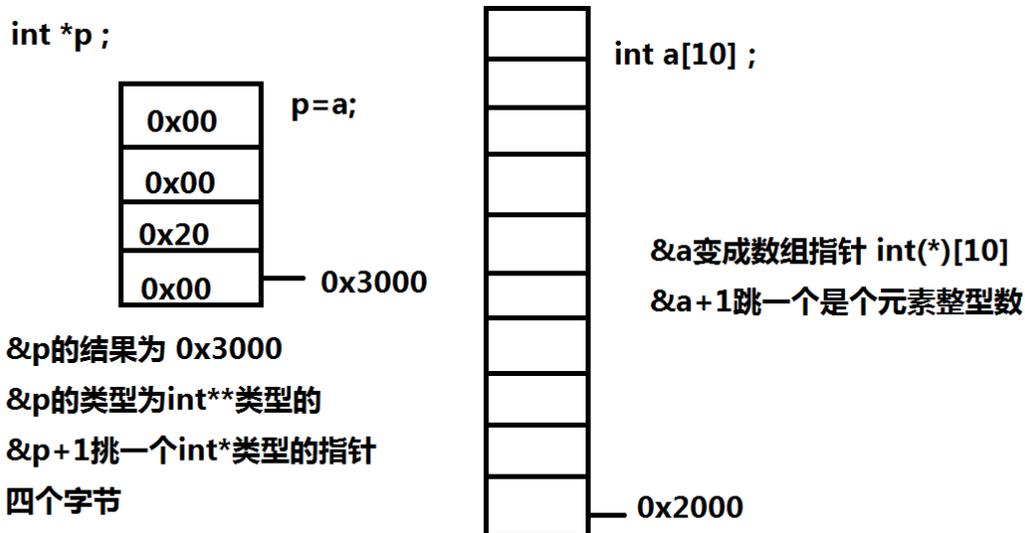
a 是数组的名字，是 a[0]的地址，p=a 即 p 也保存了 a[0]的地址，即 a 和 p 都指向 a[0]，所以在引用数组元素的时候，a 和 p 等价

引用数组元素回顾：

a[2]、*(a+2)、p[2]、*(p+2) 都是对数组 a 中 a[2]元素的引用。

不同点：

- 1、a 是常量、p 是变量
可以用等号 '=' 给 p 赋值，但是不能用等号给 a 赋值
- 2、对 a 取地址，和对 p 取地址结果不同
因为 a 是数组的名字，所以对 a 取地址结果为数组指针。
p 是个指针变量，所以对 p 取地址 (&p) 结果为指针的指针。



例 20 :

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a[10];
    int *p;
    p=a;
    printf("a=%p\n",a);
    printf("&a=%p\n",&a);
    printf("&a +1 =%p\n",&a +1);

    printf("p=%p\n",p);
    printf("&p=%p\n",&p);
    printf("&p +1=%p\n",&p +1);
    return 0;
}
```

10、多维数组中指针的转换:

在二维数组中，行地址取 * 不是取值得意思，而是指针降级的意思，由行地址（数组指针）变成这一行第 0 个元素的地址。取*前后还是指向同一个地方，但是指针的类型不一样了

例 21 :

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a[3][5];
    printf("a=%p\n",a);
    printf("a +1=%p\n",a+1);

    printf("*a =%p\n",*a);// *a 变成了第 0 行第 0 列元素的地址
    printf("(a)+1 =%p\n",(*a)+1 ); //结果为第 0 行第 1 列元素的地址
    return 0;
}
```

6.1.11 指针和函数的关系

1、指针作为函数的参数

咱们可以给一个函数传一个 整型、字符型、浮点型的数据，也可以给函数传一个地址。

例:

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

函数传参:

(1)、传数值:

例 22:

```
void swap(int x,int y)  
{  
    int temp;  
    temp=x;  
    x=y;  
    y=temp;  
}  
int main()  
{  
    int a=10,b=20;  
    swap(a,b);  
    printf("a=%d b=%d\n",a,b);//a=10  b=20  
}
```

实参: 调用函数时传的参数。

形参: 定义被调函数时, 函数名后边括号里的数据

结论: 给被调函数传数值, 只能改变被调函数形参的值, 不能改变主调函数实参的值

(2)、传地址:

例 23 :

```
void swap(int *p1,int *p2)  
{  
    int temp;  
    temp= *p1;  
    *p1=*p2;// p2 指向的变量的值 · 给 p1 指向的变量赋值  
    *p2=temp;  
}  
int main()  
{  
    int a=10,b=20;  
    swap(&a,&b);  
    printf("a=%d b=%d\n",a,b);//结果为 a=20 b=10  
}
```

结论: 调用函数的时候传变量的地址, 在被调函数中通过*+地址来改变主调函数中的变量的值

例 24 :

```
void swap(int *p1,int *p2)//&a &b
{
    int *p;
    p=p1;
    p1=p2;//p1 =&b 让 p1 指向 main 中的 b
    p2=p;//让 p2 指向 main 函数中 a
} //此函数中改变的是 p1 和 p2 的指向·并没有给 main 中的 a 和 b 赋值
int main()
{
    int a=10,b=20;
    swap(&a,&b);
    printf("a=%d b=%d\n",a,b);//结果为 a=10 b=20
}
```

总结一句话：要想改变主调函数中变量的值，必须传变量的地址，而且还得通过*+地址 去赋值。无论这个变量是什么类型的。

例 25 :

```
void fun(char *p)
{
    p="hello kitty";
}
int main()
{
    char *p="hello world";
    fun(p);
    printf("%s\n",p);//结果为 : hello world
}
```

答案分析：

在 fun 中改变的是 fun 函数中的局部变量 p，并没有改变 main 函数中的变量 p，所以 main 函数中的，变量 p 还是指向 hello world。

例 26 :

```
void fun(char **q)
{
    *q="hello kitty";
}
int main()
{
    char *p="hello world";
```

```
    fun(&p);  
    printf("%s\n",p);//结果为 : hello kitty  
}
```

总结一句话：要想改变主调函数中变量的值，必须传变量的地址，而且还得通过*+地址 去赋值。无论这个变量是什么类型的。

(3)、传数组：

给函数传数组的时候，没法一下将数组的内容作为整体传进去。
只能传数组的地址。

例 27：传一维数组的地址

```
//void fun(int p[])//形式 1  
void fun(int *p)//形式 2  
{  
    printf("%d\n",p[2]);  
    printf("%d\n",*(p+3));  
}  
int main()  
{  
    int a[10]={1,2,3,4,5,6,7,8};  
    fun(a);  
    return 0;  
}
```

例 28：传二维数组的地址

```
//void fun( int p[][4] )//形式 1  
void fun( int (*p)[4] )//形式 2  
{  
}  
int main()  
{  
    int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};  
    fun(a);  
    return 0;  
}
```

例 29：传指针数组

```
void fun(char **q) // char *q[]  
{  
    int i;  
    for(i=0;i<3;i++)
```

```
        printf("%s\n",q[i]);
    }
    int main()
    {
        char *p[3]={"hello","world","kitty"}; //p[0] p[1] p[2] char *
        fun(p);
        return 0;
    }
```

2、指针作为函数的返回值

一个函数可以返回整型数据、字符数据、浮点型的数据，
也可以返回一个指针。

例 30:

```
char * fun()
{
    char str[100]="hello world";
    return str;
}
int main()
{
    char *p;
    p=fun();
    printf("%s\n",p);
}
```

//总结，返回地址的时候，地址指向的内存的内容不能释放

如果返回的指针指向的内容已经被释放了，返回这个地址，也没有意义了。

例 31: 返回静态局部数组的地址

```
char * fun()
{
    static char str[100]="hello world";
    return str;
}
int main()
{
    char *p;
    p=fun();
    printf("%s\n",p); //hello world
}
```

原因是，静态数组的内容，在函数结束后，亦然存在。

例 32：返回文字常量区的字符串的地址

```
char * fun()
{
    char *str="hello world";
    return str;
}
int main()
{
    char *p;
    p=fun();
    printf("%s\n",p);//hello world
}
```

原因是文字常量区的内容，一直存在。

例 33：返回堆内存的地址

```
char * fun()
{
    char *str;
    str=(char *)malloc(100);
    strcpy(str,"hello world");
    return str;
}
int main()
{
    char *p;
    p=fun();
    printf("%s\n",p);//hello world
    free(p);
}
```

原因是堆区的内容一直存在，直到 `free` 才释放。

总结：返回的地址，地址指向的内存的内容得存在，才有意义。

3、指针保存函数的地址

咱们定义的函数，在运行程序的时候，会将函数的指令加载到内存的代码段。所以函数也有起始地址。

c 语言规定：函数的名字就是函数的首地址，即函数的入口地址

咱们就可以定义一个指针变量，来存放函数的地址。

这个指针变量就是函数指针变量。

(1): 函数指针变量的定义方法

返回值类型(*函数指针变量名)(形参列表);

`int(*p)(int,int);`//定义了一个函数指针变量 p,p 指向的函数
必须有一个整型的返回值，有两个整型参数。

```
int max(int x,int y)
{
}

```

```
int min(int x,int y)
{
}

```

可以用这个 p 存放这类函数的地址。

```
p=max;
p=min;
```

(2) :调用函数的方法

1.通过函数的名字去调函数（最常用的）

```
int max(int x,int y)
{
}

```

```
int main()
{
    int num;
    num=max(3,5);
}

```

2.可以通过函数指针变量去调用

```
int max(int x,int y)
{
}
int main()
{
    int num;
    int (*p)(int ,int);
}

```

```
p=max;
num=(*p)(3,5);
}
```

(3):函数指针数组

```
int(*p[10])(int,int);
```

定义了一个函数指针数组，有 10 个元素 p[0] ~p[9]，每个元素都是函数指针变量，指向的函数，必须有整型的返回值，两个整型参数。

(4):函数指针最常用的地方

给函数传参

```
#include<stdio.h>
int add(int x,int y)
{
    return x+y;
}
int sub(int x,int y)
{
    return x-y;
}
int mux(int x,int y)
{
    return x*y;
}
int dive(int x,int y)
{
    return x/y;
}
int process(int (*p)(int ,int),int a,int b)
{
    int ret;
    ret = (*p)(a,b);
    return ret;
}
int main()
{
    int num;
    num = process(add,2,3);
    printf("num =%d\n",num);
}
```

```
num = process(sub,2,3);
printf("num =%d\n",num);

num = process(mux,2,3);
printf("num =%d\n",num);

num = process(dive,2,3);
printf("num =%d\n",num);
return 0;
}
```

6.1.12 经常容易混淆的指针

第一组:

1、 `int *a[10];`

这是个指针数组，数组 a 中有 10 个整型的指针变量

`a[0]~a[9]`

2、 `int (*a)[10];`

数组指针变量，它是个指针变量。它占 4 个字节，存地址编号。

它指向一个数组，它加 1 的话，指向下个数组。

3、 `int **p;`

这个是个指针的指针，保存指针变量的地址。

它经常用在保存指针的地址：

常见用法 1:

```
int **p
```

```
int *q;
```

```
p=&q;
```

常见用法 2:

```
int **p;
```

```
int *q[10];
```

分析：q 是指针数组的名字，是指针数组的首地址，是 `q[0]` 的地址。

`q[0]` 是个 `int *` 类型的指针。所以 `q[0]` 指针变量的地址，是 `int **` 类型的

`p=&q[0];` 等价于 `p=q;`

例 34 :

```
void fun(char**p)
```

```
{
```

```
int i;
```

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

```

    {
        printf("%s\n",p[i]);/* ( p+i )
    }
}
int main()
{
    char *q[3]={"hello","world","China"};
    fun(q);
    return 0;
}

```

第二组：

1、`int *f(void)`;

注意：`*f` 没有用括号括起来

它是个函数的声明，声明的这个函数返回值为 `int *`类型的。

2、`int (*f)(void)`;

注意`*f` 用括号括起来了，`*`修饰 `f` 说明，`f` 是个指针变量。

`f` 是个函数指针变量，存放函数的地址，它指向的函数，

必须有一个 `int` 型的返回值，没有参数。

6.1.13 特殊指针

1、空类型的指针 (`void *`)

`char *` 类型的指针指向 `char` 型的数据

`int *` 类型的指针指向 `int` 型的数据

`float*` 类型的指针指向 `float` 型的数据

`void *` 难道是指向 `void` 型的数据吗？

不是，因为没有 `void` 类型的变量

回顾：对应类型的指针只能存放对应类型的数据地址

`void*` 通用指针，任何类型的指针都可以给 `void*`类型的指针变量赋值。

`int *p;`

`void *q;`

`q=p` 是可以的，不用强制类型转换

举例子：

有个函数叫 `memset`

`void * memset(void *s,int c,size_t n)`;

这个函数的功能是将 `s` 指向的内存前 `n` 个字节，全部赋值为 `c`。

`Memset` 可以设置字符数组、整型数组、浮点型数组的内容，所以第一个参数，就必须是个通用指针

它的返回值是 `s` 指向的内存的首地址，可能是不同类型的地址。所以返回值也得是通用指针

注意：`void*`类型的指针变量，也是个指针变量，在 32 为系统下，占 4 个字节

2、NULL

空指针:

```
char *p=NULL;
```

咱们可以认为 `p` 哪里都不指向，也可以认为 `p` 指向内存编号为 0 的存储单位。

在 `p` 的四个字节中，存放的是 `0x00 00 00 00`

一般 `NULL` 用在给指针初始化。

`main` 函数传参:

例 35 :

```
#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;
}
```

第7章 动态内存申请

7.1 动态分配内存的概述

在数组一章中，介绍过数组的长度是预先定义好的，在整个程序中**固定不变**，但是在实际的编程中，往往会发生这种情况，即所需的**内存空间取决于实际输入的数据**，而无法预先确定。为了解决上述问题，C 语言提供了一些**内存管理函数**，这些内存管理函数可以按需要**动态的分配**内存空间，也可把不再使用的空间回收再次利用。

7.2 静态分配、动态分配

静态分配

- 1、在程序编译或运行过程中，按事先规定大小分配内存空间的分配方式。`int a [10]`
- 2、必须事先知道所需空间的大小。
- 3、分配在栈区或全局变量区，一般以数组的形式。

4、按计划分配。

动态分配

- 1、在程序运行过程中，根据需要大小自由分配所需空间。
- 2、按需分配。
- 3、分配在堆区，一般使用特定的函数进行分配。

7.3 动态分配函数

stdlib.h

1、malloc 函数

函数原型：`void *malloc(unsigned int size);`

功能说明：

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。函数原型返回 `void*` 指针，使用时必须做相应的强制类型转换，分配的内存空间内容不确定，一般使用 `memset` 初始化。

返回值：分配空间的起始地址（分配成功）

NULL （分配失败）

注意

- 1、在调用 `malloc` 之后，一定要判断一下，是否申请内存成功。
- 2、如果多次 `malloc` 申请的内存，第 1 次和第 2 次申请的内存不一定是连续的

例 1：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
    int count,*array,n;
    printf("请输入您要申请的数组元素个数\n");
    scanf("%d",&n);
    array=(int *)malloc(n*sizeof(int));
    if(array==NULL)
    {
        printf("申请内存失败\n");
        return 0;
    }
    memset(array,0,n*sizeof(int));
    for(count=0;count<n;count++)
    {
        array[count]=count;
    }
    for(count=0;count<n;count++)
```

```
{
    printf("%d\n",array[count]);
}
free(array);//释放 array 指向的内存
return 0;
}
```

2、free 函数（释放内存函数）

头文件：`#include<stdlib.h>`

函数定义：`void free(void *ptr)`

函数说明：`free` 函数释放 `ptr` 指向的内存。

注意 `ptr` 指向的内存必须是 `malloc` `calloc` `realloc` 动态申请的内存

例 2：

```
char *p=(char *)malloc(100);
free(p);//
```

注意

(1)、`free` 后，因为没有给 `p` 赋值，所以 `p` 还是指向原先动态申请的内存。但是内存已经不能再用了，`p` 变成野指针了。

(2)、一块动态申请的内存只能 `free` 一次，不能多次 `free`

3、calloc 函数

头文件：`#include<stdlib.h>`

函数定义：`void * calloc(size_t nmemb,size_t size);`

`size_t` 实际是无符号整型，它是在头文件中，用 `typedef` 定义出来的。

函数的功能：在内存的堆中，申请 `nmemb` 块，每块的大小为 `size` 个字节的连续区域

函数的返回值：

返回 申请的内存的首地址（申请成功）

返回 `NULL`（申请失败）

注意：

`malloc` 和 `calloc` 函数都是用来申请内存的。

区别：

- 1) 函数的名字不一样
- 2) 参数的个数不一样
- 3) `malloc` 申请的内存，内存中存放的内容是随机的，不确定的，而 `calloc` 函数申请的内存中的内容为 0

例 3：调用方法

```
char *p=(char *)calloc(3,100);
```

在堆中申请了 3 块，每块大小为 100 个字节，即 300 个字节连续的区域。

4、realloc 函数（重新申请内存）

咱们调用 malloc 和 calloc 函数单次申请的内存是连续的，两次申请的两块内存不一定连续。

有些时候有这种需求，即我先用 malloc 或者 calloc 申请了一块内存，我还想在原先内存的基础上挨着申请内存。或者我开始时候使用 malloc 或 calloc 申请了一块内存，我想释放后边的一部分内存。

为了解决这个问题，发明了 realloc 这个函数

头文件#include<stdlib.h>

函数的定义: void* realloc(void *s,unsigned int newsize);

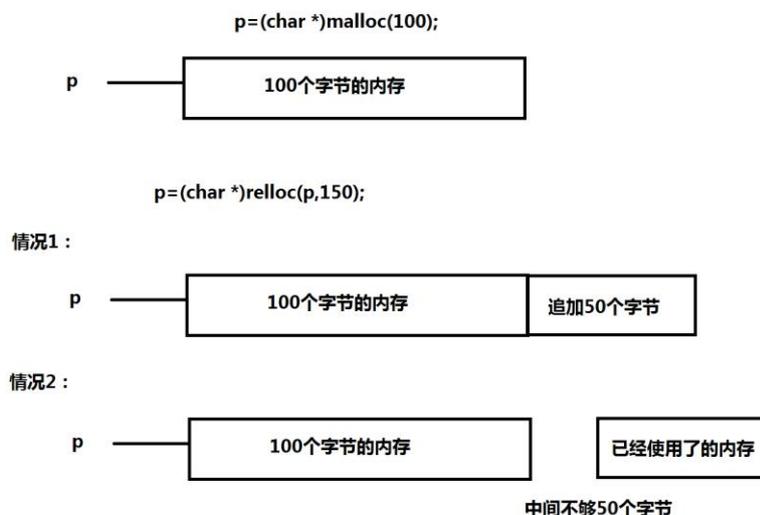
函数的功能:

在原先 s 指向的内存基础上重新申请内存，新的内存的大小为 new_size 个字节，如果原先内存后面有足够大的空间，就追加，如果后边的内存不够用，则 realloc 函数会在堆区找一个 newsize 个字节大小的内存申请，将原先内存中的内容拷贝过来，然后释放原先的内存，最后返回新内存的地址。

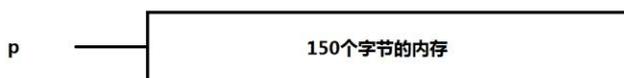
如果 newsize 比原先的内存小，则会释放原先内存的后面的存储空间，只留前面的 newsize 个字节
返回值: 新申请的内存的首地址

例 4:

```
char *p;
p=(char *)malloc(100)
//咱们想在 100 个字节后面追加 50 个字节
p=(char *)realloc(p,150);//p 指向的内存的新的大小为 150 个字节
```



如果出现这种情况，realloc函数会在堆区申请一个连续的150个字节的内存，将原来的100个字节拷贝过来，然后将原来的内存释放掉，然后返回新内存的地址。



例 5:

```
char *p;
p=(char *)malloc(100)
```

```
//咱们想重新申请内存,新的大小为 50 个字节  
p=(char *)realloc(p,50);//p 指向的内存的新的大小为 50 个字节,100 个字节的后 50 个字节的存储空间就被释放了
```

注意:malloc calloc realloc 动态申请的内存,只有在 free 或程序结束的时候才释放。

7.4 内存泄露

内存泄露的概念:

申请的内存,首地址丢了,找不了,再也没法使用了,也没法释放了,这块内存就被泄露了。

内存泄露 例 1:

```
int main()  
{  
    char *p;  
    p=(char *)malloc(100);  
    //接下来,可以用 p 指向的内存了  
  
    p="hello world";//p 指向别的地方了  
  
    //从此以后,再也找不到你申请的 100 个字节了。则动态申请的 100 个字节就被泄露了  
  
    return 0;  
}
```

内存泄露 例 2:

```
void fun()  
{  
    char *p;  
    p=(char *)malloc(100);  
    //接下来,可以用 p 指向的内存了  
    ;  
    ;  
}  
  
int main()  
{  
    fun();  
    fun();  
    return 0;  
}
```

```
}  
//每调用一次 fun 泄露 100 个字节
```

内存泄露 解决方案 1:

```
void fun()  
{  
    char *p;  
    p=(char *)malloc(100);  
    //接下来·可以用 p 指向的内存了  
    ;  
    ;  
    free(p);  
}  
  
int main()  
{  
    fun();  
    fun();  
    return 0;  
}
```

内存泄露 解决方案 2:

```
char * fun()  
{  
    char *p;  
    p=(char *)malloc(100);  
    //接下来·可以用 p 指向的内存了  
    ;  
    ;  
    return p;  
}  
  
int main()  
{  
    char *q;  
    q=fun();  
    //可以通过 q 使用·动态申请的 100 个字节的内存了
```

```
//记得释放  
free(q);  
  
return 0;  
}
```

总结：申请的内存，一定不要把首地址给丢了，在不用的时候一定要释放内存。

第8章 字符串处理函数

8.1 测字符串长度函数

头文件：`#include <string.h>`

函数定义：`size_t strlen(const char *s);`

函数功能：

测字符串指针 `s` 指向的字符串中字符的个数，不包括 `'\0'`

返回值：字符串中字符个数

例 1：

```
#include <stdio.h>  
#include <string.h>  
int main()  
{  
    char str1[20]="hello";  
    char *str2 ="hello";  
    printf("%d\n",sizeof(str1)); //20  
    printf("%d\n",sizeof(str2)); //4  
    printf("%d\n",strlen(str1)); //5  
    printf("%d\n",strlen(str2)); //5  
    return 0;  
}
```

`sizeof` 是个关键字，测量数据的占用内存空间大小。

如果测量的是数组的名字，则测的是数组占多少个字节

如果 `sizeof` 测的是指针变量，则测的是指针变量本身占几个字节，32 平台下结果为 4

`strlen` 是个库函数，它测的是字符指针指向的字符串中字符的个数，不管指针是数组的名字，还是个指针变量。

8.2 字符串拷贝函数

头文件: #include <string.h>

函数的定义: char *strcpy(char *dest, const char *src);

函数的说明:

拷贝 src 指向的字符串到 dest 指针指向的内存中, '\0' 也会拷贝

函数的返回值:

目的内存的地址

注意: 在使用此函数的时候, 必须保证 dest 指向的内存空间足够大, 否则会出现内存污染。

char *strncpy(char *dest, const char *src, size_t n);

函数的说明:

将 src 指向的字符串前 n 个字节, 拷贝到 dest 指向的内存中

返回值: 目的内存的首地址

注意:

1、strncpy 不拷贝 '\0'

2、如果 n 大于 src 指向的字符串中的字符个数, 则在 dest 后面填充 n-strlen(src) 个 '\0'

例 2 :

```
#include<stdio.h>
#include<string.h>
int main()
{
    char buf[100]="aaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
    strncpy(buf,"helloworld",5);
    printf("%s\n",buf);
}
```

结果为 helloaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

验证了不拷贝 '\0'

例 3 :

```
#include<stdio.h>
#include<string.h>
int main()
{
    int len,i;
    char buf[100]="aaaaa";
    len=strlen(buf);
    strncpy(buf,"helloworld",15);

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

```
printf("%c",buf[i]);  
  
printf("\n");  
}
```

验证了：

如果 `n` 大于 `src` 指向的字符串中的字符个数，则在 `dest` 后面填充 `n-strlen(src)` 个 `'\0'`

3、字符串追加函数

头文件：`#include <string.h>`

函数定义：`char *strcat(char *dest, const char *src);`

函数功能：

`strcat` 函数追加 `src` 字符串到 `dest` 指向的字符串的后面。追加的时候会追加 `'\0'`

注意：保证 `dest` 指向的内存空间足够大。

例 4：

```
#include <stdio.h>  
#include <string.h>  
int main()  
{  
    char str[20]="aa\0aaaaaaaaaaaaaaaa";  
    char *src ="hello";  
    strcat(str,src);  
    printf("%s\n",str);  
    return 0;  
}
```

结果是 `aahello`

验证了追加字符串的时候追加 `'\0'`

`char *strncat(char *dest, const char *src, size_t n);`

追加 `src` 指向的字符串的前 `n` 个字符，到 `dest` 指向的字符串的后面。

注意如果 `n` 大于 `src` 的字符个数，则只将 `src` 字符串追加到 `dest` 指向的字符串的后面

追加的时候会追加 `'\0'`

例 5：

```
#include <stdio.h>  
#include <string.h>  
int main()  
{  
    char str[20]="aa\0aaaaaaaaaaaaaaaa";  
    char *src ="hello";  
    strncat(str,src,3);  
    printf("%s\n",str);  
}
```

```
return 0;  
}
```

结果为: aahel

验证了会追加'\0'

strcmp/strncmp //比较

头文件: #include <string.h>

函数定义: int strcmp(const char *s1, const char *s2);

函数说明:

比较 s1 和 s2 指向的字符串的大小,

比较的方法: 逐个字符去比较 ASCII 码, 一旦比较出大小返回。

如过所有字符都一样, 则返回 0

返回值:

如果 s1 指向的字符串大于 s2 指向的字符串 返回 1

如果 s1 指向的字符串小于 s2 指向的字符串 返回-1

如果相等的话返回 0

int strncmp(const char *s1, const char *s2, size_t n);

函数说明: 比较 s1 和 s2 指向的字符串中的前 n 个字符

例 6 :

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

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

```
int main()
```

```
{
```

```
    char *str1 = "hello world";
```

```
    char *str2 = "hello kitty";
```

```
    if( strcmp(str1,str2) == 0)
```

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

```
    else if(strcmp(str1,str2) > 0)
```

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

```
    else
```

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

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

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

```
    else if(strncmp(str1,str2,5) > 0)
```

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

```
    else
```

```
printf("str1<str2\n");  
  
return 0;  
}
```

字符查找函数

头文件: #include <string.h>

函数定义: char *strchr(const char *s, int c);

函数说明:

在字符指针 s 指向的字符串中, 找 ascii 码为 c 的字符

注意, 是首次匹配, 如果过说 s 指向的字符串中有多个 ASCII 为 c 的字符, 则找的是第 1 个字符

返回值:

找到了返回找到的字符的地址,

找不到返回 NULL,

函数定义: char *strrchr(const char *s, int c);

函数的说明: 末次匹配

在 s 指向的字符串中, 找最后一次出现的 ASCII 为 c 的字符,

返回值:

末次匹配的字符的地址。

字符串匹配函数

#include <string.h>

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

函数说明:

在 haystack 指向的字符串中查找 needle 指向的字符串, 也是首次匹配

返回值:

找到了: 找到的字符串的首地址

每找到: 返回 NULL

例 7 :

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

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

```
int main()
```

```
{
```

```
char str1[20]="jfsjdklsd43$#$53klj4t5";
```

```
char ch='$';
```

```
char str2[20]="$#$";
```

```
char *result;
```

```
result=strchr(str1,ch);
```

```
printf("%s\n",result);
printf("%d\n",result-str1);

result=strstr(str1,str2);
printf("%s\n",result);
printf("%d\n",result-str1);
return 0;
}
```

memset //空间设定函数

字符串转换数值

atoi/atol/atof //字符串转换功能

头文件: #include <stdlib.h>

函数的定义: int atoi(const char *nptr);

函数的功能:

将 nptr 指向的字符串转换成整数, 返回

例 8 :

```
int num;
```

```
num=atoi( "123" );
```

则 num 的值为 123

```
long atol(const char *nptr);
```

```
double atof(const char *nptr);
```

字符串切割函数:

头文件: #include <string.h>

函数定义: char *strtok(char *str, const char *delim);

函数的功能:

字符串切割, 按照 delim 指向的字符串中的字符, 切割 str 指向的字符串。
其实就是在 str 指向的字符串中发现了 delim 字符串中的字符, 就将其变成'\0',
调用一次 strtok 只切割一次, 切割一次之后, 再去切割的时候 strtok 的第一个参数
传 NULL, 意思是接着上次切割的位置继续切

注意如果 str 字符串中出现了连续的几个 delim 中的字符, 则只将第一个字符变成'\0'

例 9 :

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

```
int main()
```

```
{
```

```
char str[100]="小明:21,,,男女,北京:haidian";
```

```
char *p=":,.,";
```

```
char *q[7];
```

```
int i=0,j;
```

```
q[i]=strtok(str,p);
```

```
while(q[i]!=NULL)
```

```

{
    i++;
    q[i]=strtok(NULL,p);
}
for(j=0;j<i;j++)
{
    printf("q[%d]: %s\n",j,q[j]);
}
printf("str=%p\n",str);
printf("q[0]=%p\n",q[0]);
}
    
```

例 10: 作业

以下为我们的手机收到的短信的格式，请利用指针数组与 strtok 函数对其解析

```
char msg_src[]="+CMGR:REC UNREAD,+8613466630259,98/10/01,18:22:11+00,ABCdefGHI";
```

参考以下的函数名字以及参数，完成相应的要求

```
int msg_deal(char *msg_src, char *msg_done[], char *str)
```

参数 1: 待切割字符串的首地址

参数 2: 指针数组: 存放切割完字符串的首地址

参数 3: 切割字符

返回值: 切割的字符串总数量

手机号:13466630259

日期: 98/10/01

时间: 18:22:11

内容: ABCdefGHI

```

#include<stdio.h>
#include<string.h>
int msg_deal(char *msg_src,char *msg_done[],char *str)
{
    int i=0;
    msg_done[i]=strtok(msg_src,str);
    while(msg_done[i]!=NULL)
    {
        i++;
        msg_done[i]=strtok(NULL,str);
    }
    return i;
}
int main()
{
    
```

```
int num,j;
char buf[]="+CMGR:REC UNREAD,+8613466630259,98/10/01,18:22:11+00,ABCdefGHI";
char *p=",";
char *q[6];
char *temp;
num=msg_deal(buf,q,p);
q[1]=q[1]+3;
printf("手机号:%s\n",q[1]);
printf("日期:%s\n",q[2]);
temp=strchr(q[3],'+');
if(temp!=NULL)
    *temp='\0';
printf("时间:%s\n",q[3]);
printf("内容:%s\n",q[4]);
}
```

8.3 格式化字符串操作函数

int sprintf(char *buf, const char *format, ...);

\\输出到 buf 指定的内存区域。

例:

```
char buf[20];
sprintf(buf,"%d:%d:%d",2013,10,1);
printf("buf=%s\n",buf);
```

int sscanf(const char *buf, const char *format, ...);

\\从 buf 指定的内存区域中读入信息

例: int a, b, c;

```
sscanf("2013:10:1", "%d:%d:%d", &a, &b, &c);
printf("%d %d %d\n",a,b,c);
```

例 11 :

```
#include <stdio.h>
#include <string.h>
int main()
{
    char buf[20];
    int a, b, c;

    sprintf(buf,"%d:%d:%d",2013,10,1);
    printf("buf=%s\n",buf);//结果为 2013:10:1
```

```
scanf("2013:10:1", "%d:%d:%d", &a, &b, &c);  
printf("a=%d,b=%d,c=%d\n",a,b,c); //结果为 a=2013,b=10,c=1  
return 0;  
}
```

sscanf 高级用法

1、跳过数据：%*s 或%*d

例：scanf("1234 5678", "%*d %s", buf);

例 12：

```
#include <stdio.h>  
int main()  
{  
    char buf[20];  
    sscanf("1234 5678", "%*d %s", buf); //跳过 1234 ,然后隔一个空格获取字符串  
    printf("%s\n", buf);  
}
```

结果为 5678

2、读指定宽度的数据：%[width]s

例：scanf("12345678", "%4s", buf);

例 13：

```
#include <stdio.h>  
int main()  
{  
    char buf[20];  
    sscanf("12345678", "%4s ", buf); //从字符串中获取字符串，只要 4 个字节，存放在 buf 中  
    printf("%s\n", buf);  
}
```

3、支持集合操作：只支持获取字符串

%[a-z] 表示匹配 a 到 z 中任意字符(尽可能多的匹配)

例 14：

```
#include <stdio.h>  
int main()  
{  
    char buf[20];  
    sscanf("agcd32DajfDdFF", "%[a-z]", buf); //从字符串中获取输入只要 'a' 和 'z' 之间的字符，碰到不在范围内的，就终止了  
    printf("%s\n", buf); //结果为 agcd
```

}

`%[aBc]` 匹配 a、B、c 中一员，贪婪性

`%[^aFc]` 匹配非 aFc 的任意字符，贪婪性

`%[^a-z]` 表示读取除 a-z 以外的所有字符

例 15: 练习:

使用 `sscanf` 两个#号之间的字符串 `abc#def@ghi`

```
#include <stdio.h>
int main()
{
    char buf[20];
    sscanf("asdf#sjjd@djkfd", "%*[^#]*c%[^@]", buf);
    printf("%s\n", buf);
}
```

8.4 const:

1: 修饰普通变量, 代表只读的意思

`const int a=100;` // 定义了一个只读变量 a 值为 100

以后在程序中, 不能再给 a 赋值了

`a=200;` // 错误的, a 只读

2: `const` 修饰指针

(1)、`const char *str`

意思是 `str` 指向的内存的内容不能通过 `str` 来修改

用来保护 `str` 指向的内存的内容

但是 `str` 的指向是可以改变的

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

(2)、`char * const str`

意思是 `str` 是只读的变量, `str` 不能指向别的地方,

但是 `str` 指向的内存的内容, 是有可能可以修改的

(3)、`const char * const str`

str 不能指向别的地方，指向的内存的内容也不能通过 str 去修改

第9章 结构体

前面学过一种构造类型——数组：

构造类型：

不是基本类型的数据结构也不是指针，它是若干个相同或不同类型的数据构成的集合

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，

以便于引用。如：

一个学生有学号/姓名/性别/年龄/地址等属性

```
int num;
```

```
char name[20];
```

```
char sex;
```

```
int age;
```

```
char addr[30];
```

显然单独定义以上变量比较繁琐，数据不便于管理

9.1 结构体类型的概念及定义

1、概念：

结构体是一种构造类型的数据结构，
是一种或多种基本类型或构造类型的数据的集合。

2、结构体类型的定义方法

咱们在使用结构体之前必须先有类型，然后用类型定义数据结构
这个类型相当于一个模具

(1).先定义结构体类型，再去定义结构体变量

```
struct 结构体类型名{  
    成员列表  
};
```

例 1：

```
struct stu{  
    int num;  
    char name[20];  
    char sex;  
};
```

//有了结构体类型后，就可以用类型定义变量了

```
struct stu lucy,bob,lilei;//定义了三个 struct stu 类型的变量
```

每个变量都有三个成员，分别是 num name sex

咱们可以暂时认为结构体变量的大小是它所有成员之和

(2).在定义结构体类型的时候顺便定义结构体变量，以后还可以定义结构体变量

```
struct 结构体类型名{  
    成员列表;  
}结构体变量 1,变量 2;
```

```
struct 结构体类型名 变量 3, 变量 4;
```

例 2 :

```
struct stu{  
    int num;  
    char name[20];  
    char sex;  
}lucy,bob,lilei;  
  
struct stu xiaohong,xiaoming;
```

3.在定义结构体类型的时候，没有结构体类型名，顺便定义结构体变量，因为没有类型名，所以以后不能再定义相关类型的数据了

```
struct {  
    成员列表;  
}变量 1, 变量 2;
```

例 3 :

```
struct {  
    int num;  
    char name[20];  
    char sex;  
}lucy,bob;
```

以后没法再定义这个结构体类型的数据了，因为没有类型名

4.最常用的方法

通常咱们将一个结构体类型重新起个类型名，用新的类型名替代原先的类型

步骤 1: 先用结构体类型定义变量

```
struct stu{
    int num;
    char name[20];
    char sex;
}bob;
```

步骤 2: 新的类型名替代变量名

```
struct stu{
    int num;
    char name[20];
    char sex;
}STU;
```

步骤 3: 在最前面加 typedef

```
typedef struct stu{
    int num;
    char name[20];
    char sex;
}STU;
```

注意: 步骤 1 和步骤 2, 在草稿上做的, 步骤 3 是程序中咱们想要的代码

以后 STU 就相当于 struct stu
STU lucy;和 struct stu lucy;是等价的。

9.2 结构体变量的定义初始化及使用

1、结构体变量的定义和初始化

结构体变量, 是个变量, 这个变量是若干个数据的集合

注:

- (1):在定义结构体变量之前首先得有结构体类型, 然后在定义变量
- (2):在定义结构体变量的时候, 可以顺便给结构体变量赋初值, 被称为结构体的初始化
- (3):结构体变量初始化的时候, 各个成员顺序初始化

例 4:

```
struct stu{
    int num;
    char name[20];
    char sex;
```

```
};  
struct stu boy;  
struct stu lucy={  
    101,  
    "lucy",  
    'f'  
};
```

2、结构体变量的使用

定义了结构体变量后，要使用变量

(1).结构体变量成员的引用方法

结构体变量.成员名

例 5 :

```
struct stu{  
    int num;  
    char name[20];  
    char sex;  
};  
struct stu bob;
```

bob.num=101;//bob 是个结构体变量，但是 bob.num 是个 int 类型的变量

bob.name 是个字符数组，是个字符数组的名字，代表字符数组的地址，是个常量

bob.name ="bob";//是不可行，是个常量

```
strcpy(bob.name,"bob");
```

例 6 :

```
#include <stdio.h>  
struct stu{  
    int num;  
    char name[20];  
    int score;  
    char *addr;  
};  
int main(int argc, char *argv[])  
{  
    struct stu bob;  
    printf("%d\n",sizeof(bob));
```

```
printf("%d\n",sizeof(bob.name));
printf("%d\n",sizeof(bob.addr));
return 0;
}
```

(2).结构体成员多级引用

例 7 :

```
#include <stdio.h>
struct date{
    int year;
    int month;
    int day;
};
struct stu{
    int num;
    char name[20];
    char sex;
    struct date birthday;
};
int main(int argc, char *argv[])
{
    struct stu lilei={101,"lilei",'m'};
    lilei.birthday.year=1986;
    lilei.birthday.month=1;
    lilei.birthday.day=8;
    printf("%d %s %c\n",lilei.num,lilei.name,lilei.sex);
    printf("%d %d %d\n",lilei.birthday.year,lilei.birthday.month,lilei.birthday.day);
    return 0;
}
```

3、相同类型的结构体变量可以相互赋值

注意：必须是相同类型的结构体变量，才能相互赋值。

例 8 :

```
#include <stdio.h>
struct stu{
    int num;
    char name[20];
    char sex;
};
int main(int argc, char *argv[])
```

```

{
    struct stu bob={101,"bob",'m'};
    struct stu lilei;

    lilei=bob;
    printf("%d %s %c\n",lilei.num,lilei.name,lilei.sex);
    return 0;
}
    
```

9.3 结构体数组

结构体数组是个数组，由若干个相同类型的结构体变量构成的集合

1、结构体数组的定义方法

struct 结构体类型名 数组名[元素个数];

例 9 :

```

struct stu{
    int num;
    char name[20];
    char sex;
};
    
```

struct stu edu[3];//定义了一个 struct stu 类型的结构体数组 edu，
这个数组有 3 个元素分别是 edu[0]、edu[1]、edu[2]

1、结构体数组元素的引用 数组名[下标]

2、数组元素的使用

```

edu[0].num =101;//用 101 给 edu 数组的第 0 个结构体变量的 num 赋值
strcpy(edu[1].name,"lucy");
    
```

例 10 :

```

#include <stdio.h>
typedef struct student
{
    int num;
    char name[20];
    float score;
}STU;
STU edu[3]={
    {101,"Lucy",78},
    {102,"Bob",59.5},
    
```

```
{103,"Tom",85}
};
int main()
{
    int i;
    float sum=0;
    for(i=0;i<3;i++)
    {
        sum+=edu[i].score;
    }
    printf("平均成绩为%f\n",sum/3);
    return 0;
}
```

9.4 结构体指针

即结构体的地址，结构体变量存放在内存中，也有起始地址
咱们定义一个变量来存放这个地址，那这个变量就是结构体指针变量。

1、结构体指针变量的定义方法：

```
struct 结构体类型名 * 结构体指针变量名;
```

```
struct stu{
    int num;
    char name[20];
};
```

struct stu * p;//定义了一个 struct stu *类型的指针变量
变量名 是 p，p 占 4 个字节，用来保存结构体变量的地址编号

```
struct stu boy;
p=&boy;
访问结构体变量的成员方法：
```

例 11：

```
boy.num=101;//可以 · 通过 结构体变量名.成员名
(*p).num=101;//可以 · *p 相当于 p 指向的变量 boy
p->num=101;//可以 · 指针->成员名
```

通过结构体指针来引用指针指向的结构体的成员，前提是
指针必须先指向一个结构体变量。

结构体指针经常用到的地方：**(1)：保存结构体变量的地址**

例 12：

```
typedef struct stu{
    int num;
    char name[20];
    float score;
}STU;
int main()
{
    STU *p,lucy;
    p=&lucy;
    p->num=101;
    strcpy(p->name,"baby");
    //p->name="baby";//错误·因为 p->name 相当于 lucy.name 是个字符数组的名字·是个常量
}
```

(2)：传 结构体变量的地址

例 13：

```
#include <stdio.h>
#include <string.h>
typedef struct stu{
    int num;
    char name[20];
    float score;
}STU;
void fun(STU *p)
{
    p->num=101;
    (*p).score=87.6;
    strcpy(p->name,"lucy");
}
int main()
{
    STU girl;
    fun(&girl);
    printf("%d %s %f\n",girl.num,girl.name,girl.score);
    return 0;
}
```

(3): 传结构体数组的地址

结构体数组，是由多个相同类型的结构体变量构成的。存放在内存里，也有起始地址，其实就是第 0 个结构体的地址。

例 14 :

```
#include <stdio.h>
#include <string.h>
typedef struct stu{
    int num;
    char name[20];
    float score;
}STU;
void fun(STU *p)
{
    p[1].num=101;
    (*(p+1)).score=88.6;
}
int main()
{
    STU edu[3];
    fun(edu);

    printf("%d %f\n",edu[1].num,edu[1].score);
    return 0;
}
```

注意:

(1): 结构体变量的地址编号和结构体第一个成员的地址编号相同，但指针的类型不同

例 15 :

```
#include <stdio.h>
struct stu{
    int num;
    char name[20];
    int score;
};
int main(int argc, char *argv[])
{
    struct stu bob;
    printf("%p\n",&bob);
}
```

```
printf("%p\n",&(bob.num));  
return 0;  
}
```

(2): 结构体数组的地址就是结构体数组中第 0 个元素的地址

例 16 :

```
#include <stdio.h>  
struct stu{  
    int num;  
    char name[20];  
    int score;  
};  
int main(int argc, char *argv[])  
{  
    struct stu edu[3];  
    printf("%p\n",edu);//struct stu *  
    printf("%p\n",&(edu[0]));//struct stu *  
    printf("%p\n",&(edu[0].num));//int *  
  
    return 0;  
}
```

9.5 结构体内存分配

1、结构体内存分配

结构体变量大小是，它所有成员之和。

因为结构体变量是所有成员的集合。

例 17 :

```
#include<stdio.h>  
struct stu{  
    int num;  
    int age;  
}lucy;  
int main()  
{  
    printf("%d\n",sizeof(lucy));//结果为 8  
    return 0;  
}
```

但是在实际给结构体变量分配内存的时候，是规则的

例 18 :

```
#include <stdio.h>
struct stu{
    char sex;
    int age;
}lucy;
int main()
{
    printf("%d\n",sizeof(lucy));//结果为 8? ? ?
    return 0;
}
```

规则 1: 以多少个字节为单位开辟内存

给结构体变量分配内存的时候, 会去结构体变量中找基本类型的成员
哪个基本类型的成员占字节数多, 就以它大小为单位开辟内存,
在 gcc 中出现了 double 类型的, 例外

- (1): 成员中只有 char 型数据, 以 1 字节为单位开辟内存。
- (2): 成员中出现了 short int 类型数据, 没有更大字节数的基本类型数据。
以 2 字节为单位开辟内存
- (3): 出现了 int float 没有更大字节的基本类型数据的时候以 4 字节为单位开辟内存。
- (4): 出现了 double 类型的数据

情况 1:

在 vc 里, 以 8 字节为单位开辟内存。

情况 2:

在 gcc 里, 以 4 字节为单位开辟内存。

无论是那种环境, double 型变量, 占 8 字节。

注意:

n 如果在结构体中出现了数组, 数组可以看成多个变量的集合。
如果出现指针的话, 没有占字节数更大的类型的, 以 4 字节为单位开辟内存。

在内存中存储结构体成员的时候, 按定义的结构体成员的顺序存储。

```
例 19 : struct stu{
    char sex;
    int age;
}lucy;
lucy 的大小是 4 的倍数。
```

规则 2: 字节对齐

- (1): char 1 字节对齐, 即存放 char 型的变量, 内存单元的编号是 1 的倍数即可。
- (2): short int 2 字节对齐, 即存放 short int 型的变量, 起始内存单元的编号是 2 的倍数即可。

(3): int 4 字节对齐，即存放 int 型的变量，起始内存单元的编号是 4 的倍数即可

(4): long int 在 32 位平台下，4 字节对齐，即存放 long int 型的变量，起始内存单元的编号是 4 的倍数即可

(5): float 4 字节对齐，即存放 float 型的变量，起始内存单元的编号是 4 的倍数即可

(6): double

a.vc 环境下

8 字节对齐，即存放 double 型变量的起始地址，必须是 8 的倍数，double 变量占 8 字节

b.gcc 环境下

4 字节对齐，即存放 double 型变量的起始地址，必须是 4 的倍数，double 变量占 8 字节。

注意 3: 当结构体成员中出现数组的时候，可以看成多个变量。

注意 4: 开辟内存的时候，从上向下依次按成员在结构体中的位置顺序开辟空间

例 20: //temp 8 个字节

```
#include<stdio.h>
struct stu{
    char a;
    short int b;
    int c;
}temp;
int main()
{
    printf("%d\n",sizeof(temp));
    printf("%p\n",&(temp.a));
    printf("%p\n",&(temp.b));
    printf("%p\n",&(temp.c));
    return 0;
}
```

结果分析:

a 的地址和 b 的地址差 2 个字节

b 的地址和 c 的地址差 2 个字节

例 21: temp 的大小为 12 个字节

```
#include<stdio.h>
struct stu{
    char a;
    int c;
    short int b;
}temp;
int main()
{
    printf("%d\n",sizeof(temp));
}
```

```
printf("%p\n",&(temp.a));  
printf("%p\n",&(temp.b));  
printf("%p\n",&(temp.c));  
return 0;  
}
```

结果分析:

a 和 c 的地址差 4 个字节

c 和 b 的地址差 4 个字节

例 22 :

```
struct stu{  
    char buf[10];  
    int a;  
}temp;  
//temp 占 16 个字节
```

例 23 :

在 vc 中占 16 个字节 a 和 b 的地址差 8 个字节

在 gcc 中占 12 个字节 a 和 b 的地址差 4 个字节

```
#include <stdio.h>  
struct stu{  
    char a;  
    double b;  
}temp;  
int main()  
{  
    printf("%d\n",sizeof(temp));  
    printf("%p\n",&(temp.a));  
    printf("%p\n",&(temp.b));  
    return 0;  
}
```

为什么要有字节对齐?

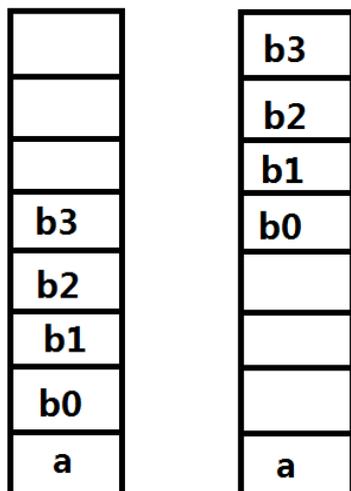
用空间来换时间, 提高 cpu 读取数据的效率

struct str{

char a;

int b;

}



指定对齐原则:

使用#pragma pack改变默认对其原则

格式:

#pragma pack (value)时的指定对齐值value。

注意:

1.value只能是: 1 2 4 8等

2.指定对齐值与数据类型对齐值相比取较小值

说明: 咱们制定一个value

(1): 以多少个字节为单位开辟内存

结构体成员中, 占字节数最大的类型长度和value比较, 取较小值, 为单位开辟内存

例 24 :

```
#pragma pack(2)
struct stu{
    char a;
    int b;
};
```

以2个字节为单位开辟内存

```
#include<stdio.h>
#pragma pack(2)
struct stu{
    char a;
    int b;
}temp;
int main()
{
    printf("%d\n",sizeof(temp));
```

```
printf("%p\n",&(temp.a));  
printf("%p\n",&(temp.b));  
return 0;  
}
```

temp的大小为6个字节

a和b的地址差2个字节

例 25 :

```
#pragma pack(8)  
struct stu{  
    char a;  
    int b;  
};
```

以4个字节为单位开辟内存

```
#include<stdio.h>  
#pragma pack(8)  
struct stu{  
    char a;  
    int b;  
}temp;  
int main()  
{  
    printf("%d\n",sizeof(temp));  
    printf("%p\n",&(temp.a));  
    printf("%p\n",&(temp.b));  
    return 0;  
}
```

temp的大小为8个字节

a和b的地址差4个字节

(2): 字节对齐

结构体成员中成员的对齐方法，各个默认的对齐字节数和value相比，取较小值

例 26 :

```
#include<stdio.h>  
#pragma pack(2)  
struct stu{  
    char a;  
    int b;  
}temp;  
int main()
```

```
{
    printf("%d\n",sizeof(temp));
    printf("%p\n",&(temp.a));
    printf("%p\n",&(temp.b));
    return 0;
}
```

b成员是2字节对齐，a和b的地址差2个字节

例 27 :

```
#include<stdio.h>
#pragma pack(8)
struct stu{
    char a;
    int b;
}temp;
int main()
{
    printf("%d\n",sizeof(temp));
    printf("%p\n",&(temp.a));
    printf("%p\n",&(temp.b));
    return 0;
}
```

a和b都按原先的对齐方式存储

如：如果指定对齐值：

设为1：则short、int、float等均为1

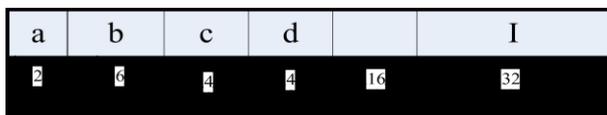
设为2：则 char 仍为 1，short 为 2，int 变为 2

9.6 位段

一、位段

在结构体中，以位为单位的成员，咱们称之为位段(位域)。

```
struct packed_data{
    unsigned int a:2;
    unsigned int b:6;
    unsigned int c:4;
    unsigned int d:4;
    unsigned int i;
} data;
```



注意：不能对位段成员取地址

例 28 :

```
#include <stdio.h>
struct packed_data{
unsigned int a:2;
unsigned int b:6;
unsigned int c:4;
unsigned int d:4;
unsigned int i;
} data;
int main()
{
    printf("%d\n",sizeof(data));
    printf("%p\n",&data);
    printf("%p\n",&(data.i));
    return 0;
}
```

位段注意:

1、对于位段成员的引用如下:

data.a =2

赋值时，不要超出位段定义的范围;

如段成员a定义为2位，最大值为3,即 (11) 2

所以data.a =5，就会取5的低两位进行赋值 101

2、位段成员的类型必须指定为整形或字符型

3、一个位段必须存放在一个存储单元中，不能跨两个单元

第一个单元空间不能容纳下一个位段，则该空间不用，

而从下一个单元起存放该位段

位段的存储单元:

(1): char 型位段 存储单元是 1 个字节

(2): short int 型的位段存储单元是 2 个字节

(3): int 的位段，存储单元是 4 字节

(4): long int 的位段，存储单元是 4 字节

```
struct stu{
    char a:7;
    char b:7;
    char c:2;
}temp;//占 3 字节，b 不能跨 存储单元存储
```

例 29 :

```
#include<stdio.h>
struct stu{
    char a:7;
    char b:7;
    char c:2;
}temp;
int main()
{
    printf("%d\n",sizeof(temp));
    return 0;
}
```

结果为：3，证明位段不能跨其存储单元存储

注意：不能取 temp.b 的地址，因为 b 可能不够 1 字节，不能取地址。

4、位段的长度不能大于存储单元的长度

- (1): char 型位段不能大于 8 位
- (2): short int 型位段不能大于 16 位
- (3): int 的位段，位段不能大于 32 位
- (4): long int 的位段，位段不能大于 32 位

例 30：

```
#include<stdio.h>
struct stu{
    char a:9;
    char b:7;
    char c:2;
}temp;
int main()
{
    printf("%d\n",sizeof(temp));
    return 0;
}
```

分析：

编译出错，位段 a 不能大于其存储单元的大小

5、如一个段要从另一个存储单元开始，可以定义：

```
unsigned char a:1;
unsigned char b:2;
unsigned char :0;
unsigned char c:3;(另一个单元)
```

由于用了长度为 0 的位段，其作用是使下一个位段从

下一个存储单元开始存放

将 a、b 存储在一个存储单元中，c 另存在下一个单元

例：31

```
#include<stdio.h>
struct m_type{
    unsigned char a:1;
    unsigned char b:2;
    unsigned char :0;
    unsigned char c:3;
};
int main()
{
    struct m_type temp;
    printf("%d\n",sizeof(temp));
    return 0;
}
```

6、可以定义无意义位段,如:

unsigned a: 1;

unsigned : 2;

unsigned b: 3;

例 32：位段的应用

```
struct data{
    char a:1;
    char b:1;
    char c:1;
    char d:1;
    char e:1;
    char f:1;
    char g:1;
    char h:1;
}temp;
int main()
{
    char p0;
    //p0=0x01;// 0000 0001
    temp.a=1;
    //p0=temp;//错的，类型不匹配
    //p0=(char)temp;//错的，编译器不允许将结构体变量，强制转成基本类型的。
```

```
p0= *((char *)&temp);  
}
```

二：共用体

1：共用体和结构体类似，也是一种构造类型的数据结构。

既然是构造类型的，咱们得先定义出类型，然后用类型定义变量。

定义共用体类型的方法和结构体非常相似，把 struct 改成 union 就可以了。

在进行某些算法的时候，需要使几种不同类型的变量存到同一段内存单元中，几个变量所使用空间相互重叠

这种几个不同的变量共同占用一段内存的结构，在C语言中，被称作“共用体”类型结构

共用体所有成员占有同一段地址空间

共用体的大小是其占内存长度最大的成员的大小

例 33：

```
typedef struct data{  
    short int i;  
    char ch;  
    float f ;  
}DATA;  
DATA temp1;
```

结构体变量 temp1 最小占 7 个字节（不考虑字节对齐）

例 34：

```
typedef union data{  
    short int i;  
    char ch;  
    float f ;  
}DATA;  
DATA temp2;
```

共用体 temp2 占 4 个字节，即 i、ch、f 共用 4 个字节

```
#include <stdio.h>  
typedef union data{  
short int i;  
char ch;  
float f;  
}DATA;  
int main()  
{  
    DATA temp2;
```

```
printf("%d\n",sizeof(temp2));  
printf("%p\n",&temp2);  
printf("%p\n",&(temp2.i));  
printf("%p\n",&(temp2.ch));  
printf("%p\n",&(temp2.f));  
return 0;  
}
```

结果：`temp2` 的大小为 4 个字节，下面几个地址都是相同的，证明了共用体的各个成员占用同一块内存。

共用体的特点：

- 1、同一内存段可以用来存放几种不同类型的成员，但每一瞬时只有一种起作用
- 2、共用体变量中起作用的成员是最后一次存放的成员，在存入一个新的成员后原有的成员的值会被覆盖
- 3、共用体变量的地址和它的各成员的地址都是同一地址
- 4、共用体变量的初始化

`union data a={ 123};` 初始化共用体为第一个成员

例 35 :

```
#include <stdio.h>  
typedef union data{  
    unsigned char a;  
    unsigned int b;  
}DATA;  
int main()  
{  
    DATA temp;  
    temp.b=0xffffffff;  
    printf("temp.b = %x\n",temp.b);  
    temp.a=0x0d;  
    printf("temp.a= %x\n",temp.a);  
    printf("temp.b= %x\n",temp.b);  
    return 0;  
}
```

结果：

`temp.b = ffffffff`

`temp.a= d`

`temp.b= ffffff0d`

例 36:

扩展一下：

```
struct type{
```

```
char a:1;
char b:1;
char c:1;
char d:1;
char e:1;
char f:1;
char g:1;
char h:1;
};
union data{
    struct type temp;
    char p0;
}m;
int main()
{
    m.temp.a=1;
}
```

三：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

枚举类型也是个构造类型的

既然是构造类型的数据类型，就得先定义类型，再定义变量

1、枚举类型的定义方法

```
enum 枚举类型名{
    枚举值列表;
};
```

在枚举值表中应列出所有可用值,也称为枚举元素

枚举变量仅能取枚举值所列元素

2、枚举变量的定义方法

```
enum 枚举类型名 枚举变量名;
```

例 37 :

定义枚举类型 week

```
enum week //枚举类型
```

```
{
    mon · tue · wed · thu · fri · sat,sun
};
```

```
enum week workday,weekday;//枚举变量  
workday 与 weekday 只能取 sun....sat 中的一个
```

```
workday = mon; //正确  
weekday = tue; //正确  
workday = abc; //错误, 枚举值中没有 abc
```

① 枚举值是常量,不能在程序中用赋值语句再对它赋值

例如: sun=5; mon=2; sun=mon; 都是错误的.

② 枚举元素本身由系统定义了一个表示序号的数值

默认是从0开始顺序定义为0, 1, 2...

如在week中, mon值为0, tue值为1, ...,sun值为6

③ 可以改变枚举值的默认值: 如

```
enum week //枚举类型
```

```
{  
mon=3, tue, wed, thu, fri=4, sat,sun  
};
```

mon=3 tue=4,以此类推

fri=4 以此类推

注意: 在定义枚举类型的时候枚举元素可以用等号给它赋值, 用来代表元素从几开始编号
在程序中, 不能再次对枚举元素赋值, 因为枚举元素是常量。

第10章 链表

10.1 链表的概念

问题?

假如: 做一个班级信息管理系统, 统计班级学生的信息

而我们事先不知道班级人数, 或者知道人数, 但是中间人员可能发生变化: 比如有新同学加入, 有同学请假, 又或者我们需要统计班级的平均成绩等等

假如: 要做一个类似 QQ、飞秋类似的通信软件, 其中有一个功能, 类似用户上下线检测: 有新的用户上线、下线实时更新显示, 可以实时查询在线状态、按姓名排序等

以上问题如何使用学过的 C 语言知识处理呢?

使用数组远远不能达到我们的要求

因为数组必须实现确定大小, 不能实现动态申请、释放

使用 malloc 动态内存分配也无法实现

malloc 申请的空间, 不能实现局部申请、释放

这里我们学习一种很强大也很重要的数据结构——链表

➤ 定义:

链表是一种物理存储上非连续,数据元素的逻辑顺序通过链表中的指针链接次序,实现的一种线性存储结构。

➤ 特点:

链表由一系列节点(链表中每一个元素称为节点)组成,节点在运行时动态生成(malloc),每个节点包括两个部分:

一个是存储数据元素的数据域

另一个是存储下一个节点地址的指针域



链表的构成:

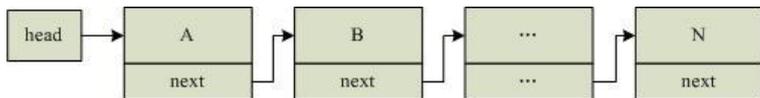
➤ 链表由一个个节点构成,每个节点一般采用结构体的形式组织,例如:

```
typedef struct student  
{  
    int num;  
    char name[20];  
    struct student *next;  
}STU;
```

➤ 链表节点分为两个域

数据域: 存放各种实际的数据,如: num、score 等

指针域: 存放下一节点的首地址,如: next 等.



10.2 链表的创建

10.3 链表的遍历

10.4 链表的释放

10.5 链表节点的查找

10.6 链表节点的删除

10.7 链表中插入一个节点

10.8 链表排序

10.9 链表逆序

10.10 双向链表

10.10.1 双向链表的创建

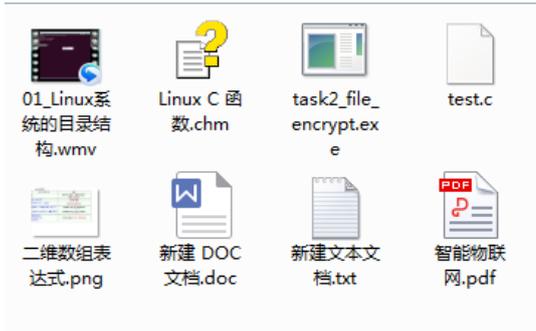
10.10.2 双向链表节点的删除

10.10.3 双向链表插入节点

第11章 文件

11.1 文件的概念

凡是使用过文件的人对文件都不会感到陌生



文件用来存放程序、文档、音频、视频数据、图片等数据的。
文件就是存放在磁盘上的，一些数据的集合。

在 windows 下可以通过写字板或记事本打开文本文件对文件进行编辑保存。写字板和记事本是微软程序员写的程序，对文件进行打开、显示、读写、关闭。

作为一个程序员，必须掌握编程实现创建、写入、读取文件等操作

对文件的操作是经常要用到的知识，比如：写飞秋软件传送文件 等

11.1.1 文件的定义：

磁盘文件：（我们通常认识的文件）

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。

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。

键盘：标准输入文件 屏幕：标准输出文件

其它设备：打印机、触摸屏、摄像头、音箱等

在 Linux 操作系统中，每一个外部设备都在/dev 目录下对应着一个设备文件，咱们在程序中要想操作设备，就必须对与其对应的/dev 下的设备文件进行操作。

标准 io 库函数对磁盘文件的读取特点



文件缓冲区是库函数申请的一段内存，由库函数对其进行操作，程序员没有必要知道存放在哪里，只需要知道对文件操作的时候的一些缓冲特点即可。

举例：放苹果到办公室的例子

1、行缓冲

标准 io 库函数，往**标准输出**（屏幕）输出东西的时候是行缓冲的

所谓的行缓冲就是缓冲区碰到换行符的时候才刷新缓冲区

刷新缓冲区的情况

1) 缓冲区里有换行符

例 1:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("hello world");
    while(1);
    return 0;
}
```

结果分析: `printf` 是标准 `io` 库函数, 往标准输出输出东西, 是行缓冲的, 因为输出的东西没有 `'\n'` 所以数据存放在缓冲区中, 没有刷新缓冲区。故不能输出到屏幕上

例 2:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("hello world\n");
    while(1);
    return 0;
}
```

结果分析: `hello world` 打印到屏幕上, 因为缓冲区中出现了 `'\n'`, 刷新了缓冲区

2) 缓冲区满了, 刷新缓冲区

例 3:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    while(1)
    {
        printf("hello world ");
    }
    return 0;
}
```

`Printf` 输出的内容虽然没有 `'\n'`, 但此程序也能将数据刷新到屏幕上, 原因是缓冲满了, 会自动刷新缓冲区。

3) 人为刷新缓冲区 `fflush(stdout)`

例 4:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
```

```
printf("hello world");  
fflush(stdout);//人为刷新缓冲区  
while(1);  
return 0;  
}
```

结果分析：hello world 会打印到屏幕上，原因是 fflush(stdout),是人为的刷新缓冲区

4) 程序正常结束 会刷新缓冲区

例 5:

```
#include <stdio.h>  
int main(int argc, char *argv[])  
{  
    printf("hello world");  
    return 0;  
}
```

2、全缓冲

标准 io 库函数，往普通文件读写数据的，是全缓冲的，碰到换行符也不刷新缓冲区，即缓冲区满了，才刷新缓冲区

刷新缓冲区的情况

- 1.缓冲区满了，刷新缓冲区
- 2.人为刷新缓冲区 fflush(文件指针)
- 3.程序正常结束 会刷新缓冲区

3.无缓冲

在读写文件的时候通过系统调用 io (read write),对文件进行读写数据这个时候是无缓冲的，即写数据会立马进入文件，读数据会立马进入内存

写文件的流程

应用程序空间->内核空间 ->驱动程序-->硬盘上

应用程序和内核程序运行在不同的空间里，目的是为了保护内核。

设置缓冲区的目的

通过缓冲可以减少进出内核的次数，以提高效率。

11.1.2 磁盘文件的分类:

一个文件通常是磁盘上一段命名的存储区

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

文本文件

基于字符编码，常见编码有 ASCII、UNICODE 等

一般可以使用文本编辑器直接打开

例如：数 5678 的以 ASCII 存储形式为：

ASCII 码：00110101 00110110 00110111 00111000

歌词文件(lrc):文本文件

二进制码文件：

基于值编码,自己根据具体应用,指定某个值是什么意思

把内存中的数据按其内存中的存储形式原样输出到磁盘上

一般需要自己判断或使用特定软件分析数据格式

例如：数 5678 的存储形式为：

二进制码：00010110 00101110

音频文件(mp3):二进制文件

图片文件 (bmp) 文件，一个像素点由两个字节来描述*****#####&&&&&

*代表红色的值

#代表绿色的值

&代表蓝色的值

二进制文件以位来表示一个意思。

文本文件、二进制文件对比：

译码：

文本文件编码基于字符定长，译码容易些；

二进制文件编码是变长的，译码难一些（不同的二进制文件格式，有不同的译码方式）。

空间利用率：

二进制文件用一个比特来代表一个意思(位操作)；

而文本文件任何一个意思至少是一个字符。

二进制文件，空间利用率高。

可读性：

文本文件用通用的记事本工具就几乎可以浏览所有文本文件

二进制文件需要一个具体的文件解码器，比如读 BMP 文件，必须用读图软件。

总结一下：

文件在硬盘上存储的时候，物理上都是用二进制来存储的。

咱们的标准 io 库函数，对文件操作的时候，不管文件的编码格式（字符编码、或二进制），而是按字节对文件

进行读写，所以咱们管文件又叫流式文件，即把文件看成一个字节流。

11.2 文件指针

文件指针在程序中用来标示（代表）一个文件的，在打开文件的时候得到文件指针，文件指针就用来代表咱们打开的文件。

咱们对文件进行读、写、关闭等操作的时候，对文件指针进行操作即可，即咱们将文件指针，传给读、写、关闭等函数，那些函数就知道要对哪个文件进行操作。

定义文件指针的一般形式为：

FILE * 指针变量标识符；

FILE 为大写，需要包含<stdio.h>

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

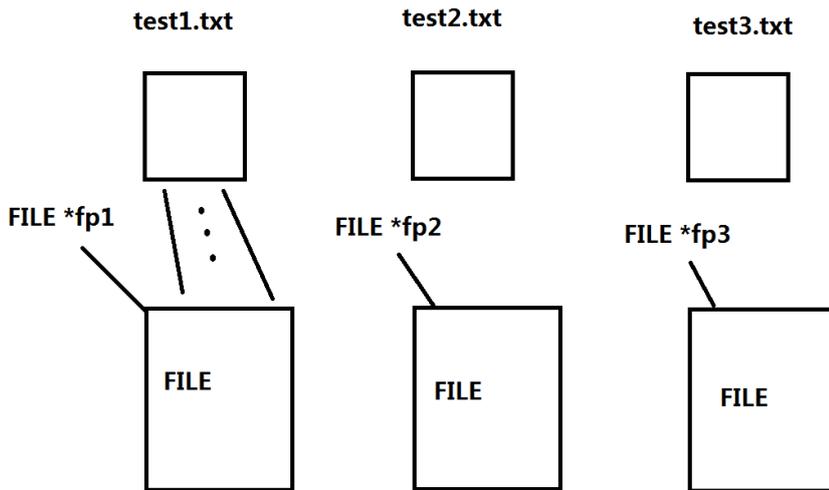
一般情况下，我们操作文件前必须定义一个文件指针标示 我们将要操作的文件

实际编程中使用库函数操作文件，无需关心 FILE 结构体的细节，只需要将文件指针传给 io 库函数，库函数再
通过 FILE 结构体里的信息对文件进行操作

FILE 在 stdio.h 文件中的文件类型声明：

```
typedef struct
{
    short level;           //缓冲区“满”或“空”的程度
    unsigned flags;       //文件状态标志
    char fd;              //文件描述符
    unsigned charhold;    //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*curp;     //指针，当前的指向
    unsigned istemp;      //临时文件，指示器
    shorttoken;          //用于有效性检查
} FILE;
```

在缓冲文件系统中,每个被使用的文件都要在内存中开辟一块 FILE 类型的区域,存放与操作文件相关的信息



对文件操作的步骤:

- 1、对文件进行读写等操作之前要打开文件得到文件指针
- 2、可以通过文件指针对文件进行读写等操作
- 3、读写等操作完毕后，要关闭文件，关闭文件后，就不能再通过此文件指针操作文件了

补充:

c 语言中有三个特殊的文件指针无需定义，在程序中可以直接使用

stdin: 标准输入 默认为当前终端（键盘）

我们使用的 `scanf`、`getchar` 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: 标准错误输出设备文件 默认为当前终端（屏幕）

当我们程序出错使用 `perror` 函数时信息打印在此终端

11.3 打开文件 `fopen`

函数的定义:

```
FILE *fopen(const char *path, const char *mode);
```

函数说明:

`fopen` 函数的功能是打开一个已经存在的文件，并返回这个文件的文件指针（文件的标示）或者创建一个文件，并打开此文件，然后返回文件的标示。

函数的参数:

参数 1:打开的文件的路径

1. 绝对路径,从根目录开始的路径名称

“/home/edu/test/test.txt”

2. 相对路径

“./test/test.txt”

“../test/test.txt”

参数 2: 文件打开的方式，即以什么样的方式（只读、只写、可读可写等等）打开文件

第二个参数的几种形式（打开文件的方式）

读写权限：r w a +

- r:以只读方式打开文件
文件**不存在**返回 NULL;
文件**存在**返回文件指针，进行后续的读操作

例 1:

```
FILE *fp;
fp=fopen("./test.txt","r");
```

- w:以只写方式打开文件
文件**不存在**，以指定文件名创建此文件，并且打开文件；
若文件**存在**，**清空文件内容**，打开文件，然后进行写操作；
如果文件打不开（比如文件只读），返回 NULL

- a:以追加方式打开文件
文件**不存在**，以指定文件名创建此文件(同 w)
若文件**存在**，从文件的**结尾处**进行写操作

说明:

如果不加 a 的话，打开文件的时候读写位置在文件的开始，对文件进行读写的时候都是从文件开始进行读写的。

如果加 a，打开已经存在的文件，读写位置在文件的末尾。

- +:同时以读写打开指定文件

模 式	功 能
r 或 rb	以只读方式打开一个文本文件（不创建文件）
w 或 wb	以写方式打开文件（使文件长度截断为 0 字节，创建一个文件）
a 或 ab	以添加方式打开文件，即在末尾添加内容，当文件不存在时，创建文件用于写
r+或 rb+	以可读、可写的方式打开文件(不创建新文件)

w+或 wb+	以可读、可写的方式打开文件 (使文件长度为 0 字节, 创建一个文件)
a+或 ab+	以添加方式打开文件, 打开文件并在末尾更改文件 (如果文件不存在, 则创建文件)

返回值:

成功: 打开的文件对应的文件指针

失败: 返回 NULL

以后调用 fopen 函数的时候, 一定要判断一下, 打开是否成功。

11.4 关闭文件 fclose

函数的头文件:

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

函数的定义:

```
int fclose(FILE *fp);
```

函数的说明:

关闭 fp 所代表的文件

注意一个文件只能关闭一次, 不能多次关闭。关闭文件之后就不能再文件指针对文件进行读写等操作了。

返回值:

成功返回 0

失败返回非 0

可以通过返回值, 来判断关闭文件是否成功。

例 6 :

```
#include <stdio.h>
int main()
{
    FILE *fp;
    int ret;
    fp=fopen("./test.txt","r+");
    if(fp==NULL)
    {
        perror("fopen");
        return 0;
    }
}
```

```
}  
printf("打开文件成功\n");  
ret=fopen(fp);  
if(ret==0)  
    printf("关闭文件成功\n");  
else  
    printf("关闭文件失败");  
return 0;  
}
```

11.5 一次读写一个字符

函数定义:

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

函数说明:

fgetc 从 stream 所标示的文件中读取一个字节，将字节值返回

返回值:

以 t 的方式：读到文件结尾返回 EOF

以 b 的方式：读到文件结尾，使用 feof(后面会讲)判断结尾

函数的定义:

```
int fputc(int c, FILE *stream)
```

函数的说明:

fputc 将 c 的值写到 stream 所代表的文件中。

返回值:

如果输出成功，则返回输出的字节值；

如果输出失败，则返回一个 EOF。

EOF 是在 stdio.h 文件中定义的符号常量，值为-1

注意：打开文件的时候，默认读写位置在文件的开始，如果以 a 的方式打开读写位置在文件的末尾
咱们向文件中读取字节或写入字节的时候，读写位置会往文件的末尾方向偏移，读写多少个字节，读写位置就往文件的末尾方向偏移多少个字节

例 7:

```
#include <stdio.h>  
int main(void)  
{  
    FILE *fp;  
    char ch;  
    fp=fopen("test.txt","r+");  
    if(fp==NULL)
```

```
{
    printf("Cannot open the file\n");
    return 0;
}
while( (ch = fgetc(fp))!=EOF )
{
    fputc(ch,stdout);
}
fclose(fp);
return 0;
}
```

例 8: 练习:

从一个文件(文本文件)中读取所有信息，写入另一个文件中

注意用 main 函数传参，第一个参数是源文件的名字，第二个参数是目的文件的名字

`./my_cp test.txt qiangge.txt`

```
#include <stdio.h>
int main(int argc,char *argv[])
{
    FILE *src_fd,*dest_fd;
    char ch;
    src_fd=fopen(argv[1],"r+");
    if(src_fd==NULL)
    {
        printf("Cannot open the file\n");
        return 0;
    }
    dest_fd=fopen(argv[2],"w");
    if(dest_fd==NULL)
    {
        printf("Cannot open the file\n");
        return 0;
    }
    while( (ch = fgetc(src_fd))!=EOF )
    {
        fputc(ch,dest_fd);
    }
    fclose(src_fd);
    fclose(dest_fd);
}
```

```
return 0;  
}
```

11.6 一次读写一个字符串

`char *fgets(char *s, int size, FILE *stream);`

从 `stream` 所指的文件中读取字符，在读取的时候碰到换行符或者是碰到文件的末尾停止读取，或者是读取了 `size-1` 个字节停止读取，在读取的内容后面会加一个 `\0`，作为字符串的结尾

返回值：

成功返回目的数组的首地址，即 `s`

失败返回 `NULL`

`int fputs(const char *s, FILE *stream);`

函数功能：

将 `s` 指向的字符串，写到 `stream` 所代表的文件中

返回值：

成功返回写入的字节数

失败返回 `-1`

例 9：

```
#include <stdio.h>  
int main(void)  
{  
    FILE *fp_read,*fp_write;  
    char string1[100];  
    if((fp_read=fopen("src.txt","r+"))==NULL)  
    {  
        printf("Cannot open the file\n");  
        return 0;  
    }  
    if((fp_write=fopen("dest.txt","w+"))==NULL)  
    {  
        printf("Cannot open the file\n");  
        return 0;  
    }  
    fgets(string1, 100, fp_read);  
    printf("%s\n",string1);  
}
```

```
fputs(string1,fp_write);  
fclose(fp_read);  
fclose(fp_write);  
return 0;  
}
```

11.7 读文件 fread

函数的定义:

```
size_t fread(void *ptr, size_t size, size_t nmem, FILE *stream);
```

函数的说明:

fread 函数从 stream 所标示的文件中读取数据，一块是 size 个字节，共 nmem 块，存放到 ptr 指向的内存里
返回值:

实际读到的块数。

例 1:

```
int num;  
num=fread(str,100,3,fp);
```

从 fp 所代表的文件中读取内容存放到 str 指向的内存中，读取的字节数为，每块 100 个字节，3 块。

返回值 num,

如果读到 300 个字节返回值 num 为 3

如果读到了大于等于 200 个字节小于 300 个字节 返回值为 2

读到的字节数，大于等于 100 个字节小于 200 个字节 返回 1

不到 100 个字节返回 0

11.8 写文件 fwrite

函数的定义:

```
size_t fwrite(void *ptr, size_t size, size_t nmem, FILE *stream);
```

函数的说明:

fwrite 函数将 ptr 指向的内存里的数据，向 stream 所标示的文件中写入数据，一块是 size 个字节，共 nmem 块。
返回值:

实际写入的块数

例 10 :

```
#include <stdio.h>  
struct stu  
{  
    char name[10];  
    int num;  
    int age;  
};  
boyb[10],boyb[2];
```

```
int main()
{
    FILE *fp;
    int i;
    if((fp=fopen("test.txt","wb+"))==NULL)
    {
        printf("Cannot open file!");
        return 0;
    }
    printf("input data\n");
    printf("name \ num \ age\n");
    for(i=0;i<2;i++)
        scanf("%s %d %d",boya[i].name,&boya[i].num,&boya[i].age);

    fwrite(boya,sizeof(struct stu),2,fp);    //将学生信息写入文件中
    rewind(fp);    //文件指针经过写操作已经到了最后·需要复位
    fread(boya,sizeof(struct stu),2,fp);    //将文件中的数据读入到内存中

    for(i=0;i<2;i++)
        printf("%s %d %d\n",boya[i].name,boya[i].num,boya[i].age);
    fclose(fp);
    return 0;
}
```

注意:

`fwrite` 函数是将内存中的数据原样输出到文件中。

`fread` 函数是将文件中的数据原样读取到内存里。

11.9 格式化读写文件函数

函数调用:

`fprintf` (文件指针, 格式字符串, 输出表列);

`fscanf` (文件指针, 格式字符串, 输入表列);

函数功能:

从磁盘文件中读入或输出字符

`fprintf` 和 `printf` 函数类似:

`printf` 是将数据输出到屏幕上 (标准输出), 而 `fprintf` 函数是将数据输出到文件指针所指定的文件中。

fscanf 和 scanf 函数类似:

scanf 是从键盘（标准输入）获取输入，而 fscanf 是从文件指针所标示的文件中获取输入。

例 11:

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    char ch1='a',ch2;
    int num1=50,num2;
    char string1[20]="hello",string2[20];
    float score1=85.5,score2;

    if((fp=fopen("test.txt","wb+"))==NULL)
    {
        printf("Cannot open the file\n");
        return 0;
    }
    fprintf(fp,"%c %d %s %f\n",ch1,num1,string1,score1);
    rewind(fp);
    fscanf(fp,"%c %d %s %f\n",&ch2,&num2,&string2,&score2);
    printf("%c %d %s %f\n",ch2,num2,string2,score2);
    fclose(fp);
    return 0;
}
```

11.10 随机读写

前面介绍的对文件的读写方式都是顺序读写，即读写文件只能从头开始，顺序读写各个数据；但在实际问题中常要求只读写文件中某一指定的部分，例如：读取文件第 200--300 个字节

为了解决这个问题可以移动文件内部的位置指针到需要读写的位置，再进行读写，这种读写称为随机读写。实现随机读写的关键是要按要求移动位置指针，这称为文件的定位。

完成文件定位的函数有：

rewind、fseek 函数

1、rewind 复位读写位置

rewind 函数

void rewind(文件指针);

函数功能:

把文件内部的位置指针移到文件首

调用形式:

```
rewind(文件指针);
```

例 12 :

```
fwrite(pa,sizeof(struct stu),2,fp) ;  
rewind(fp);  
fread( pb,sizeof(struct stu),2,fp);
```

2、ftell 测文件读写位置距文件开始有多少个字节

定义函数:

```
long ftell(文件指针);
```

函数功能:

取得文件流目前的读写位置.

返回值:

返回当前读写位置(距离文件起始的字节数), 出错时返回-1.

➤ 例如:

```
int length;  
length = ftell(fp);
```

3、fseek 定位位置指针 (读写位置)

fseek 函数 (一般用于二进制文件即打开文件的方式需要带 b)

定义函数:

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

//int fseek(文件类型指针, 位移量, 起始点);

函数功能:

移动文件流的读写位置.

参数:

whence 起始位置

文件开头	SEEK_SET	0
文件当前位置	SEEK_CUR	1
文件末尾	SEEK_END	2

位移量:

以起始点为基点, 向前、后移动的字节数, 正数往文件末尾方向偏移, 负数往文件开头方向偏移.

例 13:

```
fseek(fp,50,SEEK_SET)  
fseek(fp,-50,SEEK_END);  
fseek(fp,0,SEEK_END);
```

练习:

将一个**未知大小**的文件(文本文件)全部读入内存, 并显示在屏幕上

参考: fseek ftell rewind fread malloc

1、打开文件 fopen , 注意用 b 的方式打开

- 2、定位文件的读写位置到文件的末尾 `fseek`
- 3、测文件的字节数 `len` `ftell`
- 4、复位读写位置到文件的开始 `rewind`
- 5、根据第 3 步得到的字节数，申请内存 `malloc` 注意多申请一个字节存放 `'\0'`
- 6、从文件中读取内容，存到申请的空间里 `fread`
- 7、最后一个字节变成 `'\0'`
- 8、打印读出来的内容到屏幕上，`printf`
- 9、关闭文件 `fclose`
- 10、释放内存 `free`

第12章 Makefile

1: 什么是 make

`make` 是个命令，是个可执行程序，用来解析 `Makefile` 文件的命令
这个命令存放在 `/usr/bin/`

2: 什么是 makefile?

`makefile` 是个文件，这个文件中描述了咱们程序的编译规则

咱们执行 `make` 命令的时候，`make` 命令会在当前目录下找 `makefile` 文件，
根据 `makefile` 文件里的规则，编译咱们的程序

注意：`Makefile` 规则文件是咱们程序员根据自己的程序，编写的编译规则

3: 采用 Makefile 的好处

- 1、简化编译程序的时候输入的命令，编译的时候只需要敲 `make` 命令就可以了
- 2、可以节省编译时间，提高编译效率

12.1 make 概述

- 1、GNU `make` 是一种代码维护工具
- 2、`make` 工具会根据 `makefile` 文件定义的规则和步骤，完成整个软件项目的代码维护工作。
- 3、一般用来简化编译工作，可以极大地提高软件开发的效率。
- 4、windows 下一般由集成开发环境自动生成
- 5、linux 下需要由我们按照其语法自己编写

make 主要解决两个问题：

一、大量代码的关系维护

大项目中源代码比较多，手工维护、编译时间长而且编译命令复杂，难以记忆及维护

把代码维护命令及编译命令写在 `makefile` 文件中，然后再用 `make` 工具解析此文件自动执行相应命令，可实现代码的合理编译

二、减少重复编译时间

在改动其中一个文件的时候，能判断哪些文件被修改过，可以只对该文件进行重新编译，然后重新链接所有的目标文件，节省编译时间

12.2 makefile 语法及其执行

12.2.1 makefile 语法规则

目标：依赖文件列表

<Tab>命令列表

1、目标：

通常是要产生的文件名称,目标可以是可执行文件或其它 obj 文件,也可是一个动作的名称

2、依赖文件：

是用来输入从而产生目标的文件

一个目标通常有几个依赖文件（可以没有）

3、命令：

make 执行的动作,一个规则可以含几个命令（可以没有）

有多个命令时，每个命令占一行

例 1：简单的 Makefile 实例

```

main.c                                     main.h
1 #include <stdio.h>                       1 #define PI 3.1415926
2 #include "main.h"
3 int main (void)
4 {
5     printf("hello make world\n");
6     printf("PI=%lf\n", PI);
7     return 0;
8 }
    
```

```

makefile:
1 main:main.c main.h
2     gcc main.c -o main
3 clean:
4     rm main
    
```

12.2.2 make 命令格式

make [-f file] [targets]

1.[-f file]:

make 默认在工作目录中寻找名为 GNUmakefile、makefile、Makefile 的文件作为 makefile 输入文件

-f 可以指定以上名字以外的文件作为 makefile 输入文件

2.[targets]:

若使用 make 命令时没有指定目标，则 make 工具默认会实现 makefile 文件内的第一个目标，然后退出指定了 make 工具要实现的目标，目标可以是一个或多个（多个目标间用空格隔开）。

例 2: 稍复杂的 Makefile 实例

main.c 调用 printf1.c 中的 printf1 函数，同时需要使用 main.h 中的 PI，printf1.h 需要使用 main.h 中的 PI

<pre>main.c #include <stdio.h> #include "main.h" #include "printf1.h" int main (void) { printf("hello make world\n"); printf("PI=%lf\n",PI); printf1(); return 0; } main.h #define PI 3.1415926</pre>	<pre>printf1.c #include <stdio.h> #include "main.h" void printf1(void) { printf("hello printf1 world PI=%lf\n",PI); } printf1.h extern void printf1();</pre>
--	--

稍微复杂的 makefile 编写

```
1 main:main.o printf1.o
2     gcc main.o printf1.o -o main
3 main.o:main.c main.h printf1.h
4     gcc -c main.c -o main.o
5 printf1.o:printf1.c main.h
6     gcc -c printf1.c -o printf1.o
7 clean:
8     rm *.o main
```

如 printf1.c 和 printf1.h 文件在最后一次编译到 printf1.o 目标文件后没有改动，它们不需重新编译 main 可以从源文件中重新编译并链接到没有改变的 printf1.o 目标文件。

如 printf1.c 和 printf1.h 源文件有改动，make 将在重新编译 main 之前自动重新编译 printf1.o。

12.2.3 假想目标:

前面 makefile 中出现的文件称之为假想目标

假想目标并不是一个真正的文件名，通常是一个目标集合或者动作

可以没有依赖或者命令

一般需要显示的使用 `make + 名字` 显示调用

```
all:exec1 exec2
```

```
clean:
```

```
<Tab>rm *.o exec
```

12.3 makefile 变量

12.3.1 makefile 变量概述

makefile 变量类似于 C 语言中的宏，当 makefile 被 make 工具解析时，其中的变量会被展开。

变量的作用：

- 保存文件名列表

- 保存文件目录列表

- 保存编译器名

- 保存编译参数

- 保存编译的输出

- ...

12.3.2 makefile 的变量分类：

1、自定义变量

在 makefile 文件中定义的变量。

make 工具传给 makefile 的变量。

2、系统环境变量

make 工具解析 makefile 前，读取系统环境变量并设置为 makefile 的变量。

3、预定义变量（自动变量）

12.3.3 自定义变量语法

定义变量：

```
变量名=变量值
```

引用变量：

```
$(变量名)或${变量名}
```

makefile 的变量名：

makefile 变量名可以以数字开头

注意：

- 1、变量是大小写敏感的

- 2、变量一般都在 makefile 的头部定义

- 3、变量几乎可在 makefile 的任何地方使用

例 2_2：

修改例 2 中的 makefile，使用自定义变量使其更加通用。

```

1 cc=gcc
2 #cc=arm-linux-gcc
3 obj=main.o printf1.o
4 target=main
5 cflags=-Wall -g
6
7 $(target):$(obj)
8     $(cc) $(obj) -o $(target) $(cflags)
9 main.o:main.c main.h printf1.h
10     $(cc) -c main.c -o main.o $(cflags)
11 printf1.o:printf1.c main.h
12     $(cc) -c printf1.c -o printf1.o $(cflags)
13 clean:
14     rm $(obj) $(target)

```

make 工具传给 makefile 的变量

执行 make 命令时，make 的参数 options 也可以给 makefile 设置变量。

#make cc=arm-linux-gcc

```

1 cc=gcc
2 main:main.c main.h
3     $(cc) main.c -o main
4 clean:
5     rm main

```

12.3.4 系统环境变量

make 工具会拷贝系统的环境变量并将其设置为 makefile 的变量，在 makefile 中可直接读取或修改拷贝后的变量。

#export test=10

#make clean

#echo \$test

```

1 main:main.c main.h
2     gcc main.c -o main
3 clean:
4     rm main -rf
5     echo $(PWD)
6     echo "test=$(test)"

```

12.3.5 预定义变量

makefile 中有许多预定义变量，这些变量具有特殊的含义，可在 makefile 中直接使用。

\$@ 目标名

\$< 依赖文件列表中的第一个文件

\$^ 依赖文件列表中除去重复文件的部分

AR 归档维护程序的程序名，默认值为 ar

ARFLAGS 归档维护程序的选项

AS 汇编程序的名称，默认值为 as

ASFLAGS 汇编程序的选项

CC C 编译器的名称，默认值为 cc

CFLAGS C 编译器的选项

CPP C 预编译器的名称，默认值为\$(CC) -E

CPPFLAGS C 预编译的选项

CXX C++编译器的名称，默认值为 g++

CXXFLAGS C++编译器的选项

例 2_3:

修改例 2 中的 makefile，使用预定义变量,使其更加通用。

```

1 obj=main.o printf1.o
2 target=main
3 CFLAGS=-Wall -g
4
5 $(target):$(obj)
6     $(CC) $^ -o $@ $(CFLAGS)
7 main.o:main.c main.h printf1.h
8     $(CC) -c $< -o $@ $(CFLAGS)
9 printf1.o:printf1.c main.h
10    $(CC) -c $< -o $@ $(CFLAGS)
11 clean:
12    rm $(obj) $(target)
    
```

Makefile 练习

编写 Makefile 完成计算器程序(01_练习代码/calc)的编译，并通过假想目标 (clean) 清除目标文件

第一步：不使用任何变量完成功能

第二步：使用自定义变量让程序更加通用

第三步：使用预定义变量让程序更加通用