目录

一、导师介绍

游成伟
15801588497(微信)
409025260@qq.com

二、学习方法

笔记看三遍
代码好好敲
背代码背笔记

三、计算机定义

定义：仅仅就是一个统称,包括 PC 机,笔记本,服务器,手机,扫地机器人,无人机,路由器,电视盒子,智能音箱(小爱,小度),摩拜单车等就是一个做数字(1,2,3,-1,-2....)运算的机器。

四、计算机分类

1.软件和硬件
(1)硬件分两类
(a)硬件三大必要：
CPU:类似大脑
两大功能：[1]数据运算 例如：1+1=2；
[2]控制外设两点：
{1}：从外设读取数据
{2}：向外设写入数据
内存(RAM)：类似于肝脏
功能：暂存 CPU 运算前的数据和 CPU 运算之后的数据
特点：优点：存储数据速度快
缺点：系统重启/掉电数据立马丢失，容量小
闪存(硬盘，ROM)：类似肠胃
功能：永久性存储数据
特点：优点：数据掉电不丢失，容量大
缺点：CPU 访问闪存硬盘中存储的数据贼慢
结论：CPU 处理数据的流程：
先将闪存的数据放到内存中
然后 CPU 在从内存中获取数据然后进行计算
结果是数据的计算速度快
(b)外设(外接的硬件设备)：
定义：计算机中除了 CPU(除了寄存器)，其他都是外设
功能：给 CPU 数据运算时提供数据还能暂存或者永久性存储 CPU 运算之后的结果
(2)软件部分：三大类
(a)操作系统：
[1]主流的操作系统：
Windows

Linux(重点):主要用于服务器，推荐三大主流的 linux 系统

Ubuntu(基于 debian)

Fedora

Kali(黑客 hacker,基于 ubuntu)　c++　图片：龙

安卓：本质还是 linux 系统

Vxworks:硬实时(不卡壳)嵌入式操作系统，用于航空航天，军品

(b)操作系统功能：

操作硬件：有操作系统中的驱动完成

管理和分配各种硬件资源：CPU 资源，内存资源

(b)shell 终端程序：(windows 的终端是 cmd)

功能：从键盘上获取用户输入的命令，然后让 CPU 执行这个命令

(c)其他应用程序：例如 QQ,PUBG,

五、linux 发展简介

Linux 之父：linus

Linus 系统诞生时间：1991

推荐视频：《the code linux》

应用场合：

移动桌面系统：安卓

服务器：90%以上都是 linux

案例：打开自己电脑的 ubuntu32 系统,然后点击终端程序(shell 终端程序)然后输入命令：qtcreator,看看系统给你什么惊喜

linux 系统理念（信仰)：一切皆文件

在 linux 系统中,计算机中任何外设都是以文件的形式存在,某个程序 QQ 访问某个文件本质就是访问这个文件对应的硬件外设,也就是在 linux 系统中如果要想访问某个硬件必须先找到对应的文件。

例如：vide0 文件对应摄像头,event0 文件对应键盘,event1 文件对应触摸屏,fb0 文件对应显示器等。

六、掌握 linux 系统中跟文件相关系统的概念

目录：俗称文件夹,最终用于存放文件,当然目录里面还可以存在目录

根目录：linux 系统中所有文件的最上一层目录,用"/"来表示

类似：C:\盘,然后里面有一堆的目录和文件

子目录：位于其它目录里面的目录

例如：/home/tarena/

注意：目录之间用"/来分割"

说明：

第一个"/"：表示根目录

home:位于根目录下面的一个子目录

第二个"/":表示 home 子目录和 tarena 子目录之间的分割

tarena：位于 home 目录下的一个子目录

第三个“/”:表示 tarena 子目录和里面的子目录或者文件的分割

父目录：子目录的上一次目录

例如：/home/tarena/

home 目录是 tarena 目录的父目录
根目录/是 home 目录的父目录
当前目录：正在使用或者所在的目录,用“.”来表示
例如：./tarena/
.当前目录是 tarena 目录的父目录
tarena 目录是.当前目录的子目录
注意：当前目录.也可以省略
例如：tarena/
表示当前目录下有一个 tarena 目录
上一级目录：当前目录的上一级目录,用“..”来表示
例如：../tarena/
..是 tarena 目录的上一级目录也就是 tarena 的父目录
tarena 就是..上一级目录的子目录
主目录：登录 linux 系统时,需要指定一个 linux 用户,比如：tarena 用户
一旦登录完成,linux 系统会给这个用户指定一个固定的目录
此目录一定位于/home/目录下,并且目录名跟用户名同名
例如：tarena 用户的主目录为：/home/tarena/
还可以用"~"来表示
例如：~/stdc/
说明：stdc 目录位于当前用户的主目录,比如：/home/tarena/
所以：~/stdc/等价于/home/tarena/stdc/,两种书写方法
路径：表示文件或者目录的位置信息,又分两种：绝对路径和相对路径
绝对路径：路径的表示形式中必须以根目录"/"开头
例如：/home/tarena/stdc/,此种表示形式就是绝对路径
相对路径：路径的表示形式中不是以根目录"/"开头
例如：./stdc/,此种表示形式就是相对路径
或者：stdc/,此种表示形式就是相对路径
切记切记切记：实际开发,软件代码中如果涉及到路径问题,建议采用绝对路径,安全
例如：
open(./stdc/我的密码文件.txt)
要想打开成功,此文件必须位于当前目录下的 stdc 目录中但此时无法获取用户的当前目录，open(/home/tarena/stdc/我的密码文件.txt)
附加：df -T：显示文件系统信息

七、务必掌握以下 linux 系统中跟文件和目录相关的操作命令

由衷建议：玩 linux 就要用终端,全命令行操作,不要像 windows 中各种图标操作,各种鼠标左键,右键,否则造鄙视。
1.明确：玩 linux 系统,必须通过终端程序通过命令行模式来玩
明确：linux 命令格式：
命令名　选项(前面加-表示)　参数
明确：常用一个神奇按键：TAB 键,只需输入前几个字母,按 TAB 键自动补全后面内容
2.罗列 linux 相关的重要命令
(1)clear：清屏命令
格式：clear

(2)pwd:获取当前目录的路径命令

格式：pwd

(3)cd:目录切换命令

格式：cd 新目录的路径

例如：

cd /home/tarena/ 进入到/home/tarena/目录下

pwd 获取当前目录的路径

cd ../ 进入上一级目录

pwd

cd ../

pwd

cd ~ 进入主目录下,例如：/home/tarena/

pwd

(4)ls:查看指定路径里面的内容

格式：ls 目录的路径或者文件的路径 //查看粗略信息

ls -lh 目录的路径或者文件的路径 //查看详细信息

ls -a 查看隐藏文件

ls -l 目录 同 ls - lh

ls -R 递归显示

演示案例：

cd /home/tarena/

ls //查看当前目录下里面的内容,此时查看的粗略的信息

ls -lh //查看当前目录下所有内容的详细信息

例如：拿其中一个文件说明：

-rwxrw-r-- 1 tarena tarena 659 Aug 10 11:51 某个文件名

说明：

第一个"-"：表示此文件为一个普通的文本文件，如果第一个用'd'表示,说明此乃目录也,不是文件。

第一个"rwx":表示当前登录 linux 系统的用户对此文件可以进行读,写,可执行

第二个"rw-":表示当前用户所在组对此文件可以读,写,不可执行

第三个"r--":表示其他用户对此文件只能读,不能写,不能执行

"1"表示此文件的硬连接的个数(无需关注)

tarena:表示当前用户名

tarena:表示当前用户所在的组名

659：表示此文件的大小(单位是字节,后续课程讲解)

Aug 10 11:51：表示此文件的创建日期

最后就是文件名

ls -lh qw.c //单独查看 qw.c 文件的详细信息

ls -lh /home/tarena/ //单独查看/home/tarena 目录下所有内容的详细信息

(5)touch:创建空文件命令

命令格式：touch 新文件名

演示案例：

cd /home/tarena/

touch hello.txt //在当前目录下创建空文件 hello.txt
touch /home/tarena/hello1.txt //在指定的目录下创建空文件
ls -lh hello.txt
ls -lh /home/tarena/hello1.txt
(6)mkdir:创建空目录命令
命令格式：mkdir -p 新目录名
演示案例：
cd ~
pwd
mkdir -p hello //在当前目录下新建新目录 hello
ls -lh hello/ //查看新目录里面的内容
mkdir -p /home/tarena/hello1 //在指定的目录下创建 hello1
mkdir -p /home/tarena/hello2/hello3/hello4 //连续创建 hello2,3,4 三个目录
ls -lh /home/tarena/hello1/
ls -lh /home/tarena/hello2/
ls -lh /home/tarena/hello3/
ls -lh /home/tarena/hello4/
(7)rm:删除文件或者目录命令
命令格式：rm -fr 文件 1 文件 2 ... 文件 N 目录 1 目录 2 ...目录 N
演示案例：
cd /home/tarena/
rm -fr hello.txt hello1.txt hello hello1 hello2
ls -lh hello/
附加：rm -i：交互提示
(8)cp:拷贝文件或者目录命令
命令格式：
cp -fr 源文件 新文件
cp -fr 源文件 指定目录下/
cp -fr 源目录 新目录
cp -fr 源目录 指定目录下/
附加：cp -r 拷贝文件夹
cp -i 覆盖时交互提示
案例演示：
mkdir -p /home/tarena/test/
cd /home/tarena/test/
touch hello.txt
mkdir hello
ls -lh
cp -fr hello.txt hello1.txt
ls -lh
cp -fr hello.txt hello/
ls -lh
ls -lh hello/
cp -fr hello hello1

ls -lh
mkdir -p hello2
ls -lh
cp -fr hello hello2
ls -lh hello2/

(9)mv:剪切命令

命令格式：

mv 源文件 新文件
mv 源文件 指定目录下/
mv 源目录 新目录
mv 源目录 指定目录下/

案例：

mkdir -p /home/tarena/mvtest/
cd /home/tarena/mvtest/
touch hello.txt
mkdir -p hello
ls -lh
mv hello.txt hello1.txt
ls -lh
mv hello1.txt hello/
ls -lh
ls -lh hello/
mv hello hello1
ls -lh
mkdir -p hello2
ls -lh
mv hello1 hello2/
ls -lh
ls -lh hello2

(10)cat:快速查看文件内容命令

命令格式：cat 文件名

cat -s 多个空行合并成一个空行显示
cat -b 加行号显示
nl：显示行号
head -数字（默认 10 行）
tail -数字 （默认 10 行）

(11)echo:打印输出命令,就是在显示屏上显示输出信息

命令格式：echo 要显式的信息

例如：echo 我是大神

注意：echo 命令一般要配合">"或者">>"一起使用

命令格式：echo 要输出的信息 > 文件

说明：向文件写入要输出的信息,会将文件原先的内容清空

echo 要输出的信息 >> 文件

说明：向文件的结尾追加要输出的信息,文件原先的内容不变

综合演练：cat 和 echo

```
mkdir  -p  /home/tarena/echotest/
cd  /home/tarena/echotest/
touch  hello.txt
ls  -lh
cat  hello.txt  //查看 hello.txt 里面的内容
echo  我是大神  >  hello.txt  //向 hello.txt 文件写入内容：我是大神
cat  hello.txt  //查看内容
echo  我是小神  >  hello.txt
cat    hello.txt
echo  我是大神  >>  hello.txt
cat  hello.txt
```

(12)find:查找文件命令

命令格式：find 路径 -name 文件名

例如：find /usr/include/ -name stdio.h

意思：到/usr/include 目录下找一个名称为 stdio.h 的文件

```
cd  /home/tarena/mvtest/
find  .  -name  hello1.txt
```

(13)grep:在文件中搜索单词命令

命令格式：grep -Rn "要搜索的信息" 文件名

例如：

```
mkdir  -p  /home/tarena/greptest/
cd  /home/tarena/greptest
touch  hello.txt
echo  123  >  hello.txt
echo  abc  >>  hello.txt
echo  abc  >>  hello.txt
echo  我是大神  >>  hello.txt
echo  我是大神  >>  hello.txt
grep  -Rn  "abc"  hello.txt
grep  -Rn  "我是大神"  hello.txt
```

(14)tar:打压缩包和解压缩包命令

三条终极命令：

1.打压缩包两条命令：

tar -jcvf 压缩包名.tar.bz2 文件或者目录

意思：将文件或者目录制作成 bz2 格式的压缩包

tar -zcvf 压缩包名.tar.gz 文件或者目录

意思：将文件或者目录制作成 gz 格式的压缩包

2.解压缩命令一条：

tar -xvf 压缩包名.tar.bz2 / 压缩包名.tar.gz

案例：

```
cd  /home/tarena
tar  -jcvf  greptest.tar.bz2  greptest/   //打包
ls  -lh  greptest.tar.bz2
```

tar -zcvf greptest.tar.gz greptest/ //打包
ls -lh greptest.tar.gz
rm -fr greptest
tar -xvf greptest.tar.bz2 //解包,生成 greptest 目录
ls -lh greptest/
rm -fr greptest
tar -xvf greptest.tar.gz //解包,生成 greptest 目录
ls -lh greptest/

八、vim 编辑器(神器)

1.介绍几款计算机软件开发常用的编辑器(编写代码的工具)
vim(神器),vscode(微软),sublime(web 开发),eclipse(java 开发)
2.vim 编辑器对应的命令 vim 命令
命令格式：vim 文件名
3.vim 的三种模式
可视模式：只能看文件内容,不能编辑文件
编辑模式：可以看也可以编辑
命令行模式：给 vim 编辑器发送命令,此模式也不能修改文件
注意：刚打开文件时的模式默认为可视模式
4.三种模式的切换
可视模式-----按 i 键----->进入编辑模式
编辑模式-----按 ESC 键--->进入可视模式
可视模式-----按 shift+:->进入命令行模式
命令行模式---按 ESC 键--->可视模式
编辑模式-----先按 ESC 进入可视模式,再按 shift+:-->进入命令行模式
附加：a(append)：在光标下一个位置进入编辑模式
A：在这一行最后进入编辑模式
I：在这一行行首进入编辑模式
o：向上一行插入一行进入编辑模式
O：向下一行插入一行进入编辑模式
5.掌握 vim 命令行模式下的操作命令
w:保存文件命令
q:不保存并且退出 vim 命令
wq:保存并且退出 vim 命令
q!:不保存并且强制退出 vim 命令
%s/老信息/新信息/g:将文件中所有的老信息用新信息内容替换
.,$s/老信息/新信息/g:同上
s/老信息/新信息：当前行
g 代表全行，没有代表第一个
直接输入行号：跳转到对应的行
vs 文件名：左右分屏
sp 文件名：上下分屏
x：相当于 wq
附加：

！pwd:查看文件当前位置
！data：查看时间
w file：把当前文件另存为其他名字不退出
r file:引入文件
注意：屏幕之前的切换快捷键用：ctrl+ww(前提是先进入可视模式)
案例：用 vim 编辑一个文章并且保存退出
案例：同时编辑 4 个文件并且保存退出

6. 掌握 vim 可视模式下的快捷键(核心)

(1)方向键：h(左移)j(下移)k(上移)l(右移),严重鄙视用上下左右方向键
(2)行选中：shift+v 然后按方向键选中
(3)列选中：ctrl+v 然后按方向键选中
(4)复制：y
(5)粘贴：p
(6)剪切：x
(7)撤销：u
(8)删除：d
(9)跳转到文件的结尾：G 或者 shift + g
(10)跳转到文件的开头：gg
(11)保存并且退出 vim：ZZ
(12)自动补全：ctrl+n(前提是先进入编辑模式,然后输入一个单词开头，然后按 ctrl+n 自动补全整个单词)
(13)[N]x:剪切从光标位置开始的连续 n 个字符
(14)[N]dd:删除从光标位置开始的连续 n 行，还有剪切的作用
(15)[N]yy:复制从光标位置开始的连续 n 行
(16)p:粘贴在 main 下一行
(17)P:粘贴在 main 上一行
(18)n：光标移动到行号
(19)set number:设置行号
(20)set nonumber:取消设置行号
(21)/单词：查找单词
(22)gg=G:自动调节格式

九、C 程序的变量

1.明确：计算机的主要功能：做数据运算,例如：1+2=3 等等等，而计算机这个数据运算的过程需要程序来完成！

例如：

```
printf("1+2 = %d\n", 1+2); //输出结果：1+2 = 3
```

2.回顾：计算机内存功能和特点

内存的功能：暂存 CPU 运算前的数据和运算之后的数据，理论上硬盘,闪存也可以存放,但是他们的速度太慢了所以用内存来存储。

例如：1(内存中)+2(内存中)=3(内存中)

内存的特点：优点：速度快　缺点：容量小,掉电数据丢失

终极结论：计算机程序最终玩内存,写程序最终的目的就是为了操作内存

3.务必掌握内存相关的概念

(1)字节(byte)：计算机把内存分成一格一格,每一格用来存储一个数字(例如：1)，每一格对应的专业术语叫字节。

具体参见：字节.png 图

(2)地址(address)：计算机给内存中每个字节指定了一个编号,编号从 0 开始此编号专业术语叫地址。

(3)存储区(buffer):计算机把一个字节或者多个字节形成的存储数据的区域，简称存储区。

(4)首地址(base address):存储区中第一个字节的地址称之为存储区的首地址。

(5)问：C 程序如何获取内存的存储区呢？只要获取到存储区,C 程序就可以从存储区中读取存储的数字或者向存储区中存储一个新的数字。

答：通过定义变量来分配获取内存

4.定义变量功能：为了给 C 程序分配获取有效的内存存储区并且获取到的存储区中的数字是可以改变的。

比如：现在存一个 250,将来让它存 520

5.定义变量的语法格式：数据类型 变量名 = 初始值(可以有,可以没有);

例如：int a = 250;

语义：

int:它是一个关键字,表示将来要从内存中分配 4 字节的存储区空间

a：变量名,类似给分配的 4 字节内存空间取个名叫 a

250:就是给分配的 4 字节内存空间放一个数字 250

6.定义变量的四种形式：

(1)只定义不初始化(只分配内存,不会向内存储存数字)

例如：

int a;

切记：此时变量 a 对应的内存存储区存储的是一个乱七八糟的随机数

(2)定义并且初始化

例如：int a = 250;

(3)连续定义多个变量不初始化(连续分配多个内存存储区)

例如：int a, b, c;

注意：a,b,c 里面储存的数字都是乱七八糟的随机数

(4)连续定义并且初始化

int a = 2, b = 3, c = 4;

(5)变量不可同名

```
int a = 250;
int a = 520; //gcc 编译报错
```

案例：编写 C 程序,通过定义变量来分配内存存储区

实验步骤：

```
cd /home/tarena/stdc/day02/
vim var.c //编辑
```

保存退出

```
gcc -o var var.c //编译
./var //运行
```

7.标识符命名规则

(1)标识符概念：就是变量名(例如：a)或者函数名(例如：printf)统称标识符

(2)标识符命名规则：

(a)第一个字母不能是数字,只能是字母或者下划线,区分大小写

例如：

```
int var = 250; //可以
int 2var = 250; //gcc 报错
int v2ar = 250; //可以
int _2var = 250; //可以
int Var = 250;//和前面的 int var = 250; 不一样
```

(b)名字尽量见名知意

```
int a = 18; //无法通过 a 获取要表达 18 是一个岁数
```

改造：

```
int age = 18;
```

(c)命名方法形式两种：

驼峰方式(windows 程序员)：

例如：int CreateWindows = 250;

下划线方式(linux 程序员)：

例如：int create_windows = 250;

(d)标识符名称不能是关键字

关键字：具有一定特定功能的单词,例如：int, include void return

例如：

```
int int = 250; //gcc 报错
int include = 250; //gcc 报错
```

8.C 语言的编码风格(无对和错之分)：

(1)各种该

该有空格的加空格

该对齐的对齐

该有 TAB 键加 TAB 键

该独占一行的独占一行

例如：

```
int main(void){printf("我是大神\n");return 0}
```

改造：

```
int main(void)
{ //独占一行
        printf("我是大神\n"); //独占一行,前面加 TAB 键
        return 0; //独占一行,前面加 TAB 键,并且跟 printf 对齐
} //独占一行
```

(2)代码尽量集中在屏幕左侧 2/3 区域,右侧 1/3 留白

多余代码换行处理

例如：

```
printf("aaaaaaaaaaaaaaaaaaaaaa bbbbbbbbbbbbbbbbb ccccccccccccc\n");
```

改造：

```
printf("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa "
        "bbbbbbbbbbbbbbbbbbbbbbbbbbbbb "
        "ccccccccccccccccccccccccccccc\n");
```

(3)注意标识符的问题

十、详解变量的数据类型

1.数据类型功能：让编译器 gcc 计算变量将来要分配的内存空间大小

2.C 语言的 12 类基本数据类型：

关键字名称	含义	分配内存大小	数字范围
char	字符类型(本质是单字节的整数)	1 字节	-128~127(背)
unsigned char	非负(0 和正数)的单字节整数	1 字节	0~255(背)
short	双字节整数	2 字节	-32768~32767
unsigned short	非负双字节整数	2 字节	0~65535
int	四字节整数	4 字节	-2^{31}~2^{31}-1
unsigned int	非负四字节整数	4 字节	0~2^{32}-1
long	四字节整数	4 字节	-2^{31}~2^{31}-1
unsigned long	非负四字节整数	4 字节	0~2^{32}-1
long long	八字节整数	8 字节	-2^{63}~2^{63}-1
unsigned long long	非负八字节整数	8 字节	0~2^{64}-1
float	单精度浮点数	4 字节	省略
double	双精度浮点数	8 字节	省略

例如：

```
char a = 1; //分配 1 字节内存空间并且储存 1 这个数字
short b = 250; //分配 2 字节内存空间并且储存 250 这个数字
unsigned long c = 520; //分配 4 字节内存空间并且储存 520 这个数字
float d = 12.1; //分配 4 字节内存空间并且储存 12.1 这个数字
double e = 23.32; //分配 8 字节内存空间
```

3.注意事项

(1)每种数据类型都有有符号和无符号

(a)有符号数字有正负之分,无需添加关键字 unsigned,但是可以添加 signed 或者不加 signed 关键：

例如：

```
int a = -10;
```

等价于：

```
signed int a = -10;
```

(b)无符号数字只有 0 和正数,必须用 unsigned 关键字修饰

例如：unsigned int a = 250;

(2)切记：unsigned int 和 unsigned long 对于 gcc 编译器来说不一样!

对于 32 位系统和 64 位系统,unsigned int unsigned long 都是 4 字节

对于 32 位系统,unsigned long 都是 4 字节

对于 64 位系统,unsigned long 都是 8 字节

将来学习指针的时候用得着!

(3)利用关键字 sizeof 能够查看变量或者数据类型分配的内存大小

sizeof 语法格式：分配内存大小 = sizeof(变量名或者数据类型关键字名);

例如：

```
int a = 1;
printf("%d %d\n", sizeof(a), sizeof(int)); //打印：4 4
```

切记：sizeof 关键字圆括号里面如果对变量进行赋值修改,此修改是无效的!

案例：编写 C 程序利用 sizeof 打印出 12 个数据类型分配内存大小

实施步骤：

```
mkdir  /home/tarena/stdc/day03/
cd  /home/tarena/stdc/day03/
vim  sizeof.c
```

分步法

```
gcc  -E  -o  sizeof.i  sizeof.c
gcc  -c  -o  sizeof.o  sizeof.i
gcc  -o  sizeof  sizeof.o
./sizeof
```

一步到位法：

```
gcc  -o  sizeof  sizeof.c
./sizeof
```

4.详解 12 类基本数据类型之字符类型 char

(1)字符常量的定义：用单引号括起来表示,其值不可改变,固定的

例如：‘A'表示字符常量字符 A

‘1’表示字符常量字符 1

(2)字符变量的定义形式：

char 变量名 = 初始值

unsigned char 变量名 = 初始值

切记切记切记：字符变量分配的 1 字节内存空间本质上存储的是字符，对应的一个整数,这个整数对应的专业术语叫 ASCII 码。

问：字符和对应的整数如何对应的,如何映射呢?

答: 映射关系都是计算机硬件已经做好。

对应关系：百度查 ASCII 码表

字符常量	ASCII 码
‘A'	65
‘B’	66
‘C’	67
...	...
'a'	97
'b'	98
'c'	99
...	

所以：'d' - 'a' 本质是 100-97 = 3

例如：

```
char a = 'A';
```

语义：分配 1 字节内存空间然后向这个内存空间放一个 65 这个数字，千万注意不是放'A'在这个内存空间,而是放字符'A'对应的 ASCII 码 65，CPU 将来拿着内存中的 65 这个数字到计算机中闪存中取一个字符'A',然后把这个字符'A'在显示器上描绘出来。

类似：拿着粮票到商店拿粮食

参见：字符.png

(3)字符变量打印输出 printf 使用的占位符是:

%c:在显示器上按照字符的形式显示

%d:在显示器上按照 ASCII 码整数形式显示

```
char c = 'A';
printf("%c  %d\n", c, c); //A 65
```

案例：编写 C 程序掌握字符变量

参考代码：char.c

(4)转义字符：特定功能的字符

'\n':让光标移动到下一行的行首

'\r':让光标移动到当前行的行首

'\t':得到一个 TAB 键

'\\':得到\

'\'':得到'

'\"':得到"

'%%':得到%

案例：利用 printf 打印如下格式：

abc

(TAB 键)def\'hjk\'\"lmn\"%xyz%

答案：

```
printf("abc\n\tdef\\\'hjk\\\'\\\"lmn\\\"%%xyz%%\n"
```

5.详解 12 类基本数据类型之整型类型 int

(1)整型类型关键字为 int

可以用 short,unsigned short,long,unsigned long 进行修饰

所以 int 类型可以代表 2 字节或者 4 字节

(1)int 类型的六种形式：

形式 1：short (int)用 2 字节表示,有符号

例如：

```
short a = -250; //简化版本写法
```

等价于

```
short int a = -250; //完整版本写法
```

形式 2：unsigned short (int)用 2 字节表示,无符号

例如：

```
unsigned short a = 520;
```

等价于

```
unsigned short int a = 520;
```

形式 3：long (int)用 4 字节表示,有符号

例如：

```
long a = 520;
```

等价于：

```
long int a = 520;
```

形式 4：unsigned long (int)用 4 字节表示,无符号

例如：

```
unsigned long a = 520;
```

等价于

```
unsigned long int a = 520;
```

形式 5：int 用 4 字节表示,有符号

例如：int a = 520;

形式 6：unsigned int 用 4 字节表示 无符号

例如：unsigned int a = 520;

6.整型常数(不可改变的数字,细腻,高薪)

100:gcc 编译器默认把它当 int 类型看待

100L:gcc 编译器默认把它当 long 类型对待

100LL:gcc 编译器默认把它当 long long 类型对待

100u:gcc 编译器默认把它当 unsigned int 类型对待

100UL:gcc 编译器默认把它当 unsigned long 类型对待

这个知识点用于数据类型转换使用,后面课程详解

7.浮点类型：float,double

1.23:gcc 编译器默认把它当 double 类型对待

1.23f:gcc 编译器默认把它当 float 类型对待

注意：浮点数在计算机中是一个近似值,无限趋近,不能相等的值

0f 不等于整数 0,0f 可能是一个这么值：0.000000001

留白：float 有效数字为 4 位，double 有效数字是 8 位

8.数据类型和对应的占位符(给 printf 函数使用)

数据类型	内存大小	占位符
char	1 字节	%c,%hhd
unsigned char	1 字节	%c,%hhu
short	2 字节	%hd
unsigned short	2 字节	%hu
int	4 字节	%d
unsigned int	4 字节	%u
long	4 字节	%ld
unsigned long	4 字节	%lu
float	4 字节	%f 或者%g
double	8 字节	%lf 或者%lg

%f,%lf 会保留小数点后面的 0,2.300000

%g,%lg 不会保留,2.3

附加：%e:指数形式占位符

案例：编写 C 掌握占位符

参考代码：type.c

注意回滚现象：

```
//char:-128~127
printf("%hhd\n", 127);
printf("%hhd\n", -128);
printf("%hhd\n", 128);
printf("%hhd\n", 129);
printf("%hhd\n", 255);
//unsigned char:0~255
printf("%hhu\n", 255);
printf("%hhu\n", 256);
```

一个应用点：linux 操作系统内核的软件定时器代码就涉及回滚现象

十一、进制转换

1.明确：计算机中数字都是存于内存中,并且这些数字在内存中都是以 2 进制的形式存储
2.明确：计算机中对数字的表示形式有四种：
2 进制,8 进制,10 进制,16 进制
不管是哪种形式,都是对同一个内存中存储的数字的不同表示形式而已,内存中的数字不会因为不同的表示形式而发生改变。
类似：
人在家中(类似 2 进制)
人在学校(类似 8 进制)
人在单位(类似 10 进制)
人在社会(类似 16 进制)
但是不管在哪里,人本质不会改变,不管是哪种进制,内存中数字不变
问：内存中既然是 2 进制表示,为何还需要 8,10,16 进制呢?
答：好看
计算机只认 2 进制，程序员只认 8,10,16 进制,看起来好看。
3.明确 bit 位定义：计算机把内存中每个字节又分 8 段,每段只能记录 0 和 1，要想把一个数字存储到内存中,必须先把这个数字分拆出若干个 0 和 1 存到字节的每个段中，每段对应的专业术语叫位或者 bit 或者 bit 位。
具体参见:bit.png
公式：
1Byte=8bit
2Byte=16bit
4Byte=32bit
8Byte=64bit
例如：
char a = 'A'; //分配 1 字节内存,并且把字符'A'对应的 ASCII 码 65 分拆 8 位存储内存中
也就是分拆出 8 个 0 或者 1 存储到内存中。
short a = 250; //分配 2 字节内存,并且把 250 分拆出 16 位,16 个 0 和 1 存储都内存中。
int a = 520; //分配 4 字节内存,并且把 520 分拆出 32 位,32 个 0 和 1 存储都内存中。
4.2 进制
(1)2 进制定义：用一组 0 和 1 来表示数字的方式简称 2 进制表示形式
例如：现在有一个 10 进制数(平常所见的数字)90(前提是 char 类型)现在要把它保存到内存中,在内存中必须以二进制的形式存储,最后得到 90 的二进制,也就是将 90 分拆成 8 位进行存储,其二进制形式为：01011101
(2)2 进制特点
(a)二进制每个数的编号：从 0 开始
例如：前提是 char 类型
高位 低位
76543210 位编号
01011010 二进制数
例如：前提是 short 类型
高位 低位

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 位编号

0 1 1 1 0 1 1 0 1 0 1 1 0 1 0 1 二进制数

(b)二进制数中每个 1 单独代表一个 10 进制数字,这个数字值是 2 的位编号次方

例如：前提是 char 类型

高位 低位

76543210 位编号

01000000 二进制数,其中第 6 位的 1 对应的 10 进制数字为：2 的 6 次方=64

(c)二进制数加 1 的时候把位编号从 0 开始的多个连续的 1 变成 0,最左边的 0 变 1 简称逢二进一。

例如：

76543210 位编号

00001111 + 1 = 00010000

00001000 + 1 = 00001001

问：如何将常见的 10 进制数,例如 250 分拆成 8 位或者 16 位或者 32 位的 2 进制呢?

如果知道了一个 2 进制,怎么算出对应的 10 进制呢?

答：需要掌握 2 进制和 10 进制转换即可。

5.2 进制和 10 进制之间的转换

(1)2 进制表示的非负数(0 和正数)转换成 10 进制就是把 2 进制数中的 1 单独换算成一个 10 进制数,最后相加即可得到 2 进制对应的 10 进制数。

例如：前提是 unsigned char 类型

高位 低位

76543210 位编号

01101101 二进制数 A

A 对应的 10 进制=2 的 6 次方+2 的 5 次方+2 的 3 次方+2 的 2 次方+2 的 0 次方

=64+32+8+4+1=109

(2)10 进制转 2 进制：

采用"除 2 取余，逆序排列"法。

具体做法是：用 2 整除十进制整数，可以得到一个商和余数；

再用 2 去除商，又会得到一个商和余数，

如此进行，直到商为小于 1 时为止，

然后把先得到的余数作为二进制数的低位，

后得到的余数作为二进制数的高位，依次排列起来。

例如：将 10 进制数 91 转 2 进制,转换步骤：

91/2 = 45 余 1

45/2 = 22 余 1

22/2 = 11 余 0

11/2 = 5 余 1

5/2 = 2 余 1

2/2 = 1 余 0

1/2 = 0 余 1

结果 91 对应的 2 进制为：01011011(不够 1 字节,高位补 0)

十进制整数 255 转二进制

如：255=11111111

255/2=127=====余 1

127/2=63═══余 1
63/2=31═══余 1
31/2=15═══余 1
15/2=7═══余 1
7/2=3═══余 1
3/2=1═══余 1
1/2=0═══余 1
例如：10 进制数 789 转 2 进制：789=1100010101
789/2=394 余 1
394/2=197 余 0
197/2=98 余 1
98/2=49 余 0
49/2=24 余 1
24/2=12 余 0
12/2=6 余 0
6/2=3 余 0
3/2=1 余 1
1/2=0 余 1
也就是要想存储 789,需要 2 字节 16 位来存储,16 位中只需 10 位,高 6 位用 0 来补
(3)负数的 10 进制和 2 进制不能直接转换,需要借助相反数(例如：-5 相反数 5)
(a)10 进制负数转 2 进制三步骤：
首先计算相反数
然后将相反数转 2 进制
最后对 2 进制取反加 1,最终得到 10 进制负数的 2 进制
例如：-14(10 进制是一个负数,数据类型为 char)
[1]计算-14 的相反数 14
[2]计算 14 的 2 进制：00001110
[3]取反(0 变 1,1 变 0)加 1：11110001+1=11110010
[4]结论：-14 的二进制:11110010
(b)明确：有符号类型数字(不加 unsigned 都是有符号)才有符号
2 进制数最"左边"的位叫做符号位
此位可以确定数字的符号(正还是负)
符号位的值为 0 表示此数字为非负(0 和整数)
符号位的值为 1 表示此数字为负数
切记：讨论符号位的前提是必须确定数据类型
例如：
10100101:前提是 char 类型,此数必然是负数
01010101:前提是 char 类型,此数必然是正数
1111000010100101:前提是 short 类型,此数必然是负数
10101010:前提是 short 类型,此数必然是正数(高 8 位的 0 没有写而已)
0101111111111111:前提是 short 类型,此数必然是正数
结论：2 进制负数转 10 进制
例如：换算 2 进制 10100101(前提是 char 类型,显然此数为负数)
[1]先取反加 1:01011010+1=01011011

[2]然后将 2 进制转 10 进制:2 的 6 次方+2 的 4 次方+2 的 3 次方+2 的 1 次方+2 的 0 次方=91

[3]最后求相反数：-91

6.2 进制和 8 进制之间的转换

(1)8 进制定义：就是把内存中的 2 进制数从右边到左边每三位分成一组，每组用 0~7 的数字替换,最终形成的表示形式简称 8 进制。

注意：8 进制数前面加 0 做区分,例如：089(8 进制数,不是 10 进制)

占位符：0%o

例如：二进制 01101010(10 进制为 106),换算出对应的 8 进制：

(a)先分组：

01 101 010

(b)换算

01=2 的 0 次方=1

101=2 的 2 次方+2 的 0 次方=5

010=2 的 1 次方=2

(c)替换：0152

重磅好消息：2 转 10,10 转 2,2 转 8,8 转 2 以后实际开发拿计算器直接转换即可!

7.2 进制和 16 进制之间的转换(核心中核心,要求口算,不允许计算器)

(1)16 进制定义：把二进制数从右边到左边每 4 位分成一组每组用一个字母替换(用 a 到 f 替换 10~15 之间的数字)。

注意：16 进制数前面加 0x 或者 0X,不用区分大小写。

占位符：%#x 或者%#X。

例如：01011010(前提是 char 类型),换算出对应的 16 进制

(a)先分组：

0101 1010

(b)换算：

0101:2 的 2 次方+2 的 0 次方=5

1010:2 的 3 次方+2 的 1 次方=10

(c)替换

5->5

10->a

(d)其 16 进制为:0x5a 或者 0X5a 或者 0x5A 或者 0X5A

(2)切记切记切记：务必拿下 2 转 16,16 转 2,必须 5 秒口算出

结论：死记硬背

16 进制	2 进制
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101(重点)
6	0110
7	0111
8	1000

9	1001
a	1010(重点)
b	1011
c	1100
d	1101
e	1110
f	1111

演练：

16 进制　　2 进制

高位　　低位

0xa87cdf53　1010 1000 0111 1100 1101 1111 0101 0011

0x5fde5e76　 0101 1111 1101 1110 0101 1110 0111 0110

十二、运算符和表达式

1.概念：

计算机程序无非最终玩内存,内存中无非就是各种数字计算出程序最终无非就是对内存中的各种数字进行数据运算。

例如：加,减,乘,除等

做这些运算,C 语言提供对应的运算符号

运算符定义：对内存中的数字进行运算的符号,例如：+,-,*,/等

表达式定义：运算符和数字结合起来的式子,例如：1+2

2.C 语言提供的运算符

(1)算术运算符：加,减,乘,除,取余:+,-,*,/,%

注意事项：

(a)如果参与除法计算的两个数字都是整型数,则计算结果只保留整数

例如：5/2=2,5f(浮点数 float)/2=2.5

(b)/和%不能对整数 0 进行操作,否则程序崩溃

例如：5/0 或者 5%0

(c)%不能对浮点数使用,编译都不通过

(d)%的结果与左边数字的符号一致

例如：-7%2=-1　7%-2=1

(e)/如果除数是浮点数,结果是 inf 表示无穷大

例如：5/0.0

案例：编写 C 程序,掌握算术运算符

参考代码：oper.c

(2)赋值运算符：=,就是将右边的数字值给左边的变量,就是修改变量对应内存的值

例如：

```
int a = 20;
a = 30;
int b, c, d;
b = c = d = 250;
```

(3)复合运算符：赋值运算符和其它运算符结合起来使用

例如：

```
a += b; 等价于 a = a + b;
```

a -= b; 等价于 a = a - b;

a *= b; 等价于 a = a * b;

a /= b; 等价于 a = a / b;

a %= b; 等价于 a = a % b;

注意事项：不能给常量(不可改变的数字)和表达式赋值

例如：

```
100=200; //gcc 报错
100=a;   //gcc 报错
a+b = c; //gcc 报错,gcc 先算 a+b 得到一个常量,不可能给常量赋值为 c
```

案例：参考代码:assign.c

(4)自增运算符:++

自减运算符:--

(a)定义:

自增运算符++就是让变量对应的内存数字加 1

自减运算符--就是让变量对应的内存数字减 1

(b)四种形式：

前++：先对变量的值加 1,后计算表达式的值

语法：++变量名;

例如：

```
int a = 1;
int b = 0;
b = ++a;
printf("a = %d, b = %d\n", a, b); //a = 2 b = 2
```

后++:先计算表达式的值,后对变量的值加 1

语法：变量名++;

例如：

```
int a = 1;
int b = 0;
b = a++;
printf("a = %d, b = %d\n"); //a = 2 b = 1
```

前--:先对变量的值减 1,后计算表达式的值

语法：--变量名;

例如：

```
int a = 2;
int b = 0;
b = --a;
printf("a = %d, b = %d\n"); //a = 1 b = 1
```

后--:先计算表达式的值,后对变量值减 1

语法：变量名--;

例如：

```
int a = 2;
int b = 0;
b = a--;
printf("a = %d, b = %d\n"); //a = 1 b = 2
```

(c)注意事项：常量不能进行自增和自减

例如：100++,100--

(5)关系运算符

(a)关系运算符类型：等于:==,不等于:!=,大于:>,小于:<,大等于:>=,小于等于: <=

(b)注意事项:

[1]关系运算符的运算结果是：1(真)或者 0(假)

[2]关系运算符不要进行连续的运算

例如：5<4<3 运算结果永远为真 1

因为 gcc 先做 5<4,运算的结果为 1 或者 0,然后 gcc 拿着 1 或者 0

再跟 3 做<运算,结果永远为真 1

问：如何进行这种连续判断呢?

答：必须采用逻辑运算符来解决

(6)逻辑运算符(真真假假,假假真真)

(a)逻辑运算符分三类:

逻辑与：&&(具有并且的意思)

逻辑或：||(具有或者的意思)

逻辑非：!(具有对着干的意思)

注意：逻辑运算中真为非 0,假为 0

(b)逻辑与&&运算符特点

语法：C=表达式 A && 表达式 B;

语义：只有当 A 和 B 都为真(非 0)时,C 的值才为真，只要 A 和 B 中有一个为假,C 就为假

例如：当用户名对了并且密码对了,才能登陆

只要有一个错误,登陆失败

登陆 = 用户名 && 密码

例如:

```
250 && 1 //表达式结果为真
250 && 0 //表达式结果为假
0 && 1 //表达式结果为假
0 && 0 //表达式结果为假
```

(c)逻辑或||运算符特点

语法：C = 表达式 A || 表达式 B;

语义：只要 A 和 B 中有一个为真,C 就为真

只有 A 和 B 都为假,C 才能为假

例如:登陆抖音可以采用微信登陆或者支付宝登陆或者手机号登陆或者 QQ 登陆只要有一个登陆成功,抖音即可登陆成功。

例如:

```
250 || 1 //表达式结果为真
250 || 0 //表达式结果为真
0 || 1 //表达式结果为真
0 || 0 //表达式结果为假
```

(d)逻辑非!运算符特点

语法：C=!A

语义：A 为假,C 为真;A 为真,C 为假

例如：!250 //结果为假

!0 //结果为真

(e)切记：短路运算(笔试题必考)

形式 1：A && B:如果 A 为假,则 B 不被执行,不处理

例如：

```
int a = 1;
0 && ++a;
printf("a = %d\n", a); //a=1
int a = 1;
250 && ++a;
printf("a = %d\n", a); //a=2
```

形式 2：A || B:如果 A 为真,则 B 不被执行,不处理

例如：

```
int a = 1;
250 || ++a;
printf("a = %d\n", a); //a = 1
int a = 1;
0 || ++a;
printf("a = %d\n", a); //a = 2
```

(7)位(bit)运算符

(a)功能：这些运算符就是专门操作内存中的二进制数

(b)四类：

位 与: & (目的：将二进制数的 bit 清 0)

位 或: | (目的：将二进制数的 bit 置 1)

位异或: ^ (目的：将二进制数的局部的某些 bit 反转:1->0 或者 0->1)

位 反: ~ (目的：将二进制数的所有 bit 反转)

(c)位与&运算符特点

语法：C = A & B

例如：

数字	2 进制	16 进制
A	01011010	0x5A
B	11100111	0xE7
&----------		
C	01000010	0x42

规律：任何数跟 0 做位与,结果为 0,任何数跟 1 做位与,保持原值

(d)位或|运算符特点：

语法：C = A | B

例如：

数字	2 进制	16 进制
A	01011010	0x5A
B	11100111	0xE7
|----------		
C	11111111	0xFF

规律：任何数跟 1 做位或,结果为 1,任何数跟 0 做位或,保持原值

(e)位异或^运算符特点：

语法：C = A ^ B

例如：

数字　　2 进制　　16 进制

A　　01011010　　0x5A

B　　11100110　　0xE6

^------------------------------

C　　10111100　　0xBC

规律：相同为 0,不同为 1,局部反转

(f)位反~运算符特点

语法：C=~A

例如：

数字　　2 进制　　16 进制

A　　01011010　　0x5A

~--------------------

C　　10100101　　0xA5

规律：0 变 1,1 变 0,全部反转

(8)移位运算符

(a)功能：将二进制数整体向左边或者向右边移动 N 个位置

(b)分两种：

左移：A<<B

语义：将 A 左移 B 个位置

右移：A>>B

语义：将 A 右移 B 个位置

例如：

```
char a = 0x5a; //a=0x5a=01011010
char b = a << 2; //b=a<<2=0x5a<<2=01011010<<2 = 01101000 = 0x68
printf("a = %#x b = %#x\n", a, b); //a = 0x5a, b = 0x68

b = a >> 2; //b = a >> 2 = 0x5a >> 2 = 01011010>>2=00010110 = 0x16
printf("a = %#x b = %#x\n", a, b); //a = 0x5a b = 0x16

a = 0xa5; //a = 0xa5 = 10100101
b = a >> 2; //b = a >> 2 = 0xa5 >> 2 = 10100101>>2=11101001=0xE9
printf("a = %#x b = %#x\n", a, b); //a = 0xa5 b = 0xe9
```

(c)移位运算符特点

[1]向左移动后右边空出来的数据用 0 来填充

0x5a << 2 = 01011010 << 2 = 01101000

[2]无符号类型数字右移时左边空出来的数据用 0 来填充

unsigned char a = 0xa5; a >> 2 = 10100101 >> 2 = 00101001

[3]有符号类型数字右移时左边空出来的数据用符号位来填充

char a = 0xa5; a >> 2 = 10100101 >> 2 = 11101001

[4]移位运算符不会修改变量本身的值

[5]只要将来有 2 的多少次方处理的代码,建议用移位操作

例如：

3*4; //垃圾代码,CPU 运行*,/的效率极低

3<<2; //高薪代码,CPU 运行移位操作效率极高

12/4; //垃圾代码

12>>2; //高薪代码

(9)位运算符和移位运算符结合(核心中核心)

(a)实际开发常见的场景:位清 0 和位置 1

(b)给出以下位操作公式：

清 0 公式：

[1]将某个数据 A 的第 n 位清 0,其它位保持不变：

A &= ~(0x1 << n); //等价于:A = A & ~(1 << n);

[2]将某个数据 A 从第 n 位开始,连续两个 bit 位清 0,其它位保持不变：

A &= ~(0x3 << n);

[3]将某个数据 A 从第 n 位开始,连续三个 bit 位清 0,其它位保持不变：

A &= ~(0x7 << n);

[4]将某个数据 A 从第 n 位开始,连续四个 bit 位清 0,其它位保持不变：

A &= ~(0xF << n);

[5]将某个数据 A 从第 n 位开始,连续五个 bit 位清 0,其它位保持不变：

A &= ~(0x1F << n);

[6]将某个数据 A 从第 n 位开始,连续六个 bit 位清 0,其它位保持不变：

A &= ~(0x3F << n);

[7]将某个数据 A 从第 n 位开始,连续七个 bit 位清 0,其它位保持不变：

A &= ~(0x7F << n);

[8]将某个数据 A 从第 n 位开始,连续八个 bit 位清 0,其它位保持不变：

A &= ~(0xFF << n);

省略...

置 1 公式：

[1]将某个数据 A 的第 n 位置 1,其它位保持不变：

A |= (0x1 << n); //等价于:A = A & ~(1 << n);

[2]将某个数据 A 从第 n 位开始,连续两个 bit 位置 1,其它位保持不变：

A |= (0x3 << n);

[3]将某个数据 A 从第 n 位开始,连续三个 bit 位置 1,其它位保持不变：

A |= (0x7 << n);

[4]将某个数据 A 从第 n 位开始,连续四个 bit 位置 1,其它位保持不变：

A |= (0xF << n);

[5]将某个数据 A 从第 n 位开始,连续五个 bit 位置 1,其它位保持不变：

A |= (0x1F << n);

[6]将某个数据 A 从第 n 位开始,连续六个 bit 位置 1,其它位保持不变：

A |= (0x3F << n);

[7]将某个数据 A 从第 n 位开始,连续七个 bit 位置 1,其它位保持不变：

A |= (0x7F << n);

[8]将某个数据 A 从第 n 位开始,连续八个 bit 位置 1,其它位保持不变：

A |= (0xFF << n);

省略...

例如：将 0x5A 第 3 位清 0,其它位保持不变,代码：

```
int a = 0x5A;
a &= ~(1 << 3); //等价于 a = a & ~(1 << 3);
```

公式推导：

[1]先做 1 << 3

0000 0000 0000 0000 0000 0000 0000 1000

[2]然后做~(1 << 3)

1111 1111 1111 1111 1111 1111 1111 0111

[3]然后获取 a

0000 0000 0000 0000 0000 0000 0101 1010

[4]最后做 a & ~(1 << 3)

0000 0000 0000 0000 0000 0000 0101 0010

[5]结果 a 等于：

0000 0000 0000 0000 0000 0000 0101 0010 //a 的第 3 位清 0

例如：将 0x56 第 1,2 位清 0,其它位保持不变,代码：

```
int a = 0x56;
a &= ~(0x3 << 1);
```

公式推导：

[1]先做 0x3<<1

0000 0000 0000 0000 0000 0000 0000 0110

[2]然后做~(0x3 << 1)

1111 1111 1111 1111 1111 1111 1111 1001

[3]获取 a

0000 0000 0000 0000 0000 0000 0101 0110

[4]最后做：a & ~(0x3 << 1)

0000 0000 0000 0000 0000 0000 0101 0000

[5]结果 a 等于：

0000 0000 0000 0000 0000 0000 0101 0000 //a 的第 1,2 位被清 0

(c)如果将来要设置 2 位或者 2 位以上,建议：先清 0 后置 1

例如：将 A 数据从第 n 位开始连续 m 位设置为 B 值,其他位保持不变:

```
A &= ~(m 个 1 << n);
A |= (B << n);
```

(10)取地址运算符&和解引用运算符*(核心中的核心)

(a)明确变量对应的内存地址特性：计算机中地址由 32 位二进制构成,也就是一个地址 32 位,4 字节。

例如：内存 0 地址对应的 16 进制：0x00000000

内存 1 地址对应的 16 进制：0x00000001

内存 250 地址对应的 16 进制：0x000000FA

...

(b)取地址运算符&:功能就是获取一个变量在内存中的首地址

占位符用%p 打印地址

取地址运算符语法格式：&变量名;

例如：

```
Int a = 250; //分配 4 字节内存空间
```

printf("变量 a 的内存首地址%p\n", &a);//打印 4 字节内存的首地址

(c)解引用运算符*功能：根据变量的首地址获取对应内存中的数据，还可以根据变量的首地址改变对应内存中的数据。

语法：*变量的首地址 结果就可以访问变量对应的内存了

例如：

```
int a = 250;
prinf("变量 a 的值是%d\n", *&a); //根据变量 a 的首地址获取里面的值
*&a = 520; //根据变量 a 的首地址向变量 a 对应的内存空间写一个新的数 520
```

(11)条件运算符

(a)语法格式：D = 条件表达式 A ? 表达式 B : 表达式 C;

语义：如果 A 为真,D 的结果等于表达式 B 的结果

如果 A 为假,D 的结果等于表达式 C 的结果

例如：

```
int a;
a = 1?2:3; //a=2
```

案例：求一个负数的绝对值

(12)运算符优先级

一句话：不闲圆括号()多,它的优先级最高

例如：

```
printf("%d\n", 2+3*2); //8
printf("%d\n", (2+3)*2); //10
int a;

a = 1 > 2 && 2 < 3;
```

实在搞不清楚>,<和&&先算哪个,干脆加圆括号

```
a = (1 > 2) && (2 < 3);

a &= ~(1 << 3);
```

如果不清楚&=和~先算哪个,干脆加圆括号

等价于：

```
a &= (~(1 << 3));
```

(13)逗号运算符：

Z =(x+=5,y++,x+y); 最后 z 的值是 x+y 的值

十三、数据类型转换：隐式转换和强制转换

1.隐式转换特点：如果表达式中不同数字的数据类型不同(例如:100,100u)，gcc 编译器自 动的先把不同的数据类型，转换成相同的数据类型最后再做运算。隐式转换分成如下三 种情况：

(1)隐式转换过程中必须把占内存小的类型转换成占内存大的类型

例如：

```
int a = 0, c; //各占 4 字节
char b = 2; //占 1 字节
c = a + b; //gcc 编译器自动将 b 的数据类型转换成 int 类型然后再跟 a 做加运算
```

(2)如果既有整形数据类型还有浮点数据类型,gcc 编译器自动将整形数据类型转换成浮点数据类型然后做运算。

(3)如果既有无符号数据类型又有有符号数据类型,gcc 编译器自动将有符号数据类型转换成无符号数据类型。

2.强制转换语法格式：目标类型变量 = (目标类型)源类型变量;

例如：

```
char a = 90;
int b = (int)a; //gcc 强制把 a 转换成 int 类型
```

注意：强制转换可能会造成数据丢失现象：

```
char a = (char)300;
printf("a = %d\n", a); //a = 44
```

所以：强制转换一般都是小转大或者相等转!

建议：实际开发代码采用强制转换,目的就一个：提高代码的可读性

例如：

```
//隐式转换
printf("sizeof(1?1:0.9)=%d\n", sizeof(1?1:0.9)); //可读性很差
```

该进采用强制转换：

```
printf("sizeof(1?1:0.9)=%d\n",
                    sizeof((double)1?(double)1:(double)0.9)); //可读性很好
```

十四、标准 C 语言编程基础

1.C 语言功能:类似人类的交流语言,C 语言实现人和计算机之间的交流除了 C 语言还有其他编程语言,JAVA 语言,Python 语言,GO 语言等。

2.C 语言背景：

1973 年由 C 语言之父丹尼斯里奇发明

1978 年发布<<the c programming language>>,C 语言正式走向世界

1989 年发布 C89 标准

1999 年发布 C99 标准

3.编写人生的第一个标准 C 语言程序

具体实施步骤如下：

```
mkdir -p /home/tarena/stdc/day02/
cd /home/tarena/stdc/day02/
vim helloworld.c 添加如下内容
/*
    以下内容都是注释内容：
  这是我的第一个 C 程序
  我很开心
*/
//这也是注释内容：以下代码就是 C 语言程序的代码
#include <stdio.h>
int main(void)
{
        printf("老丹,您放心,我会好好学习 C 语言.\n");
```

```
        return  0;
    }
```

保存退出

然后继续执行以下命令来编译 C 程序：

```
gcc  helloworld.c
ls  -lh
./a.out  //运行可执行程序 a.out,然后观察屏幕上的打印输出信息
```

4.掌握 C 语言程序编程框架(规矩,原则)

(1)C 程序涉及的文件分两种：

源文件:以.c 为后缀结尾,例如:helloworld.c

头文件:以.h 为后缀结尾,例如：stdio.h

规则：任何 C 程序必须至少有一个源文件,可以没有头文件

(2)C 程序的文件注释有两种方法：

注释：就是程序的说明信息,不参与程序的运行

(a)第一种注释方式：/*中间就是注释的内容*/

优点：可以多行注释

缺点：不能嵌套注释,例如：/*这是/*注释*/内容*/,报错

(b)第二种注释方式：//

优点：可以嵌套注释,例如：/////////这是我的注释内容

缺点：不可多行注释

(3)#include <stdio.h>:就是包含头文件 stdio.h,也就是将 stdio.h 头文件里面所有的内容拷贝到 helloworld.c 文件中。

规矩：include 前面必须加#

C 程序包含头文件的方式有两种：

#include <stdio.h>:将来让 gcc 编译器直接到 linux 系统中的/usr/include 目录中找 stdio.h,并且把里面的内容拷贝到 helloworld.c 中。

#include "stdio.h":将来让 gcc 编译器先在当前目录下找 stdio.h 文件如果找不到,linux 系统中的/usr/include 目录下找 stdio.h。

(4)切记：(a)任何 C 程序必须有一个主函数,并且名称叫 main

(b)main 函数前面必须加 int 这个关键字

(c)main 函数后面必须跟一对圆括号,圆括号里的内容暂且写 void

例如：int main(void)

(d)main 函数圆括号后面再跟一对花括号,花括号里面的内容就是 C 程序将来运行执行的语句内容。

例如：

```
int  main(void)
{
        里面的内容就是将来要执行的语句
}
```

(e)main 函数花括号里面的每条语句后面必须跟分号;

例如：

```
    int  main(void)
{
        1+1;
```

```
        2+2;
        printf("hello,world\n");
    }
```

(f)当 CPU 执行此程序时,CPU 永远从 main 函数开始依次向下执行也就是先执行:1+1,后执行 2+2,最后执行 printf 直到后面没有语句,程序结束。

(g)main 函数最后要跟一个 return,表示 CPU 执行 return 时,整个程序立马结束,return 后面跟 0 表示告诉操作系统,此程序执行很正常，return 后面如果跟非 0 表示告诉操作系统,此程序执行有点问题。

例如：

```
int  main(void)
{
        1+1;
        2+2;
        printf("hello,world\n");
    return  0;  //正常
    或者
    return  -1;  //有问题
}
```

(5)掌握人生的第一个标准 C 库函数 printf(俗称大神写好的函数,咱直接拿来用)

printf 函数语法格式：

printf("圆括号里的内容就是将来要在显示器上输出看到的内容", 数字);

printf 函数的占位符：%d(帮你占个位置,这个位置将来放一个数字)

例如：

```
printf("hello,world\n");  //仅仅在显示器上输出一个:hello,world 信息
printf("1+1  =  %d\n",  2);  //将来在显示器上输出一个:1+1  =  2
printf("%d  %d\n",  2,  3);  //将来在显示器上输出一个:2  3
printf("%d  %d  %d\n",  2,  3,  4);  //将来在显示器上输出一个:2  3
```

注意：

'\n':表示将前面的信息输出到显示器之后,再换到下一行

注意：为了能够使用大神的 printf 函数,必须包含头文件：stdio.h

5.大名鼎鼎的 gcc 编译器(发明者 GNU 软件之父)

(1)明确：C 程序的源文件 helloworld.c 是不能直接去运行的

例如：./helloworld.c

要想运行,必须通过 gcc 编译器进行编译

(2)gcc 编译器的功能：将 C 程序源文件翻译生成可执行文件,类似翻译官

(3)gcc 编译器编译程序命令格式：gcc 选项 源文件(.c)

重点掌握三个选项：

-o:后面跟要编译生成的文件名

例如：-o helloworld.o,表示将来编译生成一个新文件叫 helloworld.o

-E:预处理：可查看头文件路径

-c:只编译不链接

(4)gcc 编译器编译源文件经过三步骤：

(a)预处理：就是将#include <stdio.h>这句话要包含的头文件内容全部拷贝过来,对应的编译命令：

gcc -E -o helloworld.i helloworld.c

意思：对 helloworld.c 进行预处理,生成新文件 helloworld.i

此时 helloworld.i 已经包含了 stdio.h 全部内容

(b)只编译不链接：就是对预处理之后的源文件 helloworld.i 单独进行翻译,翻译成 CPU 能够运行的程序对应的编译命令：

gcc -c -o helloworld.o helloworld.i(或者 helloworld.c)

意义：对 helloworld.i 或者 helloworld.c 单独编译

生成 helloworld.o 文件

(c)编译：最后将大神写好的 printf 函数代码拷贝到 helloworld.o 中最后组合生成最终的可执行程序对应的编译命令：gcc -o helloworld helloworld.o。

(d)结论：编译 C 程序的三种方法：

[1]流氓法：

gcc helloworld.c //最终生成 a.out

建议：不要用此法

./a.out //运行

[2]分步法：

gcc -E -o helloworld.i helloworld.c //先预处理

gcc -c -o helloworld.o helloworld.i //只编译不链接

gcc -o helloworld helloworld.o //编译

./helloworld //运行

[3]一步到位法：

gcc -o helloworld helloworld.c

./helloworld //运行

由衷建议：前期课程学习用分步法!

案例：将上面的 helloworld.c 文件采用三种方法编译并且运行

案例：编写 C 程序,实现打印个人信息:年龄,身高,体重

实验步骤：

cd /home/tarena/stdc/day02

vim person.c 添加如下内容

```
/*打印个人姓名,年龄,身高,体重信息*/
#include <stdio.h>
int main(void)
{
        printf("我的名字是:游成伟\n");
        printf("我的年龄是:%d 岁\n", 18);
        printf("我的身高是:%d 厘米, 体重是:%d 斤\n", 173, 130);
        return 0;
}
```

保存退出

```
//分步编译程序
gcc -E -o person.i person.c
gcc -c -o person.o person.i
gcc -o person person.o
./person //运行
```

```
//一步到位编译程序
gcc -o person person.c
./person
```

十六、C 语言的流程控制

附加：字符输出函数

格式：putchar(c)

参数：c 为字符常量、变量或表达式

功能：把字符 c 输出到显示器上

反值：正常，为显示代码值

字符输入函数

格式：getchar()

功能：从键盘读一字符

反值：正常，返回读取的代码；出错或结束输入返回-1（Ctrl d）；

gets()

puts()

1.明确：C 程序是一个结构化程序,就是由顺序,分支,循环三种结构构建的单入口单出口的程序。

结论：C 程序三大结构：顺序,分支,循环

目前已经掌握了顺序结构：就是 CPU 从 main 函数开始从上往下运行

类比：

C 程序	一栋楼
顺序结构	一居室
分支结构	两居室
循环结构	三居室

2. 接下来掌握三大结构之分支结构：

(1)分支结构功能：实现多选一

(2)分支结构又分两类：

条件分支

开关分支

(3)条件分支语法形式：

形式 1：

```
if(表达式 1) {
        语句 1;
}
```

语义：如果表达式 1 为真,那么 CPU 就执行语句 1

例如：

```
int a = 1;
if(a == 1) { //中规中矩的程序
        printf("a 和 1 相等");
}
//好程序员
if(1 == a) {
        printf("a 和 1 相等");
}
```

目的：为了防止将"=="误写成"="

```
if(a=1)  //gcc 编译通过并且意思是把 1 赋值给 a,不再是判断 a 和 1 是否相等
if(1=a)  //gcc 直接报错,帮助你检测书写错误
```

形式 2：

```
if(表达式 1) {
        语句 1;
}
else {
        语句 2;
}
```

语义：如果表达式 1 位真,那么执行语句 1,否则执行语句 2

例如：

```
int a = 1;
if(1 == a) {
        printf("a 和 1 相等");
}
else {
        printf("a 和 1 不相等");
}
```

形式 3：

```
if(表达式 1) {
        语句 1;
}
else if(表达式 2) {
        语句 2;
}
...
else if(表达式 N) {
        语句 N;
}
```

语义：如果表达式 1 为真,执行语句 1

如果表达式 1 为假,执行表达式 2

如果表达式 2 为真,执行语句 2

如果表达式 2 为假,依次向下判断执行

例如：

```
int a = 1;
if(1 == a) {
        printf("a 和 1 相等");
}
else if(2 == a) {
        printf("a 和 2 相等");
}
else if(3 == a) {
```

```
            printf("a 和 3 相等");
        }
```

形式 4:

```
        if(表达式 1) {
            语句 1;
        }
        else if(表达式 2) {
            语句 2;
        }
        ...
        else if(表达式 N) {
            语句 N;
        }
        else {
            语句 M;
        }
```

语义: 如果表达式 1 为真,执行语句 1
如果表达式 1 为假,执行表达式 2
如果表达式 2 为真,执行语句 2
如果表达式 2 为假,依次向下判断执行
如果前面 N 个表达式都为假,最终执行语句 M

例如:

```
        int a = 1;
        if(1 == a) {
            printf("a 和 1 相等");
        }
        else if(2 == a) {
            printf("a 和 2 相等");
        }
        else if(3 == a) {
            printf("a 和 3 相等");
        }
        else {
            printf("谁知道 a 等于几呢.\n");
        }
```

(4)总结:

(a)配对原则: else,else if 和上面最近的 if 配对

(b)如果语句只有一条,花括号可以省略(由衷建议尽量加上)

否则必须添加

例如:

```
    if(1 == a) {
        printf("a 和 1 相等");
    }
```

等价于

```
    if(1 == a)
        printf("a 和 1 相等");
```
(c){}独占一行问题,根据公司的编码规范要求

例如：

```
if(1 == a){  //linux 操作系统开发软件开发风格
     printf("a 和 1 相等");
}
```

还可以：

```
if(1 == a)
{
     printf("a 和 1 相等");
}
```

(5)开关分支语法形式：

(a)开关分支语法：

```
switch(控制表达式) {
        case 常量表达式 1:
                语句 1;
                break;
        case 常量表达式 2:
                语句 2;
                break;
        ...
        case 常量表达式 N：
                语句 N；
                break;
        default:
                语句 M;
                break;
}
```

执行流程：当控制表达式的值和下面 case 对应的常量表达式的值相等，那么就执行对应的 case 语句,一旦遇到 break,switch...case，立马结束,如果控制表达式的值和 case 对应的常量表达式值都不相等,最终只能执行 default 对应的语句 M。

例如：

```
int a = 1;
switch(a) {
        case 1:
                printf("a 和 1 相等\n");
                break;
        case 2:
                printf("a 和 2 相等\n");
                break;
        default:
                printf("a 既不等于 1 也不等于 2\n");
                break;
```

}

(b)注意事项:

[1]控制表达式被当成一个整数来处理(int),可以是字符(字符本质就是整数)

但控制表达式不能是浮点数和字符串(例如："abc")

例如:

```
switch(5)  //可以
switch(a)  //可以,a 是一个变量
switch("abc")  //不可以
switch(5.5)  //不可以
switch('A')  //可以
```

[2]常量表达式必须是常量,例如：'5',5,2+5

不允许有可重复的常量表达式

```
int  a  =  1;
switch(a)  {
   case  1:
                ....
   case  1:  //gcc 报错
                ...
}
```

[3]如果 case 的 break 没有添加,当这个 case 成立并且运行完对应的语句之后,将会继续执行下一个 case 语句

例如:

```
int  a  =  1;
switch(a)  {
   case  1:
                printf("a 和 1 相等\n");
   case  2:
                printf("a 和 2 相等\n");
                break;
   default:
                printf("a 既不等于 1 也不等于 2\n");
                break;
}
```

结果：打印 a 和 1 相等 a 和 2 相等

[4]如果 case 或者 default 分支中定义变量,记得加{}括起来否则 gcc 编译报错。

例如:

```
int  a  =  1;
switch(a)  {
   case  1:
                printf("a 和 1 相等\n");
                break;
   case  2:  {
                     int  b  =  250;
```

```
                printf("a 和 2 相等\n");
                break;
            }
        default:{
                int c = 520;
                printf("a 既不等于 1 也不等于 2\n");
                break;
            }
    }
    printf("b = %d c = %d\n", b, c); //gcc 报错
```

结论：此时 b 和 c 只能在 case 2 和 default 中使用,除了{}就不能用

案例：给一个成绩(0~100 分),打印等级

考分	等级
90~100	A
80~89	B
70~79	C
60~69	D
其它	E

要求用 switch...case 实现

(6)总结：if...else 和 switch...case 对比：

(a)switch...case 能实现的,if...else 都能实现

(b)if...else 能实现的,switch...case 不一定都能实现,例如：浮点数和字符串

(c)switch...case 在某些场合使用起来极其繁琐,例如：

判断 0~1000 的范围,switch...case 要写 1001 个 case

而用 if 只需一条语句：if(xxx > 0 && xxx < 1000)...

(d)gcc 编译器对 swtich...case 编译时生成的代码量要比 if...else 要少

所以 switch...case 代码执行的效率要比 if...else 高

3.接下来掌握三大结构之循环结构：

(1)循环结构功能：将一组语句重复多次执行

(2)循环结构又分三类：

for 循环

while 循环

do...while 循环

(3)详解三大循环之：for 循环

(a)for 循环的语法格式：

```
for(表达式 1; 表达式 2; 表达式 3) {
        循环语句;
} //到此 for 循环结束
```

执行流程：

第一步：首先执行表达式 1(只做一次)

第二步：然后执行表达式 2，如果表达式 2 的结果为真,那么就执行循环语句，如果表达式 2 的结果为假,那么 for 循环立刻结束。

第三步：如果表达式 2 的结果为真,那么久执行循环语句,执行完毕，接着在执行表达式 3,表达式 3 执行完毕,重复第二步

(b)for 循环使用形式：

形式 1：

```
int i;
for(i = 0; i < 5; i++) {
    printf("i = %d\n", i);
}
```

注意：i 变量专业术语叫循环变量

形式 2：

```
int i = 0;
for(; i < 5; i++) {
    printf("i = %d\n", i);
}
```

形式 3：

```
int i = 0;
for(; i < 5;) {
    printf("i = %d\n", i);
    i++;
}
```

形式 4：死循环,无限循环,CPU 玩命执行循环

```
for(;;) {
    printf("hello,world\n");
}
```

按 ctrl+c 强制结束程序

```
int i = 0;
for(;;) {
    printf("i = %d\n", i);
    i++;
    if(i >= 5)
        break; //退出 for 循环
}
```

形式 5：

```
int i, j; //两个或者两个以上循环变量
for(i = 0, j = 0; i < 5 && j < 5; i++, j++){
    printf("i = %d, j = %d\n", i, j);
}
```

形式 6：

```
for(int i = 0; i < 5; i++) {
    printf("i = %d\n", i);
}
printf("i = %d\n", i); //gcc 报错,此时的 i 只能用于 for 循环中
```

形式 7：

```
for(int i = 0; i < 5; i++) {
    if(3 == i){
        continue; //中断本次循环,继续下一次循环,也就是执行表达式 3,i++
```

```
    }。
    printf("i = %d\n", i);
}
```
形式 8：如果循环语句就一条,花括号也可以省略(由衷建议不要省略)

例如：

```
for(int i = 0; i < 5; i++)
    printf("i = %d\n", i);
```

```
        提示：
        for(....) { //负责打印多少行
            for(....) { //负责打印一行有多少个*
            }
        }
```

(4)详解三大循环之：while 循环

(a)语法格式：

```
while(循环控制表达式) {
        循环语句;
}
```

执行流程：

第一步：首先执行循环控制表达式,如果为真,执行循环语句,如果为假,退出 while 循环

第二步：如果循环控制表达式为真,执行完循环语句,继续重复第一步

例如：

```
int i = 0;
while(i < 5) {
        printf("i = %d\n", i);
        i++;
}
//死循环
while(1) {
        printf("hello,world\n");
}
```

运行：

```
./while > a.txt //将 printf 打印的信息存储到 a.txt 文件中
```

(b)如果循环语句就一条,{}可以省略,建议不要这么做

例如：

```
int i = 0;
while(i++ < 5)
        printf("i = %d\n", i);
```

(4)详解三大循环之：do....while 循环

(a)语法格式

```
do {
        循环语句;
}while(循环控制表达式); //切记此处一定要加分号
```

语义：执行流程

第一步：先执行循环语句

第二步：然后执行循环控制表达式，如果为真,继续执行循环语句，如果为假,退出循环。

特点：do...while 循环至少要执行一次!

```
do {
        printf("hello,world\n");
}while(0);
```

例如：

```
int i = 0;
do {
        printf("i = %d\n", i);
        i++;
}while(i < 5);
//死循环
do {
        printf("hello,world\n");
}while(1);
```

(5)关键字：break,continue

(a)break:结束循环,如果是循环嵌套,只能结束最内层循环

例如：

```
int i = 0;
for(; i < 5; i++) {
        if(3 == i)
            break; //当 i=3 时,退出 for 循环
}
int i, j;
for(i = 0; i < 5; i++) {
        for(j = 0; j < 5; j++) {
                if(3 == j)
                    break; //当 j=3 时,退出最内层的 for 循环
        }
}
```

(b)continue:结束本次循环,继续下一次循环

十七、goto 语句

1.goto 语句的作用：可以让 CPU 跳转到任意一条语句去执行(有点严重破坏了 C 语言顺序,分支和循环结构的意味)但是在 linux 操作系统的代码中大量使用 goto 语句

2.goto 语法格式：

```
goto 标签名; //标签名就是指定要跳转的地方，标签名要符合标识符的命名规则
```

3.使用形式两种：

形式 1：

```
        标签名:
                要执行的语句;
        ...
        goto 标签名; //表示让 CPU 跳转到上面的标签名处继续执行
```

例如：

```
label:
        printf("1\n");
        printf("2\n");
goto  label;
printf("3\n");
```

形式 2：

```
goto  标签名;  //表示让 CPU 跑到下面的标签名处继续执行
...
标签名：
        要执行的语句
...
例如：
goto  label;
    printf("1\n");
label:
    printf("2\n");
```

4.为 goto 语句平反:linux 内核源码中 goto 语句经典的使用模板

需求：启动一个程序,要求分配三块内存,只要有一块内存分配失败，程序立马结束,并且结束前把之前分配成功的内存释放并且归还给 linux 操作系统

参考代码：goto1.c

十八、空语句

1.语法：仅仅包含一个;

例如：

```
int  a  =  1;
printf("a  =  %d\n",  a);
;  //此乃空语句也,虽然没有语句,但实际 CPU 处理空语句也需要时间
   //所以空语句起到了延时作用,让 CPU 在这空转一下
```

2.应用场景:用于实现一个空循环,起到延时等待作用,让 CPU 停一下

例如：

```
for(;;);  //空死循环,让 CPU 跑到这里别再往下执行了
```

例如：

```
int  i  =  100000;
for(;  i  >=  0;  i--);  //让 CPU 空转 100000 次,目的就是让 CPU 在这里稍微等待一段时间,
```

专业术语叫忙等待或者忙延时,此代码的延时时间不够精确,后期课程讲解,如果得到一个精确的延时时间。

案例：编写空循环程序,掌握 linux 的 top 命令(类似 windows 的任务管理器)

一个终端执行：./empty

另一个终端执行：top 观察 empty 这个程序的 CPU 占用率(第 9 列%CPU)

按 Q 键退出 top 命令

3.千万别做以下傻事,可悲的事情：

```
int  i;
for(i  =0;  i  <  5;  i++);  //有效的循环五次变成了空循环
        printf("i  =  %d\n",  i);
int  i  =  0;
```

```
while(i++ < 5); //有效循环瞬间变空循环,代码逻辑完全发生了改变
    printf("i = %d\n", i);
```

切记：在 while 和 for 循环中圆括号的后面的分号要注意！

十九、数组：

1.明确：计算机程序最终玩内存,玩内存的前提是先分配获取到内存,目前分配的内存的方法只有一种：定义变量。

例如：

```
int a;
int b;
int c;
...
int z;
int aa;
...
int aaa;
...
int zzzzz....;
```

问题来了,如果要定义大量数据类型一致的变量,之前定义变量,分配内存的代码显然垃圾,非常的繁琐。

问：有没有一种方法,可以分配大量内存,并且保证数据类型又一致

答：有,通过数组来实现

结论：只要将来代码中定义了大量数据类型相同的变量。此时此刻想想计算机中的数组技术！来优化之前的垃圾代码！

2.数组的定义：是计算机中分配内存的第二种方法,分配的内存能够储存,多个数据类型相同的数据。

3.数组的特点：

优点：能够分配大量的内存,不用繁琐的定义大量的变量，分配的内存是连续的。

缺点：数据类型必须相同,但是某些场合有时候不需要数据类型相同

例如：分数(浮点数 float),姓名(字符 char),年龄(unsigned char)

财富(unsigned long long)

4.数组分配内存的语法格式(定义数组的语法格式)：

元素数据类型 数组名[长度(又称数组元素个数)]

= {初始化列表(如果有多个,用逗号,分开)};

例如：

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

语义：连续分配 5 个元素的内存空间,并且每个元素的数据类型为 int，也就是每个元素占 4 字节内存空间,最终连续分类了 5*4=20 个字节的内存空间,并且每个元素的值也就是每 4 个字节的内存空间分别放数字 1,2,3,4,5，类似之前定义变量的方式：

```
int a = 1;
int b = 2;
int c = 3;
int d = 4;
int e = 5;
```

显然这种定义方式垃圾!而用数组定义让代码变得非常简洁，此时此刻务必在脑子中浮

现数组的内存分布图!

5.数组的使用特点

(1)数组分配的内存都是连续的

(2)数组名就是整个数组的首地址,等于数组第 0 个元素的首地址

(3)数组的长度又称数组元素个数,不是整个数组分配内存的大小

例如:

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

数组的长度=数组元素个数=5

数组分配内存大小=20

(4)数组下标就是数组中每个元素的编号(专业术语叫索引号 index)

数组下标从 0 开始

例如:int a[5] = {1,2,3,4,5};

数组每个元素值	下标
1	0
2	1
3	2
4	3
5	4

(5)数组中每个元素的访问时通过运算符"[]"和下标配合使用进行访问的

格式:数组元素值=数组名[数组元素对应的下标];

例如:int a[5] = {1,2,3,4,5};

第 0 个元素值=a[0]=1

第 1 个元素值=a[1]=2

第 2 个元素值=a[2]=3

第 3 个元素值=a[3]=4

第 4 个元素值=a[4]=5

打印第 3 个元素的值:printf("%d\n", a[3]);

将第 3 个元素的值 4 修改为 250:a[3] = 250;

案例:循环打印数组元素的值和修改元素的值

(6)切记切记切记:千万注意数组的越界访问(笔试题必考)

越界访问:访问了不该访问的内存

例如:int a[5] = {1,2,3,4,5};//目前操作系统给你分配了 20 个字节的内存空间存放了 5 个元素值。

```
printf("%d\n", a[4]); //合法
printf("%d\n", a[5]); //打印不存在的第 5 个元素,也就是非法的访问 20 个字节以后的内存,就是越界访问虽然没有造成程序的崩溃,此代码也是错误的代码。
a[5] = 520; //非法的修改内存,此代码不光是错,还造成了程序的崩溃
```

6.数组定义的形式:

(1)形式 1:int a[5] = {1,2,3,4,5};

(2)形式 2:int a[5] = {1,2,3}; //a[0]=1,a[1]=2,a[2]=3,a[3]=0,a[4]=0

(3)形式 3:int a[5] = {0}; //全是 0

(4)形式 4:int a[5] = {}; //全是 0

(5)形式 5: int a[5]; //只定义只分配内存但是对内存没有初始化,此时内存都是乱七八糟

的随机数,很危险

(6)形式 6：int a[] = {1,2,3,4,5}; gcc 编译器自动算出元素个数为 5 个

(7)形式 7：int a[5] = {1,2,3,4,5,6,7,8}; gcc 编译器直接忽略 6,7,8,不要

注意错误形式：int a[];gcc 迷茫了,gcc 编译报错

7.务必记住求数组元素个数(数组长度)的公式：

数组占用的内存大小=sizeof(数组名);

数组每个元素占用的内存大小=sizeof(数组第 0 个元素)

数组长度=数组元素个数=sizeof(数组名)/sizeof(数组的第 0 个元素);

例如：

int a[5];

结论：

数组首地址=a=&a[0]

数组占用的内存大小=sizeof(a)=20 个字节

数组第 0 个元素的内存大小=sizeof(a[0]) = 4 个字节

数组元素个数=sizeof(a)/sizeof(a[0])=20/4=5 个元素

8.人生的第二标准 C 库函数：scanf(知道即可)

(1)此函数的功能：从键盘上获取输入的数字并且保存到变量中

(2)此函数的使用语法：scanf("格式化列表", 变量的地址列表);

例如：

int a;

scanf("%d", &a);//运行程序的效果是 CPU 执行此函数,此时代码停止不前，等待用户从键盘输入信息,然后将输入的信息保存到变量 a 中,一回车,程序立马继续向下执行，或者

int a, b, c;

scanf("%d%d%d", &a, &b, &c); //从键盘连续输入 3 个数字分别保存到变量 a,b,c 中,每个数字中间用空格(帅哥)或者回车或者 TAB 键分开。

9.变长数组(了解即可)

定义：数组元素个数不定

例如：

int a[5]; //此数组又称定长数组

int len;

int a[len]; //此数组又称变长数组

10.多维数组(重点研究二维数组，上面讲的都是一维数组)

(1)二维数组的定义：由多个一维数组组成的数组，二维数组中每个元素是一个一维数组,二维数组的本质还是一维数组,只是将元素再次分组而已。

(2)定义二维数组的语法：

数组元素类型 二维数组名[二维数组长度][一维数组长度]={初始化表};

例如：int a[4][5];

语义：

(a)a 代表是一个包含 4 个一维数组的数组,其中每个一维都是一个包含 5 个元素的数组其中每个元素的数据类型为 int，所以 a 就是二维数组的首地址。

(b)a[0]代表 a 的第 0 个元素,而第 0 个元素又是一个一维数组

所以 a[0]就是二维数组的第 0 个一维数组的首地址

a[1]就是二维数组的第 1 个一维数组的首地址

a[2]就是二维数组的第 2 个一维数组的首地址

a[3]就是二维数组的第 3 个一维数组的首地址

(c)a[0][0]代表二维数组的第 0 个一维数组中的第 0 个元素

(d)结论：访问二维数组中的第 i 个一维数组的第 j 个元素：a[i][j]

例如：

二维数组中第 1 个一维数组的第 1 个元素的值：a[1][1]

(3)定义二维数组的形式：

形式 1：int a[3][4]; //只定义不初始化,每个元素值都一个乱七八糟的随机数

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

形式 3：int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12}; gcc 自动分配

形式 4：int a[3][4] = {0}; //全 0

形式 5：int a[3][4] = {{1,2},{0}};

形式 6：int a[][4] = {1,2,3,4,5,6,7,8,9,10,11,12}; gcc 自动算二维数组长度 3

形式 7：int a[2][3] = {{1,2},{3,4},{5,6}}; //越界,报错

注意：二维数组的长度可以省略,但是一维数组的长度不可省略

例如：

```
int a[3][4] = {{1,2,3,5},{5,6,7,8},{9,10,11,12}};
printf("%d\n", a[2][3]); //12
printf("%d\n", a[0][2]); //3
printf("%d\n", a[1][2]); //7
a[2][2] = 250; //11->250
```

(4)二维数组设计的公式：

例如：int a[2][3] = {0};

(a)a 是二维数组的首地址

(b)a[0]是二维数组第 0 个一维数组的首地址

(c)&a[0][0]是二维数组第 0 个一维数组的第 0 个元素的首地址

(d)结论：a=a[0]=&a[0][0]

(e)sizeof(a):获取整个二维数组占用的内存大小=2*3*4=24 个字节

(f)sizeof(a[0]):获取二维数组第 0 个一维数组占用的内存大小=3*4=12 个字节

(g)sizeof(a[0][0]):获取每个元素的内存大小=4

(h)sizeof(a)/sizeof(a[0]):获取二维数组的长度

(i)sizeof(a[0])/sizeof(a[0][0]):获取一维数组的长度

(j)sizeof(a)/sizeof(a[0][0]):二维数组中所有元素的个数

二十、函数(核心中的核心)

1.明确：任何 C 程序.c 源文件都包含两部分内容：

一堆的变量(包括数组)和一堆的函数组成

2.函数概念：函数就是一堆语句的组合,用于实现一些相对独立并且具有一定通用性的功能

问：为何需要函数这个技术?

答：举例子

用户的需求是：实现两个正数相加

张三写的代码：

```
vim main1.c
#include <stdio.h>
int main(void)
{
```

```
    int a;
    int b;
    int sum;
    scanf("%d%d", &a,&b);
    if(a < 0 && b < 0) {
    printf("数字有负数,请重新输入.\n");
    return -1; //让程序结束
                }
    //正式相加
    sum = a + b;
    printf("sum = %d\n", sum);
    return 0;
}
```

李四写的代码：

```
vim main2.c
#include <stdio.h>
int main(void)
{
    int a;
    int b;
    int sum;
    a = 2;
    b = 2;
    if(a < 0 && b < 0) {
        printf("数字有负数,请重新输入.\n");
        return -1; //让程序结束
            }
    //正式相加
    sum = a + b;
    printf("sum = %d\n", sum);
    return 0;
}
```

结论：main3.c,main4.c 等,发现这些文件的代码有很多重复的代码,并且这些代码本身具有一定的独立性,并且具有一定的通用性,例如：

```
if(a < 0 && b < 0) {
    printf("数字有负数,请重新输入.\n");
    return -1; //让程序结束
                }
    //正式相加
    sum = a + b;
    printf("sum = %d\n", sum);
```

只要给这些代码传递 a,b 的值,这些代码就会做判断和相加,如果每个人都重复的写这些代码,无形加大了开发的工作量。

期望：这些通用的代码只写一次,其它人,其它代码将来要想实现两个正数相加,无需编写,

只需访问这些通用的代码即可,减少开发的工作量。
问：如何实现呢?
答：采用函数技术,采用函数技术优化之后的代码如下：

```
        vim add.c
        //制造一个 add 函数
        int add(int a, int b)
        {
        if(a < 0 && b < 0) {
        printf("数字有负数,请重新输入.\n");
        return -1; //让程序结束
        }
        //正式相加
        sum = a + b;
        printf("sum = %d\n", sum);
        return sum;
    }
```

张三写 main1.c

```
 #include <stdio.h>
int main(void)
{
   int a;
   int b;
   int sum;
   scanf("%d%d", &a,&b);
   sum = add(a, b);
   return 0;
}
```

李四写 main2.c

```
   #include <stdio.h>
int main(void)
{
   int a;
   int b;
   int sum;
   a = 2, b = 2;
   sum = add(a, b);
   return 0;
}
```

3.函数特点

(1)由一条或者多条语句组成。

(2)可以重复使用。

4.函数使用三步骤(三剑客):

(1)函数声明

(a)函数声明功能：告诉 gcc 编译器,将来这个函数可以给别人或者自己使用函数声明是不

会分配内存的。

(b)函数声明的语法：extern 返回值数据类型 函数名(形参表);

建议：函数声明加 extern 关键字(理论上可以不加),主要提高代码的可读性。

例如：extern int add(int, int); 或者：extern int add(int a, int b);

语义：声明函数 add,并且将来调用此函数时可以给函数 add 传递，两个 int 类型的变量,并且 add 函数执行完毕会返回一个 int 类型的数值。

注意：形参表中的变量名可以写也可以不用写。

(c)函数声明特点：

[1]如果函数定义在函数调用之前,可以省略函数声明,否则必须声明。

[2]该声明的没有声明,那么编译器 gcc 就会给一个默认的函数声明。

形式如下：extern int 函数名(); //建议以后别这么干!

[3]函数名必须遵循标识符的命名规则。

(2)函数定义

(a)函数定义功能：就是一个函数的具体实现过程,里面会包含一堆的执行语句，将来别人或者自己可以直接使用,访问到,实实在在存在，函数定义会分配内存。

(b)函数定义语法：

返回值数据类型 函数名(形参表)

{

一堆的函数体语句;

}

例如：

```
//定义 add 函数
int add(int a, int b) //a=100,b=200
{
        int c = a + b; //c=300
        return c; //add 函数返回 300
}
int main(void)
{
        int a = 100;
        int b = 200;
        int sum;
        sum = add(a, b); //调用,使用 add 函数,add 函数的返回值 300 给 sum
}
```

(c)函数定义特点：

[1]返回值数据类型：就是函数运行完毕要给使用这个函数的代码返回一个数字,并且这个数字必然有对应的数据类型，如果函数没有返回值,返回值数据类型用 void 关键字。

例如：

```
void add(int a, int b)
{
....
}
```

[2]形参表：就是一堆的变量定义,并且这些变量只能在函数体内部使用，也就是出来花括号就不能用,如果有多个变量,用逗号,分开形参表中变量的值由使用这个函数的代码来赋

值,例如 main 函数如果函数没有形参表,用 void 关键字。

例如:

```
void  add(void)
{
  ...
 }
```

建议:形参表中变量的个数不要超过 4 个,要小于等于 4 个,否则会影响函数使用的效率。

例如:

```
int  add(int  a,  int  b,  int  c,  int  d)  //漂亮
int  add(int  a,  int  b,  int  c,  int  d,  int  e);  //效率低
```

[3]函数的返回值

如果函数需要给使用函数的代码返回一个数字,用关键字 return 进行返回。

语法:return 数值;

例如:return 200 或者 return c 或者 return a+b 等

切记切记切记:如果函数本身需要返回一个数值,而代码中恰恰又忘记了写 return 返回,gcc 将来会默认返回一个随机数(好危险),返回值的数字的类型要和函数定义时或者函数声明时,要求的返回值的数据类型保持一致。

例如:

```
char  add(int  a,  int  b)
{
      int  c  =  a  +  b;  //c=100+200=300
      return  c;//return  300;返回值要求为 char,最后发生回滚现象。
 }
```

如果函数没有返回值,可以不用写 return 或者 return 后面么都不跟,例如:

```
void  add(void)
{
      ...
      return;  //或者 return 不用写
 }
```

(3)函数调用

(a)函数调用功能:俗称使用函数,调用函数,访问函数。

(b)函数调用语法格式:接收函数返回值的变量 = 函数名(实参表);

(c)函数调用特点:

实参表:就是给函数的形参表要赋的值

例如:add(100, 200); //100,200 就是实参

或者

```
a  =  100,  b  =  200;
add(a,  b);  //a,b 就是实参
```

切记:实参表中的变量和形参表中的变量占用的内存是独立的,不一样的

各自是各自的。

(4)函数使用的形式:

形式 1:无参无返回值

```
void  函数名(void)
{
```

函数语句;
}
形式 2：有参无返回值
void 函数名(int a, int b, int c, int d) //形参个数小于等于 4 个
{
函数语句;
}
形式 3：无参有返回值
int 函数名(void)
{
函数语句;
}
形式 4：有参有返回值
int 函数名(形参) //形参个数小于等于 4 个
{
函数语句;
}
(5)return 关键字和 exit 标准库函数
return 关键字功能：用于函数返回,函数执行到这里结束返回
exit 函数功能：让程序立马结束,必须添加头文件#include <stdlib.h>
(6)切记：实参变量和形参变量对应的内存是独立的,各自是各自的，当实参给形参传递数值时,仅仅是把实参的数值拷贝一份给形参。
参见经典代码：swap.c,利用 swap 函数实现两个实参数值的交换。
问：此代码的问题如何解决呢?
答：一般采用指针来解决,后续课程讲解!
5.函数和数组的那点事(重点)
(1)明确：之前的代码都是研究函数和变量之间的关系,也就是如何通过函数来操作变量分配的内存。
问：C 语言除了定义变量分配内存,还有数组来分配内存，函数如何访问操作数组分配的内存呢?
答：利用以下公式即可让函数和数组结合起来。
(2)明确：只要获取变量或者数组的首地址,通过首地址可以任意查看或者修改对应内存的数值。
(3)函数访问数组编程公式：
/*定义访问数组的函数:关键是形参,返回值类型随意*/
void 函数名(数组的首地址, 数组的长度)
{
可以查看数组元素的值
可以修改数组元素的值
}
例如：
void array_function(int a[], int lenght)
{
查看数组元素的值：

```
        printf("%d\n", a[下标]);
        修改数组元素的值:
        a[下标] = 新值;
}
```

切记:

形参: a[]本质保存了一个数组的首地址,它不是数组,如果是数组,gcc 编译器肯定报错,既然不报错说明他不是数组,目前是可以把它当数组来用, 等后期讲完指针即可明白!

调用操作数组的函数:

```
int main(void)
{
        int arr[5] = {1,2,3,4,5};
        //第一个实参传递数组的首地址
        array_function(arr, sizeof(arr)/sizeof(arr[0]));
}
```

二十一、作用域和可见性

1.明确: C 语言变量按照数据类型分类: 12 类(char...double)

2.明确: C 语言变量按照作用域和可见性分两类: 局部变量和全局变量

3.局部变量定义: 定义在函数内部({})的变量

例如:

```
void A(void)
{
        int a = 250; //局部变量
}
```

4.全局变量定义: 定义在函数外部的变量

例如:

```
int g_a = 250; //全局变量
void A(void)
{
    ...
}
 int g_b = 520; //全局变量
```

5.static 关键字(重中之重,笔试题必考)

如果定义变量时前面加 static 关键字修饰,表示此变量为静态变量。

如果定义变量时前面没有 static 关键字修饰,表示此变量为非静态变量。

语法: static 数据类型 变量名 = 初始值;

例如:

```
int a = 250; //非静态变量
static int a = 250; //静态变量
```

6.终极结论: 最终 C 语言变量按照作用域和可见性细分变量为四种:

局部非静态变量/局部静态变量/全局非静态变量/全局静态变量

7.详解局部非静态变量特点:

形式 1:

```
void A(void)
{
```

```
        printf("a = %d\n", a); //gcc 报错,报没有定义的错误信息
        int a = 250; //定义局部非静态变量
        printf("a = %d\n", a); //可以
}
```

形式 2：

```
void A(void)
{
        if(1) {
                int a = 250; //定义局部非静态变量
                printf("a = %d\n", a); //可以
        }
        printf("a = %d\n", a); //gcc 报错,报没有定义的错误信息
}
```

形式 3：函数的形参

例如：

```
void A(int c) //c 在整个函数体内部都可以访问,出了函数失效。
```

总结局部非静态变量特点:

(1)此种变量使用范围：从定义的地方开始依次向下直到最近的花括号结束。

(2)此变量分配的内存生命周期：从定义的地方操作系统就会给变量分配内存，直到最近的花括号操作系统立马收回变量的内存,只能等下一次运行操作系统才能重新分配内存。

8.详解局部静态变量特点(钉子户)：

形式 1：

```
void A(void)
{
        printf("a = %d\n", a); //不可以
        static int a = 250; //定义局部静态变量
        printf("a = %d\n", a); //可以
}
```

形式 2：

```
void A(void)
{
        if(1) {
                static int a = 250; //定义局部静态变量
                printf("a = %d\n", a); //可以
        }
        printf("a = %d\n", a); //gcc 报错,报没有定义的错误信息
}
```

局部静态变量特点：

(1)此变量使用范围：从定义的地方开始到最近的花括号。

(2)此变量分配的内存生命周期：从启动程序开始操作系统分配内存到程序结束操作系统回收内存。

9.详解全局非静态变量特点：

形式 1：全局非静态变量的定义和访问都是在同一个文件中

例如：

```
vim  var.c
void  B(void)
{
        printf("g_a  =  %d\n",  g_a);  //不可以
}

int  g_a  =  250;  //定义全局非静态变量

void  A(void)
{
        printf("g_a  =  %d\n",  g_a);  //可以
}
```

形式 2：全局非静态变量的定义和访问在不同的源文件中实现,简称跨文件访问。

切记：如果一个源文件定义了全局非静态变量,而另一个源文件要想访问这个全局非静态变量,必须声明全局非静态变量，声明全局非静态变量的语法格式：extern 数据类型 全局非静态变量名。

结论：一旦声明,程序即可访问全局非静态变量

例如

```
vim  var2.c
int  g_a  =  250;  //此源文件定义全局非静态变量
```

保存退出

```
vim  var3.c
/*声明 var1.c 中的全局非静态变量,只有这样,var2.c 才能访问*/
extern  int  g_a;
/*定义 A 函数*/
void  A(void)
{
        printf("g_a  =  %d\n",  g_a);  //如果没有声明,gcc 编译报错,提示没有声明定义
}
int  main(void)
{
        A();
        return  0;
}
```

保存退出

编译命令：gcc -o var var2.c var3.c //将 var2.c 和 var3.c 一起编译生成 var

参考代码：var2.c 和 var3.c

二十二、作用域和可见性

1.C 语言按照作用域和可见性分：局部变量和全局变量

局部变量：函数内部定义变量

全局变量：函数外部定义变量

2.static 关键修饰的变量又分两类：静态变量和非静态变量

3.C 语言变量细分为四类：

局部非静态变量/局部静态变量/全局非静态变量/全局静态变量

4.局部非静态变量特点:
(1)使用范围：从定义的地方开始到最近的花括号结束
(2)分配的内存生命周期：从定义的地方开始分配内存到最近的花括号内存回收
5.局部静态变量特点:
(1)使用范围：从定义的地方开始到最近的花括号结束。
(2)分配的内存生命周期：从程序启动开始分配内存到程序结束内存回收,一次分配,终身使用。
6.全局非静态变量特点
形式 1：全局非静态变量的定义和使用在同一个源文件中
例如：

```
vim  var1.c
#include  <stdio.h>
/*定义函数 A*/
void  A(void)
{
        printf("%d\n",  g_a);  //gcc 报错,报没有定义错误
}
int  g_a  =  250;  //定义全局非静态变量
/*定义函数 B*/
void  B(void)
{
        printf("%d\n",  g_a);  //可以
}
int  main(void)
{
        A();
        B();
        printf("%d\n",  g_a);  //可以
        return  0;
}
```

形式 2：全局非静态变量的定义和访问跨文件(在不同文件中进行)
切记：如果一个源文件访问另一个源文件定义的全局非静态变量必须先变量的声明,声明
语法：

```
    extern  数据类型   全局非静态变量名;  //声明是不会分配内存的
vim  var2.c
#include  <stdio.h>
int  g_a  =  250;  //定义全局非静态变量
/*定义函数 print*/
void  print(void)
{
        printf("%d\n",  g_a);  //可以,打印全局变量的值
}
/*定义函数 inc_var*/
void  inc_var(void)
{
```

```
        g_a++;  //可以,让全局变量加 1
}
保存退出
vim  var3.c
include  <stdio.h>
/*定义函数 A*/
void  A(void)
{
      printf("%d\n",  g_a);  //不可以
}
/*声明 var2.c 的全局变量*/
extern  int  g_a;
/*定义函数 dec_var*/
void  dec_var(void)
{
        g_a--;
}
//声明 var2.c 的函数
extern  void  print(void);
extern  void  inc_var(void);
int  main(void)
{
        print();//调用 var2.c 的 print 函数
        inc_var();//调用 var2.c 的函数
        print();
        dec_var();  //调用自己文件的函数
        print();
        return  0;
}
保存退出
gcc  -o  var  var2.c  var3.c
```

全局非静态变量总结:

(1)此变量的使用范围,分两种情况:

在同一文件,从定义的地方开始依次向下所有函数都可以访问

在不同文件,从声明的地方开始依次向下所有函数都可以访问

(2)此变量的生命周期:从启动程序那刻操作系统就给它分配内存,程序结束操作系统自动回收对应的内存。

(3)切记切记切记:全局非静态变量实际开发中尽量少用,慎用。否则极易出现乱序访问的现象,此问题极其难与排查。如果非要使用,必须添加互斥访问代码加以保护,互斥访问的意思就是一个程序在访问全局变量时,另一个程序不允许访问,但这种互斥反而降低了代码的执行效率,加大了代码的开发难度!互斥访问的实现到第二阶段课程详解!

问:如何乱序的?

答:以自增函数为例

```
void  inc_var(void)
```

```
{
        g_a++;
}
```

前提是：有两个程序 A 和 B,都调用 inc_var 函数进行自增操作,每个程序自增 10 万次,理论上 g_a 最后结果是 20 万，实际是 g_a 小于 20 万,原因就是乱序造成!

分析：C 语言 g_a++表面看是一条语句,实际 CPU 看到的是至少三条语句,CPU 执行这三条语句步骤：

第一步：先从内存加载 load 全局变量的值到 CPU 内部

第二步：CPU 做自增运算,加 1

第三步：CPU 最后把运算结果存储 store 到内存中，当 A 程序刚执行完第一步,此时 B 程序也突然启动,B 程序的优先级高于 A 程序,A 程序立马停止运行,B 程序开始运行三步骤,显然 A 程序少一次!

7.全局静态变量特点

形式 1：全局静态变量的定义和使用在同一个源文件中

例如：

```
vim  var1.c
#include  <stdio.h>
/*定义函数 A*/
void  A(void)
{
        printf("%d\n",  g_a);  //gcc 报错,报没有定义错误
}
static  int  g_a  =  250;  //定义全局静态变量
/*定义函数 B*/
void  B(void)
{
        printf("%d\n",  g_a);  //可以
}
int  main(void)
{
        A();
        B();
        printf("%d\n",  g_a);  //可以
        return  0;
}
```

形式 2：全局静态变量的定义和访问跨文件(在不同文件中进行),行不通,不能进行跨文件访问。

结论：static 修饰的全局变量只能用于当前文件,其它文件禁止访问!

```
vim  var4.c
#include  <stdio.h>
staic  int  g_a  =  250;  //定义全局静态变量
/*定义函数 print*/
void  print(void)
{
        printf("%d\n",  g_a);  //可以,打印全局变量的值
```

```
}
/*定义函数 inc_var*/
void  inc_var(void)
{
        g_a++;  //可以,让全局变量加 1
}
保存退出
vim  var5.c
#include  <stdio.h>
/*声明 var4.c 的全局变量*/
extern  int  g_a;
/*定义函数 dec_var*/
void  dec_var(void)
{
        g_a--;   //不允许访问
}
//声明 var4.c 的函数
extern  void  print(void);
extern  void  inc_var(void);
int  main(void)
{
        print();//调用 var4.c 的 print 函数
        inc_var();//调用 var4.c 的函数
        print();
        dec_var();  //调用自己文件的函数
        print();
        return  0;
}
保存退出
gcc  -o  var  var4.c  var5.c
```

全局静态变量总结：

(1)此变量的使用范围：只能在同一文件使用,从定义的地方开始依次向下所有函数都可以访问。

(2)此变量的生命周期：从启动程序那刻操作系统就给它分配内存，程序结束操作系统自动回收对应的内存。

(3)切记切记切记：全局静态变量实际开发中尽量少用,慎用，否则极易出现乱序访问的现象,此问题极其难与排查，如果非要使用,必须添加互斥访问代码加以保护，互斥访问的意思就是一个程序在访问全局变量时，另一个程序不允许访问,但这种互斥反而降低了代码的执行效率,加大了代码的开发难度！但是他出现乱序访问的概率要比全局非静态变量要低！互斥访问的实现到第二阶段课程详解！

问：如何乱序的?

答：以自增函数为例

```
void  inc_var(void)
{
```

```
        g_a++;
    }
```

前提是：有两个程序 A 和 B,都调用 inc_var 函数进行自增操作,每个程序自增 10 万次,理论上 g_a 最后结果是 20 万，实际是 g_a 小于 20 万,原因就是乱序造成！

分析：C 语言 g_a++表面看是一条语句,实际 CPU 看到的是至少三条语句,CPU 执行这三条语句步骤：

第一步：先从内存加载 load 全局变量的值到 CPU 内部

第二步：CPU 做自增运算,加 1

第三步：CPU 最后把运算结果存储 store 到内存中，当 A 程序刚执行完第一步,此时 B 程序也突然启动,B 程序的优先级高于 A 程序,A 程序立马停止运行,B 程序开始运行三步骤,显然 A 程序少一次！

(4)static 关键字终极总结(笔试题必考)

(a)static 修饰的全局变量只能用于当前文件,其它文件不能访问

例如：

```
static int g_a = 250;
```

(b)static 修饰的函数只能在当前文件使用,其它文件不可访问此函数

例如：

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

(c)static 在某些场合可以降低乱序访问的概率

二十三、指针(C 语言的灵魂)

1.指针的定义：指针本质就是一个变量,只是这个变量永远只能保存一个内存地址,所以此变量对应的专业术语叫指针变量,简称指针。通过指针保存的地址就可以对内存进行读查看和写修改，而指针指向的内存保存一个数字,而这个数字具有一个对应的数据类型。

2.指针变量定义的语法格式

书写形式 1：

```
int  *  指针变量名;
```

例如：int * pa; //定义一个指针变量

书写形式 2：

```
int* 指针变量名;
```

例如：int* pa; //定义一个指针变量

书写形式 3：

```
int *指针变量名;
```

例如：int *pa; //定义一个指针变量

语义：都是用来定义一个指针变量,将来这个指针变量 pa 能够保存一块内存区域的首地址,并且这块内存区域保存着一个 int 类型的数据，又由于指针变量也是变量,同样也需要分配内存，而他分配的内存保存着一个地址。

问：指针变量占用的内存空间多大呢?

答：这个跟计算机硬件相关，32 位系统,一个地址值 32 位,4 字节，64 位系统,一个地址值 64 位,8 字节。

结论：指针变量分配的内存要不是 4 字节要不是 8 字节,所以指针变量本质没有数据类型一说！只是它指向的内存区域保存的数字有数据类型,所以 int 不是给指针变量 pa 使用,而是

给指针变量指向的内存区域使用的，说明这块内存区域将来存储 4 字节数据

例如：

```
char *pa; //将来 pa 指向的内存区域能够保存 1 个字节数据
```

连续定义指针变量形式：

```
int *pa, *pb; //定义两个指针变量
int *pa, pb; //pa 是指针变量,pb 就是一个普通变量
```

3.指针变量的初始化是通过取地址运算符&进行：

例如：

```
int a = 250; //分配 4 字节内存空间保存 250
int *pa = &a; //定义指针变量,同样分配 4 字节内存,保存着变量 a 的首地址
char b = 'B'; //分配 1 字节内存空间保存'B'的 ASCII 码
char *pb = &b; //定义指针变量,同样分配 4 字节内存,保存着变量 b 的首地址
short c = 520; //分配 2 字节内存空间保存'B'的 ASCII 码
short *pc = &c;//定义指针变量,同样分配 4 字节内存,保存着变量 c 的首地址
```

务必脑子浮现内存的分布图,参见：指针.png

问：一旦通过指针变量获取到指向的内存区域的首地址,如何通过指针变量来访问指向的内存区域呢,实现读查看和修改呢?

答：通过解引用运算符：*

4.解引用运算符*:

功能：通过指针变量对指向的内存区域进行读查看和写修改

语法格式：*指针变量名;

例如：

```
char a = 'a';
char *pa = &a;
printf("a = %c\n", *pa); //查看打印变量 a 的值
*pa = 'b'; //修改变量 a 的值为'b'
```

切记：sizeof(指针变量名) = sizeof(pa) = 永远 4 字节或者 8 字节

5.特殊指针：空指针和野指针

(1)空指针：指针变量保存一个空地址,空地址用 NULL 表示,其实就是地址为 0，空指针不可随意访问,否则造成程序的崩溃!

例如：

```
int *pa = NULL; //pa 指向 0 地址,pa 保存空指针
printf("pa 指向的 0 地址储存的数据是%#x\n", *pa);
*pa = 250; //向空地址 0 写入 250
```

(2) 野指针：如果指针变量是一个局部的指针变量并且没有初始化,此指针变量保存着一个随机地址,可能指向一块无效的内存区域,因为这块内存区域操作系统没有给你分配,你就没有权利访问，如果访问就是非法访问,会造成程序的崩溃!

（如果全局变量没有初始化，数据为 0，局部的没有初始化，数据为一个随机数）

例如：

```
int *pa; //没有初始化,此时保存着一个随机地址,此指针变量又称野指针
printf("pa 指向的随机地址存储的数据是%#x\n", *pa);
*pa = 250; //向野指针指向的内存写入数据
```

(3)实际开发代码编程规范(公式)：

如果定义一个指针变量,一开始不清楚到底指向谁,千万不能不初始化，否则变成野指针,

所以建议要求初始化为空指针 NULL,一旦初始化为 NULL,将来程序后面使用指针变量时,记得要时时刻刻判断指针变量是否安全，只需判断指针变量是否为 NULL 即可,如果为 NULL,说明此指针不能访问，如果为 NULL,让程序可以结束,让函数可以中途返回,如果不为 NULL,说明此指针变量保存一个有效地址,方可安全访问。

例如：

```
int *pa; //不建议这么写,危险
//安全写法,假设 pa 指向 NULL：
int *pa = NULL; //初始化为空指针
if(NULL == pa) {
        printf("pa 为空指针,程序不能访问.\n");
        return -1 或者 exit(0);
} else {
        printf("pa 指向有效的内存区域,程序可以踏实访问.\n");
        printf("%d\n", *pa); //读查看
        *pa = 250; //写修改
}
//安全写法,假设 pa 指向有效内存：
int *pa = NULL; //初始化为空指针
int a = 520;
pa = &a; //pa 指向 a,也就是保存有效内存的首地址
if(NULL == pa) {
        printf("pa 为空指针,程序不能访问.\n");
        return -1 或者 exit(0);
} else {
        printf("pa 指向有效的内存区域,程序可以踏实访问.\n");
        printf("%d\n", *pa); //读查看
        *pa = 250; //写修改
}
```

重要结论：如果程序中一个指针不再适用,记得最后将这个指针赋值为 NULL;

6.指针运算(核心并且坑爹)

(1)指针可以和一个整数做加减法运算,简称地址运算,指针运算

切记：计算结果和指针指向的变量数据类型有关

(2)指针计算公式：

(a)char 型指针加 1,计算的结果是实际地址+1

例如：

```
char *pa = 0x1000; //假设 pa 指向 0x1000 这个地址
pa++; //pa 指向 0x1001 这个地址
```

(b)short 型指针加 1,计算的结果是实际地址+2

```
short *pa = 0x1000; //假设 pa 指向 0x1000 这个地址
pa++; //pa 指向 0x1002 这个地址
```

(c)int 型指针加 1,计算的结果是实际地址+4

```
int *pa = 0x1000; //假设 pa 指向 0x1000 这个地址
pa++; //pa 指向 0x1004 这个地址
```

7.指针和数组的那点事儿(之前都是研究指针和变量的那点事儿)

(1)回顾数组

定义数组：int a[4] = {1,2,3,4};

立马脑子浮现内存分配图

结论：

(a)数组名是数组的首地址,所以数组名本质也是一个指针

只是此指针不能改变,不像:

```
int a = 250;
int *pa = &a;
int b = 520;
pa = &b;
```

(b)同样遵循指针计算

数组首地址=数组第 0 个元素地址=a = &a[0] = a + 0

数组第 1 个元素的地址：a + 1 = &a[1]

数组第 2 个元素的地址：a + 2 = &a[2]

数组第 3 个元素的地址：a + 3 = &a[3]

结论：&a[3] - a = 3 表示这两个地址之间相差 3 个元素,实际地址相差 12 字节

(c)切记"[]"运算符 gcc 编译器对它要经过两步运算：

例如：int a[4] = {1,2,3,4};

a[2]意义是：

[1]先求第 2 个元素地址：a + 2

[2]根据地址取出值：*(a + 2)

结论：

第 0 个元素的值=a[0]=*(a+0)=1

第 1 个元素的值=a[1]=*(a+1)=2

第 2 个元素的值=a[2]=*(a+2)=3

第 3 个元素的值=a[3]=*(a+3)=4

(1)指针变量和数组的公式：

```
int a[4] = {1,2,3,4};
int *pa = a; //定义指针变量 pa 保存数组的首地址,pa 指向数组，后面如何通过 pa 访问操作数组跟通过 a 来访问操作数组一样的。
```

注意：a 不能变,名称固定的,地址固定,但是 pa 能变

(a)通过指针变量访问数组的写法 1：

```
int a[4] = {1,2,3,4};
int *pa = a;
int lenght = sizeof(a)/sizeof(a[0]);
//读查看
for(int i = 0; i < lenght; i++) {
        printf("%d, %d\n", pa[i], *(pa+i));
}
//写修改
for(int i = 0; i < lenght; i++) {
        *(pa + i) *= 10;
        pa[i] *= 10;
}
```

(b)通过指针变量访问数组的写法 2:

```
int a[4] = {1,2,3,4};
int *pa = NULL;
int lenght = sizeof(a)/sizeof(a[0]);
//读查看
for(pa = a; pa < a + lenght; pa++) {
        printf("%d\n", *pa);
}
//写修改
for(pa = a; pa < a + lenght; pa++) {
        *pa *= 10;
}
```

(c)注意:

求数组元素个数,数组长度数组公式:sizeof(a)/sizeof(a[0])

但是不能是:sizeof(pa)/sizeof(pa[0])

注意:*pa++:先算*pa,后算 pa++

注意:指针变量相减得到的是元素的个数,实际的地址相差个数*数据类型

(d)终极公式:a[i] = *(a+i) = *(pa+i) = pa[i]

二十四、常量,常量指针,指针常量,常量指针常量:围绕 const 关键字(笔试题必考)

1.常量定义:不可修改的值,例如:250,'A'等

2.const 关键字功能:将变量常量化,四种形式:

(1)const 可以修饰普通变量,一旦修饰该变量,该变量以后就当常量处理, 即其值一经初始化不可再改变。

例如:

```
const int a = 250; //将变量 a 常量化
a = 200; //gcc 编译器报错
```

(2)常量指针:不能通过指针变量来修改指向的内存区域的数据主要目的保护数据不可篡改

例如:

```
int a = 250;
const int *pa = &a; //定义初始化一个常量指针
```

或者

```
int a = 250;
int const *pa = &a; //定义初始化一个常量指针
*pa = 200; //gcc 编译器报错
printf("a = %d\n", *pa); //可以查看
int b = 300;
pa = &b; //pa 重新指向 b 变量,可以修改指针变量本身保存的地址
*pa = 400; //gcc 编译器报错
printf("b = %d\n", *pa); //可以查看
```

(3)指针常量:指针永远指向一块内存区域,不能再指向别的内存,但是可以修改内存的值。

例如:

```
int a = 100;
int* const pa = &a; //定义指针常量
*pa = 300; //可以
```

```
int b = 200;
pa = &b; //gcc 报错
```

(4)常量指针常量：指针本身和指向的内容都不能修改

例如：

```
int a = 100;
const int* const p = &a;
*p = 200; //报错
int b = 300;
p = &b; //报错
printf("a = %d\n", *p);
```

二十六、指针和函数那点事儿(目前掌握了指针和变量的关系,指针和数组的关系)

1.指针作为函数的形参

结果：函数通过指针可以访问指向的内存区域

参考代码：昨天的 swap.c 和 bit.c

2.指针作为函数的返回值

结果：此类函数又称指针函数,也就是函数的返回值是一个指针,

将来函数执行完毕返回一个地址

切记：千万要注意别返回一个野指针(注意：变量的内存生命周期,不要返回局部变量)

指针函数定义语法格式：

```
返回值数据类型 *函数名(形参表)
{
        函数体语句;
}
```

例如：

```
int g_a = 250;//定义全局非静态变量
static int g_b = 250; //定义全局静态变量
//定义一个指针函数
int *A(void)
{
        int a = 250; //定义局部非静态变量
        static int b = 250; //定义局部静态变量
        ...
        return &a; //结果返回一个野指针,因为函数返回变量 a 的内存无效了
                   //返回的 a 的地址也是无效的
        或者
        return &b; //可以,因为变量 b 的内存是一次分配终身使用
        或者
        return &g_a; //可以
        或者
        return &g_b; //可以
}
int main(void)
{
        //调用函数 A 并且用 p 来接收函数 A 返回的地址
```

```
    int *p = A();
    *p = 3000;
    printf("%d\n", *p);
    return 0;
}
```

3.函数,指针,数组

```
void A(int a[], int lenght)
```

等价于

```
void A(int *p, int lenght)
```

```
#include<stdio.h>
static void print1(int a[],int len)
{
    for(int i = 0;i<len;i++){
        a[i]+=1;
        printf("a[%d]=%d\n",i,a[i]);
    }
}
static void print2(int *p,int len)
{
    for(int i = 0;i<len;i++){
        p[i]+=1;
        printf("p[%d]=%d\n",i,p[i]);
    }
}
int main()
{
    int a[] = {1,2,3,4,5};
    int len = sizeof(a)/sizeof(a[0]);
    print1(a,len);
    print2(a,len);
    return 0;
}
```

二十五、无数据类型指针:void *(核心)

1. 无数据类型指针的概念:它也是一个指针变量,也保存一个地址,同样占用4字节内存空间,只是它指向的内存区域的数据类型是不可知的,称这种指针为无数据类型指针。

写法:void *

例如:

```
int a = 250;
void *p = &a; //CPU 通过 p 是不知道 a 的数据类型
```

2.无数据类型指针特点:

(1)通过此种指针变量时无法获取指向内存区域的数据类型

(2)不能直接对此种指针进行解引用操作*,因为你不知道他指向的内存区域的数据类型,也就不知道将来要获取几个字节的数据,如果要想通过无数据类型指针来获取内存的数据必须做数据类型的转换(建议采用强制转换,目的提高代码的可读性)。

例如：

```
int a = 250;
void *p = &a;
printf("a = %d\n", *p); //gcc 报错,gcc 很迷茫
*p = 520; //gcc 报错
 int *p1 = (int *)p; //将无类型指针 p 强转为 int 类型的指针变量
 printf("a = %d\n", *p1); //此时可以,gcc 知道将来要操作 4 字节数据
 *p1 = 520; //此时可以,gcc 知道将来要操作 4 字节数据
```

还可以直接强转：

```
printf("a = %d\n", *(int *)p);
*(int *)p = 520;
```

(3)无类型指针 void *加减几,实际的地址就是加减几，有类型指针加减几,实际的地址加减几跟类型相关。

例如：

```
void *p = 0; //假设 p 等于 0
p++; //p=1
p++; //p=2
```

3.经典指针操作案例

需求：

```
int a = 0x12345678;
int *p = &a;
printf("a = %#x\n", *p); //一次性获取 4 字节数据并且按照 16 进制打印输出
```

务必画出内存分布图,参见:void.pn

问：以上代码是一次性操作 4 字节内存数据,如何做到操作访问 4 字节中的某 1 个字节或者某 2 个字节数据呢?

答：操作方式两种：

方式 1：采用有数据类型指针来获取 1 字节,2 字节数据

例如：

```
//获取 1 字节
int a = 0x12345678;
char *p = (char *)&a; //将 a 的 int 类型指针强转为 char 类型的指针
printf("%#x\n", *p++); //0x78
printf("%#x\n", *p++); //0x56
printf("%#x\n", *p++); //0x34
printf("%#x\n", *p++); //0x12
//获取 2 字节
int a = 0x12345678;
short *p = (short *)&a; //将 a 的 int 类型指针强转为 short 类型的指针
printf("%#x\n", *p++); //0x5678
printf("%#x\n", *p++); //0x1234
```

方式 2：采用无类型指针 void*来获取 1 字节,2 字节,4 字节

```
int a = 0x12345678;
void *p = &a;
//1.获取 1 字节
```

```
char *p1 = (char *)p;
printf("%#x\n", *p1++); //0x78
printf("%#x\n", *p1++); //0x56
printf("%#x\n", *p1++); //0x34
printf("%#x\n", *p1++); //0x12
printf("#x\n", *(char *)(p+0));
printf("#x\n", *(char *)(p+1));
printf("#x\n", *(char *)(p+2));
printf("#x\n", *(char *)(p+3));
//2.获取 2 字节
short *p2 = (short *)p;
printf("%#x\n", *p2++); //0x78
printf("%#x\n", *p2++); //0x56
printf("#x\n", *(short *)(p+0));
printf("#x\n", *(short *)(p+2));
```

二十七、字符串相关内容

1.回顾字符常量：用单引号括起来,例如：'A','B','0'等，实际内存储存的是对应的整型数,ASCII 码，占位符：%c,%hhd,%hhu,内存连续的

2.字符串定义：由一组连续的字符组成,并且用双引号""括起来,并且最后一个字符必须是'\0',此字符'\0'又称字符串的结束符，表示字符串的结尾,此'\0'的 ASCII 码为 0

注意：'0'的 ASCII 码是 48

例如："abcd\0"或者"abcd"(其中'\0'可以省略)

3.字符串特点：

(1)字符串的占位符是%s：string

例如：printf("%s\n", "abcd\0"); //可以直接跟字符串

或者

printf("%s\n", 字符串的首地址); //可以直接跟字符串的首地址

(2)字符串占用的内存空间是连续的,一个字符挨着一个字符

具体参见字符串.png 图

一旦碰到'\0'字符串结束!

printf("%s\n", "abcd\0efg"); //输出 abcd

(3)多个并列的字符串最终 gcc 会把它合并成一个字符串

例如："abc""efg"->"abcdefg"

printf("%s\n", "abc""efg"); //输出 abcdefg

4.字符串和指针的那点事儿

(1)明确：研究字符串最终研究的是字符串中的每个字符,而每个字符的数据类型,要不是 char,要不是 unsigned char,都是占用 1 字节内存空间

(2)定义一个字符指针变量并且指向一个字符串,本质就是字符指针变量，保存字符串的首地址,也就是保存着字符串中第 0 个字符的首地址，定义并且初始化字符指针变量的形式：

```
char *p = "abcd";
```

p 保存着字符串"abcd"的首地址,等于其中字符'a'的首地址

(3)如果让一个字符指针变量指向一个字符串,其中字符串的'\0'可以不用写

gcc 编译器自动帮你添加'\0'

(4)切记切记(笔试题必考)：不能通过字符指针变量修改字符串中每个字符的值，只能通过字

符串指针变量查看字符的值。

例如：

```
char *p = "abcd";
printf("%s\n", p); //打印输出字符串
*(p + 2) = 'C'; //错误,不允许
printf("%c\n", *p++); //a
printf("%c\n", *p++); //b
printf("%c\n", *p++); //c
printf("%c\n", *p++); //d
```

5.字符串和数组的那点事儿,两种写法：

写法 1：

```
char a[] = {'a', 'b', 'c', 'd', '\0'};
```

注意：a 即可认为是一个普通的数组,数组每个元素是一个字符，也可以认为 a 是一个字符串,如果 a 是一个字符串,记得最后必须手动添加'\0',如果不添加,a 仅仅就是一个普普通通的数组而已。

写法 2：

```
char a[] = "abcd";
```

注意：a 必然是一个普通的数组,也必然是一个字符串,gcc 编译器自动帮你添加'\0'

切记切记：不管是哪种写法,字符数组中的元素都可以修改(笔试题必考)

例如：

```
a[2] = 'C'; //可以
```

或者

```
*(a + 2) = 'C'; //可以
```

6.字符串处理相关的标准 C 库函数(都是大神编写好的函数,咱直接调用即可)

(1)这些函数功能：便于程序员处理字符串

注意：为了使用这些字符串,需要添加头文件：#include <string.h>

为了做函数的声明

(2)strlen 函数：获取字符串中有效字符的长度,不包含结束符'\0'

(3)strcat 函数：字符串拼接函数

(4)strcmp 函数：字符串比较函数

(5)strcpy 函数：字符串拷贝覆盖函数

(6)sprintf 函数：格式化输出函数(游哥最爱)

(7)strtoul 函数：字符串转整形函数(游哥最爱)

参考代码：string1.c

```
1 //字符串操作函数
2 #include<stdio.h>
3 #include<string.h>
4 int main(void)
5 {    //1.strlen 函数:显示有效的字符串长度
6     printf("字符串长度%d\n",strlen("abc"));
7     //函数里面是字符串的首地址
8     char *p = "abc";
9     printf("字符串长度%d\n",strlen(p));
10    //函数里面是字符串的首地址
```

```
11      //2.strcat 函数演示：字符串拼接函数
12      char a[10] = "abc";
13      char *p1 = NULL;
14      p1 = strcat(a,"xyz");//接收函数是指针，a 代表数组,
15                                  // 意思是把"xyz"拼到数组 a 里
16      printf("%s %s\n",p1,a);
17      printf("%d %d\n",sizeof(a),strlen(p1));
18      //3.strcmp 字符串比较函数
19      int ret = 0;
20      ret = strcmp("abc","abd");
21      printf("ret = %d\n",ret);//-1 说明后面大
22      ret = strcmp("abd","abc");
23      printf("ret = %d\n",ret);//1 说明前面大
24      ret = strcmp("abc","abc");
25      printf("ret = %d\n",ret);//0
26      //第二种写法
27      char *p2 = "abc";
28      char *p3 = "abd";
29      ret = strcmp(p2,p3);
30      //-1 "abc"<"abd"
31      printf("ret = %d\n",ret);
32     //4.strcpy 函数    字符串拷贝覆盖函数
33      char b[10] = "abc";
34      char *p4 = NULL;
35      p4 = strcpy(b,"xyz");//把字符串 xyz 拷贝到数组 b 中，并且 p4=b
36      printf("%s %s\n",p4,b);//strcpy 函数里面 b 为数组，接收函数是指针
37      char *p5 = "hello";
38      p4 = strcpy(b,p5);//把 p5 指向的字符串拷贝到 b 中
39      printf("%s %s\n",p4,b);
40     //5.sprintf 格式化输出函数
41     char c[100] = {0};
42     sprintf(c,"友哥：%d,成绩：%c,分数%g,体重%d 公斤",18,'A',99.9,65);
43     //将这些数字按照格式保存到字符数组 c 中
44     printf("%s\n",c);//最终都变成了字符串
45      return 0;
46 }
```

二十八、指针数组(比较常用)

1.明确:[]运算符的计算步骤,两步运算：

例如：

```
int a[3] = {1,2,3};
int *pa = a;
printf("%d %d\n", a[2], pa[2]);
```

a[2]实际的计算是：

(1)先求地址

a + 2
或者
pa + 2

(2)后取内容

*(a + 2)
或者
*(pa + 2)

2.指针数组定义：数组中每个元素都是一个指针(地址),每个元素只能是地址,不能是普通数据。

3.定义指针数组的语法：

数据类型 *数组名[数组长度/元素个数] = {地址列表};

例如：

```
int a = 10, b = 20, c = 30;
int *p[3] = {&a, &b, &c};
```

结果：p[0] = &a; p[1] = &b; p[2] = &c;

等价：*(p+0) = &a; *(p+1)=&b; *(p+2)=&c;

取值：*p[0]= 10; *p[1]=20;*p[2]=30

等价：**(p+0)=10;**(p+1)=20;**(p+2)=30;

元素个数=sizeof(p)/sizeof(p[0]);

应用场合：将来如果要定义大量同类型的指针变量时,要想起采用指针数组的形式简化代码。

例如：

```
int a = 1, ......;
int *pa = &a; ......;
```

相当类啊,定义了一堆的变量和指针变量

此时采用指针数组优化：

```
int *p[] = {&a, ....};
```

二十九、字符指针数组

定义：字符指针数组中每个元素是一个字符串的首地址

例如：

```
char *p[2] = {"abc", "efg"};
```

或者

```
char *p1 = "abc";
char *p2 = "efg";
char *p[2] = {p1, p2};
```

结果：

p[0] = p1 = "abc" = 第一个字符串的首地址

p[1] = p2 = "efg" = 第二个字符串的首地址

三十、预处理(核心)

1.回顾：C 程序编译三步骤

(1)预处理:替换,编译命令：gcc -E -o xxx.i xxx.c

(2)只编译不链接:仅仅编译自己,编译命令：gcc -c -o xxx.o xxx.i/xxx.c

(3)链接:包含其他大神的代码,编译命令：gcc -o xxx xxx.o

2.C 语言预处理指令(以#开头,不跟;分号)的形式分类

(1)头文件包含指令：#include,又分两类：

(a)#include <stdio.h>

语义：告诉 gcc 编译器直接到/usr/include 目录下找 stdio.h 头文件,找到之后在预处理阶段将头文件的内容拷贝到源文件进行替换。

(b)#include "stdio.h"

语义：告诉 gcc 编译器先在当前目录下找 stdio.h 头文件，如果找不到再去/usr/include 目录下找 stdio.h，如果找到在预处理阶段将头文件的内容拷贝到源文件进行替换。

(c)问：目前 gcc 只能在/usr/include 或者当前目录下找头文件，那么如果头文件不再这两个目录下,gcc 怎么找呢?

答：如果头文件在其他目录下,只需添加-I 选项来指定头文件所在的路径即可，建议绝对路径(核心,实际开发必用)

预处理命令格式：gcc -E -o xxx.i xxx.c -I /home/tarena/stdc/day10/

(2)宏定义指令：#define

优点：提高代码的可移植性,将来让代码改起来方便

建议：宏的名称用大写(小写也可以),续行用\

(a)宏定义指令又分两类：常量宏和参数宏(宏函数)

[1]常量宏

语法格式：#define 宏名 (值)

例如：#define PI (3.14)

语义：在预处理阶段 gcc 将程序中所有的宏 PI 替换成 3.14

案例：参考代码 circle.c

编译命令：gcc -E -o circle.i circle.c //预处理并且生成预处理文件

vim circle.i //跑到文件的最后查看 PI 是否替换成了 3.14

gcc -o cirecle cirecle.i

./circle

[2]参数宏(宏函数)

语法格式：#define 宏名(宏参数 1,宏参数 2,...宏参数 N) (宏值)

注意：宏值里面的宏参数不要少圆括号

如果宏参数有多个,用逗号分开

例如：#define SQUARE(x) ((x)*(x))

#define ADD(x, y) ((x) + (y))

语义：gcc 在预处理阶段,gcc 先将宏参数替换成实际的值,然后将代码中所有的宏名最终全部替换成宏值

案例：参考代码:define.c

编译命令：gcc -E -o define.i define.c

vim define.i 查看替换的结果

gcc -o define define.i

./define

[3]问：宏函数和普通的函数有何区别

答：

宏函数优点：代码执行速度快,替换了

宏函数缺点：不便于程序的跟踪调试

普通函数缺点：代码执行速度慢,函数传递参数需要拷贝

普通函数优点：便于代码的跟踪调试,因为有调用过程,既有工具 gdb 跟踪调试,而宏函数做不到

linux 系统下的 C 语言还有一个关键字:inline,就是体现了这两点:

例如:

```
inline void A(int a, int b)
{
        ....
}
```

{1}当代码进行优化编译时,例如添加-O2 选项(优化),
inline 修饰的函数 A 瞬间变成宏函数

{2}当代码进行调试编译时,例如添加-g 选项
inline 修饰的函数 A 就是函数

3.#(转字符串指令)和##(粘贴指令)

(1)#作用:将其后面跟的宏参数转换为字符串

例如:#N 最终 gcc 在预处理阶段会替换成"N"

参考代码:define.c

(2)##作用:先将后面的宏参数进行替换成实际的值,然后在与前面的部分粘在一起,最终作为宏值替换宏名

参考代码:define.c

```
1  /*参数宏,宏函数:大神最爱*/
2  //gcc -E -o define.i define.c
3  //vim define.i
4  //gcc -o define define.i
5  //./define
6  #include <stdio.h>
7  //定义宏函数 SQUARE:求平方
8  #define SQUARE(x)  ((x) * (x)) //别忘记添加圆括号
9  //定义宏函数 ADD:求两个数相加
10 #define ADD(x, y)    ((x) + (y))
11 //定义宏函数 SUB:求两个数相减
12 #define SUB(x, y)    ((x) - (y))
13 //定义宏函数 CLEAR_BIT:将某个数的某个 bit 位清 0
14 #define CLEAR_BIT(data, n)  ((data) &= ~(1 << (n)))
15 //定义宏函数 SET_BIT:将某个数的某个 bit 位置 1
16 #define SET_BIT(data, n)  ((data) |= (1 << (n)))
17 //定义求圆周长宏函数
18 #define  PI  (3.14)
19 #define CALP(r)       (2*PI*(r))
20
21 //玩玩#:公司代码的打印将来不在使用 printf,利用宏函数将 printf 函数封装,让代码调用简洁
22 //按 10 进制整型数打印
23 #define PRINT(N)      printf(#N"=%d\n", N)
24
25 //玩玩##
26 #define ID(N)          id##N
```

```
27
28 int main(void)
29 {
30     //宏函数
31     printf("%d\n", SQUARE(10));
32     printf("%d\n", SQUARE(3+7));
33     printf("%d\n", ADD(3, 7));
34     printf("%d\n", SUB(3, 7));
35     int a = 0x55, n = 0;
36     CLEAR_BIT(a, n);
37     printf("a=%#x\n", a); //a=01010101->01010100->0x54
38     SET_BIT(a, n);
39     printf("a=%#x\n", a); //a=01010100->01010101->0x55
40
41     //#转换
42     int b = 10, c = 20;
43     PRINT(b); //替换结果:printf("b""=%d\n", b) = printf("b = %d\n", b);
44     PRINT(c); //替换结果:printf("c""=%d\n", c) = printf("c = %d\n", c);
45
46     //##黏贴
47     int ID(1) = 100, ID(2) = 200; //替换结果:int id1 = 100, id2 = 200;
48     printf("%d %d\n", ID(1), ID(2));
49
50     //gcc 预定义的宏
51     printf("这里错误了:%s,%s,%s,%s,%d,"
52             "出现了一个野指针的访问错误.\n",
53             __DATE__, __TIME__, __FILE__,
54                 __FUNCTION__, __LINE__);
55     return 0;
56 }
```

公司代码:

```
1 /*define 宏函数终极使用参考代码*/
2 #include <stdio.h>
3
4 #define MAX_FUNC(TYPE)  \
5     TYPE TYPE##_max(TYPE a, TYPE b) { \
6         return a > b ? a : b; \
7     }
8
9 MAX_FUNC(int) //int int_max(int a, int b)
10                 //{
11                 //  return a > b ? a : b;
12                 //}
13 MAX_FUNC(double) //double double_max(double a, double b)
```

```
14                          //{
15                          // return a > b ? a : b;
16                          //}
17
18 #define MAX(TYPE)     TYPE##_max
19
20 int main(void)
21 {
22         printf("%d\n", MAX(int)(100, 200)); //int_max(100,200)
23         printf("%d\n", MAX(double)(10.1, 12.2)); //double_max(10.1,12.2)
24         return 0;
25 }
```

4.编译器已经定义好的宏(直接使用,不用自己#define),实际开发必用!

注意：以下宏都是前后两个下划线

__FILE__:代表当前文件名,占位符%s

__LINE__:代表当前行号,占位符%d

__FUNCTION__:代表当前函数名,占位符%s

__DATE__:文件创建日期,%s

__TIME__:文件创建时间,%s

例如：实际开发常用于 log 日志的记录!

```
printf("这里错误了:%s,%s,%s,%s,%d,出现了一个野指针的访问错误.\n",
        __DATE__, __TIME__, __FILE__, __FUNCTION__, __LINE__);
```

5.用户预定义宏：通过 gcc 的-D 选项指定

作用：程序在预处理的时候或者编译的时候给程序传递一个数值

注意：如果宏是一个字符串,那么-D 后面需要用\"转义

例如：gcc -DSIZE=5 -DWELCOME=\"达内\"

语义程序编译时给程序传递一个 SIZE=5 和 WELCOME="达内"宏

优点：让程序变得灵活

参考代码：define2.c

```
gcc -o define2  define2.c -DSIZE=5 -DWELCOME=\"达内\"
```

```
1 /*预定义宏*/
2 #include <stdio.h>
3 //编译命令:gcc -o define2 define2.c -DSIZE=5 -DWELCOME=\"达内教育\"
4 //运行:./define2
5 int main(void)
6 {
7         printf("SIZE=%d, WELCOME=%s\n", SIZE, WELCOME);
8
9         int a[SIZE] = {0};
10
11        for(int i = 0; i < SIZE; i++) {
12                a[i] += i;
13        }
14
```

```
15      for(int i = 0; i < SIZE; i++) {
16          printf("a[%d] = %d\n", i, a[i]);
17      }
18      return 0;
19 }
```

6.条件编译命令(用的非常多)

#if //如果
#ifdef //如果定义了...
#ifndef //如果没有定义...
#elif //否则如果...
#else //否则...
#endif //必须和#if 或者#ifdef 或者#ifndef 配对使用
#undef //取消定义,和#define 死对头,例如#define PI (3.14) #undef PI

参考代码：if.c

```
1 /*条件编译预处理指令*/
2 #include <stdio.h>
3
4 int main(void)
5 {
6       //1.#if 演示
7       //编译命令:gcc -E -o if.i if.c 然后 vim if.i 然后 gcc -o if if.i 然后./if
8       //编译命令:gcc -DA=1 -E -o if.i if.c;gcc -DA=1 -o if if.i; ./if
9 #if A==1
10      printf("1.\n");
11 #endif //跟#if 配对
12
13      //2.#if ... #else 演示
14      //编译命令:gcc -E -o if.i if.c;vim if.i;gcc -o if if.i; ./if
15      //编译命令:gcc -DB=1 -E -o if.i if.c;vim if.i;gcc -DB=1 -o if if.i
16 #if B==1
17      printf("2.\n");
18 #else
19      printf("3.\n");
20 #endif
21      //3.#ifdef 或者 ifndef ...#else 演示
22      //编译命令:gcc -E -o if.i if.c;vim if.i;gcc -o if if.i; ./if
23      //编译命令:gcc -DC -E -o if.i if.c;vim if.i;gcc -DC -o if if.i
24 //#ifdef C
25 #ifndef C
26      printf("4.\n");
27 #else
28      printf("5.\n");
29 #endif
30      //4.if defined...#else 演示
```

```
31        //编译命令:gcc -E -o if.i if.c;vim if.i;gcc -o if if.i; ./if
32        //编译命令:gcc -DD -E -o if.i if.c;vim if.i;gcc -DD -o if if.i; ./if
33        //编译命令:gcc -DE -E -o if.i if.c;vim if.i;gcc -DE -o if if.i; ./if
34        //编译命令:gcc -DD -DE -E -o if.i if.c;vim if.i;gcc -DD -DE -o if if.i; ./if
35  #if defined(D)
36        printf("6.\n");
37  #elif !defined(D) && !defined(E)
38        printf("7.\n");
39  #else
40        printf("8.\n");
41  #endif
42        return 0;
43  }
```

实际开发的演示代码：

利用条件编译可以让一套代码支持不同厂家不同架构的CPU(X86,ARM,POWERPC,MIPS,DSP,FPGA 等)

```
void A(void)
{
        #if ARCH==X86
              ...//支持 X86 的底层代码
        #elif ARCH==ARM
         ...//支持 ARM 的底层代码
         #elif ARCH==POWERPC
              ...//支持 POWERPC 底层代码
         ...
        #else
         printf("请重新配置源码,指定 ARCH.\n");
        #endif
}
```

只需在编译的时候指定 CPU 类型即可,例如：

```
gcc -DARCH=ARM ....
```

三十一、大型程序实现

1.掌握：头文件卫士

(1)结论：以后编写任何头文件必须遵循以下规范：

```
vim A.h 添加
#ifndef __A_H //头文件卫士,宏名一般跟文件名同名
#define __A_H
中间部分就是头文件要包含的内容
#endif
```

头文件卫士功能：防止头文件包含的内容重复定义

(2)问：头文件卫士__A_H 如何起到防止重复定义呢?

答：分析原理(了解即可)

例如：

//没有添加头文件卫士的情形：

vim a.h 添加
int a = 250; //定义变量
保存退出
vim b.h 添加
#include "a.h"
保存退出
vim main.c 添加
#include <stdio.h>
#include "a.h"
#include "b.h"
int main(void)
{
	printf("a = %d\n", a);
	return 0;
}
保存退出
gcc -E -o mani.i main.c
vim main.i
结果:
int a = 250;
int a = 250; //重复定义
gcc -o main main.i //gcc 报错重复定义了
问: 如何解决呢?
答: 添加头文件卫士
vim a.h 添加
#ifndef __A_H
#define __A_H
int a = 250; //定义变量
#endif
保存退出
vim b.h 添加
#ifndef __B_H
#define __B_H
#include "a.h"
#endif
保存退出
vim main.c 添加
#include <stdio.h>
#include "a.h"
#include "b.h"
int main(void)
{
	printf("a = %d\n", a);
	return 0;

```
}
保存退出
gcc -E -o mani.i main.c
vim main.i
结果：
int a = 250;
gcc -o main main.i

预处理时展开:main.c
//#include "a.h" 展开
#ifndef __A_H   //一开始定义__A_H 了吗? 么有
#define __A_H   //么有定义,成立,定义__A_H
int a = 250; //定义变量
#endif
//#include "b.h" 展开
#ifndef __B_H //一开始定义__B_H 了吗? 么有
#define __B_H //么有定义,成立,定义__B_H
//#include "a.h" 展开
#ifndef __A_H   //不成立,因为前面已经定义了,后面内容消失不见
#define __A_H
int a = 250; //定义变量
#endif
#endif
int main(void)
{
        printf("a = %d\n", a);
         return 0;
}
```

2.实际开发产品代码组成部分：三部分

(1)代码开发原则：实际产品代码不可能一个源文件横扫天下,所以产品需要根据功能将源文件分拆出多个源文件。

(2)代码组成三部分：

(a)头文件部分：负责变量和函数的声明(不会分配内存)，将来别的源文件只需包含此头文件即可做到变量和函数的声明。

例如：dog.h,duck.h,chicken.h

(b)源文件部分：负责变量和函数的定义(会分配内存)

例如：dog.c,duck.c,chicken.c

(c)主文件部分：负责统领源文件和头文件,主要调用源文件定义的变量和函数，主文件包含 main 函数。

例如：main.c

(d)建议：一个源文件对应一个头文件(dog.c->dog.h...)，并且源文件包含自己对应的头文件。

例如：

vim dog.c

#include "dog.h"
...

(e)建议：如果头文件中有公共的信息,最好再提炼出来单独放在一个头文件中,而其他头文件包含这个功能的头文件即可。

例如：两只眼睛(共性)

vim common.h

两只眼睛

vim dog.h

#include "common.h"

vim duck.h

#include "common.h"

vim chicken.h

#include "common.h"

(f)注意 static

案例：编写大型程序,实现两个数相加和相减

vim add.h

vim add.c

vim sub.h

vim sub.c

vim main.c

//分步编译

gcc -E -o add.i add.c

gcc -E -o sub.i sub.c

gcc -E -o main.i main.c

gcc -c -o add.o add.i

gcc -c -o sub.o sub.i

gcc -c -o main.o main.i

gcc -o main main.o sub.o add.o

./main

//一步到位

gcc -o main main.c sub.c add.c

三十二、大型程序编译

1.问：如果产品设计的源文件有 3 万个.c 源文件,采用目前的编译方式

编译命令：gcc -o xxx 1.c 2.c 3.c ... 3 万个

结论：此种编译方式极其变态,极其繁琐

如何简化编译方式呢

答：必须采用 Makefile 来实现

2.Makefile 功能：本质就是一个文本文件,它能够制定一个编译规则，将来让 gcc 编译器根据这个规则对源文件进行编译 Makefile 文件给 make 命令使用,也就是当执行 make 命令时,Makefile 会别 make 命令执行,然后启动 gcc 再根据 Makefile 里的规则进行编译。

3.Makefile 制定编译规则的语法格式：

目标：依赖 1 依赖 2 依赖 3 ... 依赖 N
(TAB 键)编译命令 1
(TAB 键)编译命令 2
....
(TAB 键)还可以是其它命令
语义：利用编译命令将依赖编译生成目标
注意：命令是从上往下按个执行
注意：Makefile 文件的注释使用#
例如：利用 Makefile 来编译 helloworld.c
写法 1：一步到位
vim Makefile 添加如下内容
#制定编译规则
helloworld:helloworld.c
    gcc -o helloworld helloworld.c
写法 2：分步
vim Makefile 添加如下内容
#制定编译规则 1
helloworld:helloworld.o
    gcc -o helloworld helloworld.o
#制定编译规则 2
helloworld.o:helloworld.c
    gcc -c -o helloworld.o helloworld.c
执行 make 命令,即可运行 Makefile,启动 gcc 根据里面的编译规则进行编译
案例：利用 Makefile 编译 helloworld.c
具体实施步骤：
mkdir -p /home/tarena/stdc/day11/Makefile1/
cd /home/tarena/stdc/day11/Makefile1/
vim helloworld.c
vim Makefile
make //开始编译
ls -lh //观察时间信息
./helloworld
make //再次编译,提示 update to data:helloworld 文件已经是最新了
vim helloworld.c //随意修改代码
ls -lh //看时间信息
make //再次编译
ls -lh //观察时间信息
rm helloworld helloworld.o
make //再次编译

4.Makefile 工作原理

当执行 make 命令时,make 首先在当前目录下找 Makefile，一旦找到,make 命令打开 Makefile 文件,然后罗列出所有的编译规则,最终确定最终的目标是 helloworld,最终的源文件 helloworld.c，然后 make 命令首先在当前目录下找最终的目标 helloworld 是否存在：

(1)如果存在,然后再检查最终的源文件 helloworld.c 的时间戳是否要比最终的 helloworld

要新,如果新说明源文件修改过,重新执行编译命令重新编译,如果时间戳旧,那就提示文件最新,无需编译。

(2)如果发现 helloworld 不存在,执行编译命令进行编译

案例：利用 Makefile 将昨天的大型程序进行编译

实施步骤：

mkdir /home/tarena/stdc/day11/Makefile2/

cd /home/tarena/stdc/day11/Makefile2

把昨天大型程序的文件拷贝到 Makefile2 目录下

vim Makefile

make //编译

./main

5.Makefile 小技巧：

如果将来有 1 万个.c 源文件,如果制定规则：

```
xxx.o:xxx.c
        gcc -c -o xxxx.o xxxx.c
```

这类代码量极其庞大

如何简化呢?

答：必须采用以下技巧,只需两句话横扫所有源文件

```
%.o:%.c
    gcc -c -o $@ $<
```

说明：

%.o:所有的目标

%.c:所有的源文件

$@:目标文件

$<:源文件

案例：将上一个案例的 Makfile 优化

参考代码：Makefile3

cp -fr /home/tarena/stdc/day11/Makfile2 /home/tarena/stdc/day11/Makfile3

cd /home/tarena/stdc/day11/Makfile3

vim Makefile //修改

rm *.o main //删除所有的.o 文件和 main 文件

make

./main

三十三、复合类型之结构体(核心中的核心)

1.明确：目前 C 程序分配内存的方法两种：定义变量和定义数组

定义变量缺陷：不能大量分配内存,最大 8 字节，所以大量分配采用数组

定义数组缺陷：数据类型一致,有些场合需要数据类型不一致

例如：描述学生信息：

```
int sexy; //1:男性,0:女性
char name[10]; //姓名
float score; //成绩
```

问：如何做到即可大量分配内存,也可以保证数据类型不一致呢?

答：采用结构体,结构体可以保证数据类型不一致,但是分配大量内存，还是需要定义数组,无形之中发现结构体还是有点瑕疵。

2.结构体特点：

(1)能够包含大量变量,并且这些变量的数据类型可以不一致

(2)对应的关键字：struct

(3)信念：结构体也是一种数据类型,就是程序员自己定义的一种数据类型，把它类比成 int 类型,int 类型怎么用,它基本就怎么用!

(4)结构体中每个成员/字段变量分配的内存都是连续的,一个变量挨着一个变量

3.结构体数据类型声明,定义的使用方法：

(1)形式 1：直接定义结构体变量(很少用)

(a)语法格式：

```
struct {
        结构体成员或者又称结构体字段;
}结构体变量名;
```

(b)例如：用结构体描述一个学生的基本信息

```
struct {
        int age; //年龄
        int id; //学号
        char name[20]; //姓名
        float score; //学分
}student1; //会分配内存
//再定义一个学生的基本信息
struct {
        int age; //年龄
        int id; //学号
        char name[20]; //姓名
        float score; //学分
}student2; //会分配内存
```

(c)缺陷：每次定义一个结构体变量,结构体成员都需要重新写一次,非常烦躁!

(1)形式 2：先声明一个结构体数据类型,然后用这个结构体数据类型定义结构体变量(常用)

(a)声明结构体数据类型语法：

```
struct 结构体名 {
        结构体成员;
};
```

注意：

[1]不会分配内存

[2]如果是大型程序,声明代码放在头文件

[3]类比：把 struct 结构体名类比 int 类型

(b)用声明好的结构体数据类型定义结构体变量语法：

```
struct 结构体名 结构体变量名;
```

注意：

[1]会分配内存

[2]如果是大型程序,定义代码放在源文件

[3]例如：描述学生信息

```
//第一步：先声明一个结构体数据类型,描述学生的信息
```

```
struct student {
        int age; //年龄
        int id; //学号
        char name[20]; //姓名
        float score; //学分
};
//第二步：定义两个结构体变量分别描述两个学生信息
struct student student1; //一个学生的结构体变量
struct student student2; //另一个学生的结构体变量
或者
struct student student1, student2;
```

[4]缺陷：每次定义结构体变量,都要写一堆 struct student 这两个单词，如何将这两个单词所见呢,类似喊一个人的姓名太长了,干脆喊的外号。

(3)方式 3：先用 typedef 关键字对声明的结构体数据类型取别名(外号)，然后用别名定义结构体变量,此种方式又分三种形式：(必用)

(a)必须掌握 typedef 关键字

```
功能：给数据类型取别名
语法：typedef 原来的数据类型  别名;
例如：对基本数据类型区别名(实际开发的代码)：
        typedef   char   s8; //s=signed,有符号 8:8 位
        typedef   unsigned char u8; //u=unsigned,无符号
        typedef   short s16;
        typedef   unsigned short u16;
        typedef   int s32;
        typedef   unsigned int u32;
        typedef   float f32;
        typedef   double f64;
        用法：
        unsigned int a = 250;
        等价于：
        u32 a = 250;
```

(b)typedef 给声明的结构体取别名形式 1：

```
注意：别名后面加_t
语法：
typedef struct {
    结构体成员
}结构体别名_t;
结果是将来定义结构体变量只能用别名定义；
例如：
typedef struct {
        int age; //年龄
        int id; //学号
        char name[20]; //姓名
        float score; //学分
```

}stu_t;

(c)typedef 给声明的结构体取别名形式 2：

语法：

typedef struct 结构体名{

结构体成员

}结构体别名_t;

结果是将来定义结构体变量即可用全名 struct 结构体名还可以用别名

例如：

```
typedef struct student{
        int age; //年龄
        int id; //学号
        char name[20]; //姓名
        float score; //学分
}stu_t;
```

(d)typedef 给声明的结构体取别名形式 3：

先声明结构体数据类型：

struct 结构体名{

结构体成员

};

然后用 typedef 对声明的结构体数据类型取别名：

语法：typedef struct 结构体名 别名_t;

例如：

先声明结构体数据类型：

```
struct student{
        int age; //年龄
        int id; //学号
        char name[20]; //姓名
        float score; //学分
};
```

然后取别名：

typedef struct student stu_t;

(e)不管采用哪种方式取别名,如果用别名定义结构体变量形式都是一样滴：

定义结构体变量语法：别名 结构体变量;

例如：

stu_t student1;

stu_t student2;

或者

stu_t student1, student2;

4.结构体变量的初始化方式,两种初始化方式：

(1)传统初始化方式

语法格式：struct 结构体名 结构体变量名 = {初始化值,用逗号，分开};

或者

结构体别名 结构体变量名 = {初始化值,用逗号,分开};

例如：

//定义结构体变量并且初始化变量
struct student student1 = {18, 666, "游哥", 99.9};
或者
stu_t student1 = {18, 666, "游哥", 99.9};
结果：
age=18,id=666,name="游哥",score=99.9
缺陷：必须全部初始化并且按照顺序初始化，因为有些场合有些成员是不需要初始化的
问：如何不用按照顺序也不用全部初始化呢,只初始化感兴趣的成员呢?
答：采用结构体的标记初始化方式

(2)标记初始化方式
语法格式：struct 结构体名 / 别名 结构体变量 = {
.成员名 = 初始值,
.成员名 = 初始值,
...
};
例如：
```
struct student student1 = {
        .name = "游哥",
        .age = 18,
        .id = 666
}; //gcc 把没有初始化的字段一律给 0
```
或者：
```
stu_t student1 = {
        .name = "游哥",
        .age = 18,
        .id = 666
}; //gcc 把没有初始化的字段一律给 0
```
优点：不用按照顺序,也不用全部初始化!

5.结构体成员(字段)的访问：读查看,写修改,两种形式访问：

(1)通过"."运算符来访问结构体成员
语法：结构体变量名.结构体成员; //将来就可以获取到成员的值
例如：
```
stu_t student1 = {
        .name = "游哥",
        .age = 18,
        .id = 666
};
//读查看:打印游哥学生信息：
printf("%s %d %d\n", student1.name, student1.age, student1.id);
//写修改：
strcpy(student1.name, "猛哥");
student1.age = 20;
student1.id = 888;
```

(2)通过"->"运算符来间接访问结构体变量的成员

语法：结构体指针变量名->结构体成员; //将来就可以获取到成员的值

例如：

```
//定义初始化一个结构体变量
stu_t student1 = {
        .name = "游哥",
        .age = 18,
        .id = 666
};
//定义初始化一个指针变量 p 指向 student1 变量,保存 student1 变量的地址
stu_t *p = &student1;
//读查看:打印游哥学生信息：
printf("%s %d %d\n", p->name, p->age, p->id);
//写修改：
strcpy(p->name, "猛哥");
p->age = 20;
p->id = 888;
```

脑子务必浮现内存分布图!

结论：结构体的首地址等于结构体第一个成员的首地址

也就是：切记以下公式

p = &student1 = &p->age

= p->name - 4/sizeof(数据类型)

= &p->id - 0x18/sizeof(数据类型)

(3)结构体变量之间可以直接整体赋值

例如：

```
stu_t student1 = {
        .name = "游哥",
        .age = 18,
        .id = 666
};
stu_t student2 = student1; //整体赋值
```

或者：

```
stu_t *p = &student1;
stu_t student2 = *p; //通过指针整体赋值
```

(4)结构体数组

```
stu_t student[2] = {
    {18, "关羽", 67},
    {19, "张飞", 76}
};
```

或者

```
stu_t student[] = {
    {
        .age = 18,
        .name = "关羽",
        .score = 67
```

```
    },
    {
      .age = 19,
       .name = "张飞",
       .score = 76
    }
};
//打印
for(int i = 0; i < sizeof(student)/sizeof(student[0]); i++) {
        printf("%d %s %g\n",
                student[i].age, student[i].name, student[i].score);
}
//通过结构体指针来访问结构体数组
stu_t *p = student; //p 指向结构体数组
//打印
for(int i = 0; i < sizeof(student)/sizeof(student[0]); i++) {
        printf("%d %s %g %d %s %g\n",
                p[i].age, p[i].name, p[i].score,
                (p+i)->age, (p+i)->name, (p+i)->score);
}
```

结论：结构体指针变量加减 1,实际地址会加减一个结构体占用内存大小

(5)结构体嵌套:结构体成员还是一个结构体

两种嵌套方式：

(a)嵌套结构体变量

例如：

```
 //声明描述学生生日信息的结构体数据类型
typedef struct birthday {
        int year; //年
        int month; //月
        int day; //日
}birth_t;
//声明描述学生信息的结构体数据结构
typedef struct student {
        char name[20]; //姓名
        int age; //年龄
        birth_t birth; //生日
}stu_t;
//定义初始化一个结构体变量描述关羽同学
stu_t student1 = {"关羽", 18, {2001, 2, 2}};
```

或者：

```
stu_t student1 = {
        .name = "关羽",
        .age = 18,
        .birth = {
```

```
                .year = 2001,
                .month = 2,
                .day = 2
        }
};
```

(b)嵌套结构体指针变量

例如：

```
 //声明描述学生生日信息的结构体数据类型
typedef struct birthday {
        int year; //年
        int month; //月
        int day; //日
}birth_t;
//声明描述学生信息的结构体数据结构
typedef struct student {
        char name[20]; //姓名
        int age; //年龄
        birth_t *pbirth; //生日
}stu_t;
//定义关羽学生生日的结构体变量
birth_t birth = {2001, 2, 2};
//定义初始化一个结构体变量描述关羽同学
stu_t student1 = {"关羽", 18, &birth};
```

或者：

```
stu_t student1 = {
        .name = "关羽",
        .age = 18,
        .pbirth = &birth //pbirth 指向 birth 结构体变量
};
```

6.结构体和函数的那点事儿：

(1)结构体变量作为函数的形参：

```
void A(stu_t student)
{
        printf("%s %d %d\n", student.name, student.age,student.score);
        student.age++; //只改变了形参的值 age=19
}
int main(void)
{
        stu_t student1 = {"关羽", 18, 55.5};
        A(student1);
        printf("%s %d %d\n", student.name, student.age,student.score);
        //实参 age 没有改变,因为形参就是实参的一个拷贝而已
        return 0;
}
```

结论：结构体变量作为函数的形参,函数只能查看结构体变量的值，无法通过函数修改实参,关键代码执行效率极低,拷贝结构体需要开销。

(2)结构体指针变量作为函数的形参：

```
void B(const stu_t *p)
{
        p->age++; //gcc 报错,不允许修改
}
void A(stu_t *p)
{
        printf("%s %d %d\n", p->name, p->age,p->score);
        p->age++; //实参发生改变
}
int main(void)
{
        stu_t student1 = {"关羽", 18, 55.5};
        A(&student1);
        printf("%s %d %d\n", student.name, student.age,student.score);
        //实参 age=19 改变
        return 0;
}
```

结论：函数的形参是结构体指针,将来函数通过指针可以任意修改实参,如果将来不想让函数通过指针修改实参,添加 const，关键是代码执行效率极高,因为拷贝只拷贝了一个指针,4 字节。

(3)结构体变量作为函数的返回值：

```
stu_t A(void)
{
        static stu_t student1 = {....};
        return student1; //切记不能返回局部非静态变量,只能是局部静态或者全局
}
int main(void)
{
        stu_t student1 = A(); //将 A 函数的返回值赋值给 student1
        printf("%s %d %d\n", student.name, student.age,student.score);
        return 0;
}
```

结论：函数返回一个结构体的代码执行效率同样低下,拷贝结构体需要开销

(4)结构体指针作为函数的返回值：

```
stu_t *A(void) //指针函数
{
        static stu_t student1 = {....};
        return &student1; //切记不能返回局部非静态变量,只能是局部静态或者全局
}
int main(void)
{
```

```
    stu_t *p = A(); //将 A 函数的返回值赋值给 p,返回了地址
    printf("%s %d %d\n", p->name, p->age,p->score);
    return 0;
}
```

结论：代码执行效率高

7.结构体内存对齐问题(笔试题必考)

(1)明确：gcc 编译器对结构体成员编译时,默认按照 4 字节对齐,超过 8 字节一律按 4 字节对齐。

例如：

```
struct A {
    char buf[2];
    int val;
};
```

求：sizeof(struct A) = 8

立刻脑子浮现内存分布图,内存对齐.png

(2)终极结构体内存对齐参考代码

```
/*结构体内存对齐演示*/
#include <stdio.h>
//声明结构体数据类型 A
struct A {
    char buf[2]; //4 字节
    int val; //4 字节
};
//声明结构体数据类型 B
struct B {
    char c; //4
    short s[2]; //4
    int i; //4
};
#pragma pack(1) //让 gcc 强制从这个地方开始后面代码按照 1 字节对齐方式编译
//声明结构体数据类型 C
struct C {
    char c; //1
    short s[2]; //4
    int i; //4
};
#pragma pack() //让 gcc 到这里再恢复到 4 字节对齐方式,也就是后面的代码继续 4 字节对齐
//声明结构体数据类型 D
struct D {
    int i; //4
    char c; //4
};
//声明结构体数据类型 E
struct E {
    double d; //8
```

```
    char c; //4
};
int main(void)
{
    printf("sizeof(struct A) = %d\n", sizeof(struct A)); //8
    printf("sizeof(struct B) = %d\n", sizeof(struct B)); //12
    printf("sizeof(struct C) = %d\n", sizeof(struct C)); //9
    printf("sizeof(struct D) = %d\n", sizeof(struct D)); //8
    printf("sizeof(struct E) = %d\n", sizeof(struct E)); //12
    return 0;
}
```

终极结论：

1.明确结构体将来在内存中的分步

2.会做笔试题

如果以后出现：

```
struct A {
    char buf[2];
    int val;
};
```

求：sizeof(struct A) = 答案参见背面

3.以后将来写结构体注意内存对齐问题,一般来说都是按照 4 字节对齐

```
//糟糕代码
struct A {
    char buf[3];
    int val;
};
//优秀代码,提高代码的可读性
struct A {
    char buf[3];
    char reserved; //预留
    int val;
};
```

三十三、复合类型：联合体

1.特点

(1)它和结构体的使用语法一模一样,只是它的关键字用 union

(2)联合体中所有的成员是共用一块内存,优点节省内存

(3)联合体占用的内存按照成员中占内存最大的来分配

例如：

```
union A {
    char a;
    short b;
    int c;
};
```

sizeof(union A) = 4;

(4)初始化问题

union A a = {8}; //gcc 默认给第一个成员:a = 8;

union A a = {.c = 8}; //强制给 8

注意：a,b,c 的地址都是一样的!

(5)经典笔试题

(1)明确：CPU 的大端和小端

(a)X86(intel 处理器)架构的 CPU 是小端模式：

数据的低位放在内存的低地址,数据的高位放在内存的高地址

例如：

int a = 0x12345678;

内存条

低地址 高地址

0-----1------2------3------4------------ //地址

0x78 0x56 0x34 0x12

(b)POWERPC 架构的 CPU 是大端模式：

数据的低位放在内存的高地址,数据的高位放在内存的低地址

例如：

int a = 0x12345678

内存条

低地址 高地址

0-----1------2-----3-----4 -------------//地址

0x12 0x34 0x56 0x78

(2)编写一个程序获取当前处理器是 X86 架构还是 POWERPC 架构

思路：采用 union 联合体实现

提示：

```
union A {
    char a;
    int b;
};
```

参考代码：

vim union1.c

```
#include <stdio.h>
//声明联合体 union
typedef union A {
        char a;
        int b;
}A_t;
int main(void)
{
        A_t c; //分配 4 字节内存的联合体变量
        c.b = 0x12345678; //向内存写入数据 0x12345678
        if(c.a == 0x78)
            printf("当前处理器为小端模式.\n");
```

```
        else if(c.a == 0x12)
            printf("当前处理器为大端模式.\n");
        return 0;
    }
```

三十四、复合类型：枚举

1.枚举的本质就是一堆整数的集合,就是给整数取了个别名而已

类似#define 取别名(#define PI (3.14)),提高代码的可读性

2.枚举的特点：枚举值默认是从 0 开始,后面的成员依次加 1

关键字是 enum

3.声明枚举数据类型的语法：

enum 枚举数据类型名 {枚举值};

例如：

```
enum COLOR {RED, GREEN, BLUE};
```

结果是：RED=0,GREEN=1,BLUE=2

本质就是给 0,1,2 三个数字取别名分别叫 RED,GREEN,BLUE

将来程序可以直接使用 RED,GREEN,BLUE,类似使用 0,1,2

例如：printf("%d %d %d\n", RED, GREEN, BLUE);

或者

```
enum COLOR {RED, GREEN=250, BLUE};
```

结果是：RED=0,GREEN=250,BLUE=251

将来程序中描述红,绿,蓝三种颜色就不要用 0,1,2 了,用它们的别名：RED,GREEN,BLUE

4.枚举的经典代码演示(linux 操作系统核心代码片段)

vim enum2.c 添加

```
#include <stdio.h>
//定义检测函数
int check(int a)
{
    if(a != 0) {
            printf("表示成功了.\n");
            return 0; //表示成功
    } else {
            printf("表示失败了.\n");
            return 1; //表示失败
    }
}
int main(void)
{
    printf("%d\n", check(1));
    return 0;
}
```

保存退出

5.结论：程序员很难分辨 0 和 1 到底谁是成功了谁是失败了,代码可读性非常差

提高代码可读性的方法两种：

(1)利用#define 宏修饰 0 和 1,例如：

```
#define     RETURN_OK         (0)
#define     RETURN_FAILED  (1)
```

所以利用#define 宏优化之后的 check 函数：

```
//定义检测函数
    int  check(int  a)
    {
        if(a  !=  0)  {
                printf("表示成功了.\n");
                return  RETURN_OK;  //表示成功
        }  else  {
                printf("表示失败了.\n");
                return  RETURN_FAILED;  //表示失败
        }
    }
```

(2)利用枚举方法,也就是分别给 0 和 1 取别名：RETURN_OK,RETURN_FAILED

该进之后的代码：

vim enun3.c 添加

```
#include  <stdio.h>
//声明枚举类型
enum  RETURN  {RETURN_OK,  RETURN_FAILED};
//对声明的枚举类型取别名
typedef  enum  RETURN  return_t;
//定义检测函数
return_t  check(int  a)
{
    if(a  !=  0)  {
                printf("表示成功了.\n");
                return  RETURN_OK;  //表示成功
        }  else  {
                printf("表示失败了.\n");
                return  RETURN_FAILED;  //表示失败
    }
}
int  main(void)
{
    printf("%d\n",  check(1));
    printf("%d\n",  check(0));
    return  0;
}
```

三十五、函数指针(核心中的核心)

1.回顾：

指针函数：就是一个函数,只是函数的返回值是一个地址而已

例如：int *add(int a, int b)

2.明确：

函数名就是整个函数的首地址

例如：

```
int  add(int  a,  int  b)
{
      printf("....");
      printf("....");
      printf(".....");
}
```

结论：add 就是函数的首地址,等于 add 函数中第一条语句 printf 这条语句的首地址,看到 add 就是看到了一个地址(32 位,4 字节)。

3.函数指针概念：切记本质就是一种数据类型而已,类似 int,结构体,union,enum 等
这种数据类型由程序员自行定义。

4.函数指针数据类型声明的语法格式：不会分配内存,大型程序中写在头文件中,
返回值数据类型 (*函数指针名)(形参表);

例如：

```
int  (*pfunc)(int,  int);  //pfunc 就是一种数据类型,函数指针数据类型
```

或者

```
typedef  int  (*pfunc_t)(int,  int);  //给函数指针数据类型取别名为 pfunc_t
```

5.函数指针变量定义的语法格式：函数指针名 函数指针变量;

例如：

```
pfunc_t  pfunc;  //定义一个函数指针变量,将来保存函数的首地址
```

6.函数指针变量的初始化

```
pfunc_t pfunc    = add; //定义一个函数指针变量并且保存 add 函数的首地址也就是//pfunc
                        指向 add 函数
```

或者：

```
pfunc_t  pfunc;
pfunc  =  add;
```

7.通过函数指针变量来间接调用指向的函数的语法；

函数指针变量名(实参表)；

例如：

```
pfunc(100,  200);  //本质就是调用 add 函数,类似 add(100,  200)
```

8.注意：

函数指针的返回值和形参务必要跟指向的函数的返回值和形参保持一致！

回调函数：函数作为参数传递给其它函数,这个其它函数的形参一定是函数指针来保存这个传递的函数地址。

例如：add,sub 称之为回调函数,让别人调用自己的函数

```
1  #include<stdio.h>
2  typedef  int  (*pfunc_t)(int,int);
3  int  add(int  a,int  b){
4        return  a+b;
5  }
6  int  sub(int  a,int  b){
7        return  a-b;
```

```
 8 }
 9 int cal(int a,int b,pfunc_t pfunc){
10         int ret = pfunc(a,b);
11         return ret;
12 }
13 int main()
14 {
15         int ret = 0;
16         ret = cal(100,200,add);
17         printf("%d\n",ret);
18         printf("%d\n",sizeof(cal));
19         ret = cal(100,200,sub);
20         printf("%d\n",ret);
21
22         return 0;
23 }
```

9.函数指针经典用法

需求：目前有一堆函数,要求程序启动时,把这一堆函数挨个调用一遍

//笨办法

```
vim pfunction3.c
#include <stdio.h>
一堆函数的定义
void add(int a, int b)
{
    return a + b;
}
...
int main(void)
{
    //挨个调用
    add(1,2);
    sub(1,2);
    mul(2,2);
    div(3,2);
    ...
    return 0;
}
```

缺点：太 low 了!

//优秀代码：

```
vim pfunction4.c
#include <stdio.h>
//声明函数指针数据类型并且取别名
typedef int (*pfunc_t)(int, int);
//定义一堆函数
```

```
int add(int a, int b) {return a + b;}
int sub(int a, int b) {return a - b;}
int mul(int a, int b) {return a * b;}
int div(int a, int b) {return a / b;}
int main(void)
{
    //1.定义函数指针数组,每个元素是一个函数的地址
    pfunc_t pfunction_array[] = {add, sub, mul, div, NULL};
    //2.挨个调用
    for(pfunc_t *pfunc = pfunction_array; *pfunc; pfunc++) {
        int ret = (*pfunc)(200, 100); //取出每个函数调用
        printf("ret = %d\n", ret);
    }
    return 0;
}
```

三十六、多级指针(掌握二级指针)

1.回顾一级指针：就是之前所学的指针统称一级指针，指向一个普通变量的内存区域

例如：

```
int a = 250;
int *p = &a;
printf("a 的首地址%p, a 的值是%d\n", p, *p);
```

2.二级指针定义：指向一级指针的指针,也就是保存一级指针变量的地址

3.定义二级指针变量的语法格式：

数据类型 **二级指针变量名 = 一级指针变量的地址;

例如：

```
int a = 250;
int *p = &a; //p 是一级指针
int **pp = &p; //pp 是二级指针
printf("p 的地址是%p, a 的地址是%p, a 的值是%d\n", pp, *pp, **pp);
//修改
**pp = 300;
```

二级指针处理普通变量多余,采用一级指针即可拿下!

4.二级指针和字符串那点事儿：二级指针处理字符串

经典的笔试题：编写一个函数实现两个字符串的交换

例如：

```
char *pa = "hello";
char *pb = "world";
```

目标：pa->"world", pb->"hello",指针互换

参考代码：swap.c

```
1 /*字符串交换*/
2 #include <stdio.h>
3
4 void swap(char *px, char *py)
5 {
6       char *pz = px;
```

```
 7      px = py;
 8      py = pz;
 9 }
10
11 void swap1(char **px, char **py)
12 {
13      char *pz = *px; //将 hello 字符串的地址给 pz
14      *px = *py; //将 world 字符串地址给 px
15      *py = pz;  //将 hello 字符串地址给 py
16 }
17
18 int main(void)
19 {
20      char *pa = "hello";
21      char *pb = "world";
22
23      swap(pa, pb);
24      printf("%s, %s\n", pa, pb);
25
26      swap1(&pa, &pb);
27      printf("%s, %s\n", pa, pb);
28      return 0;
29 }
```

5.二级指针和字符指针数组的那点事儿:

(1)回顾:字符指针数组

```
char *p[] = {"hello", "world"};
```

或者

```
char *p1 = "hello";
char *p2 = "world";
char *p[] = {p1, p2};
```

p[0] = p1 = "hello"的首地址 = *(p+0)

p[1] = p2 = "world"的首地址 = *(p+1)

```
printf("%s %s\n", p[0], p[1]);
```

结论:显然 p 具有二级指针的意味,通过 p 获取的是一级指针的地址,也就是字符串的首地址。

变形:

```
char **pp = p; //pp 二级指针保存字符指针数组的首地址 p
printf("%s %s %s %s %s %s\n",
                    p[0], p[1], *(pp+0), *(pp+1), pp[0], pp[1]);
```

公式:char **pp 等价于 char *p[元素个数]

参考代码:ppstring.c

三十七、实际产品开发,主函数 main 函数将来必须这么写,否则遭鄙视!

1.main 函数的编写公式:

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

或者

int main(int argc, char **argv)

建议不要这么写：int main(void) 或者 void main(void)

2.切记：只要在命令行终端程序中(不管是什么命令终端程序)通过键盘输入的任何数据,计算机都把它当成字符串处理。

例如：

./helloworld 100 200

实际计算机存储这些输入的内容都是当成字符串：

"./helloworld"

"100"

"200"

内存存储的也是字符串

3.问：main 函数的 argc 和 argv 形参到底保存着什么值呢?

答：当运行程序时(例如./helloworld),程序员需要在键盘上输入信息,

例如：./helloworld,那么操作系统会自动将这些输入的信息传递给 main 函数,其中 argc 保存着命令行中输入的信息个数 argv 保存着输入的信息值

例如：

当键盘输入：./helloworld

此时：argc=1,argv[0] = "./helloworld"=保存着字符串的首地址

当键盘输入：./helloworld 100 200

此时：argc=3,argv[0]="./helloworld", argv[1]="100" argv[2]="200"

结论：

argc:保存给程序运行时传递的参数个数

argv:保存给程序运行时传递的参数信息值

目的：将来运行程序时,可以很方便给程序传递额外的数据信息，但是目前传递的信息计算机都把它当成字符串,程序员要求 100 就是一个整型数字,而不是字符串

问：怎么办?

答：利用大名鼎鼎的字符串转整型函数将传递的参数进行转换

unsigned long strtoul(char *str, char **end, int base)

函数功能：将字符串转整型数字

例如："100"->100

参数：

str:要传递转换的字符串的首地址

end：无需关注,一般给 NULL

base：指定转换时的进制:0,8,10,16

返回值：返回转换以后的整型数字

例如：

```
char *p = "100";
int a;
//将字符串"100"转成 10 进制整型数 100 保存给 a
a = strtoul(p, NULL, 0);
printf("a = %d\n", a); //a = 100
```

或者

```
a = strtoul(p, NULL, 10);//将 100 再强制转换 10 进制
```

```
    printf("a = %d\n", a); //a = 100
    或者
    a = strtoul(p, NULL, 8); //将 100 再强制转换成 8 进制
    printf("a = %o\n", a); //a = 100 的 8 进制
    或者
    a = strtoul(p, NULL, 16); //将 100 再强制转换成 16 进制
    printf("a = %#x\n", a); //a = 100 的 16 进制
  char *p = "0x100";
  //将字符串"100"转成 16 进制整型数保存给 a
    a = strtoul(p, NULL, 0);
    printf("a = %#x\n", a); //a = 0x100
  char *p = "0100";
  //将字符串"100"转成 8 进制整型数保存给 a
    a = strtoul(p, NULL, 0);
    printf("a = %o\n", a); //a = 0100
```

4.实际开发 main 函数的编码公式：main1.c

三十八、malloc 和 free 标准库函数(大神写好的,咱直接调用即可)

1.回顾：目前 C 程序分配内存的三种方式：

(1)定义变量：缺陷：不能一次性大量分配内存

(2)数组：缺点：虽然可以做到大量一次性连续分配内存,但是数据类型一致

(3)结构体：缺点：虽然可以做到数据类型不一致,但是要大量分配内存还是基于数组(同样数据类型也一致)

(4)不管是变量,数组还是结构体都涉及到局部和全局问题，如果是局部的使用范围相对比较小，如果是全局的使用范围大,但是不推荐用。

问：如何大量一次性分配内存并且做到数据类型无关关键没有使用范围的限定,也就是想什么时候分配内存就什么时候分配内存想什么时候释放内存就什么时候释放内存?

答：采用 malloc 和 free 标准库函数

2.详解 malloc 和 free 函数

(1)malloc 函数原型

```
void *malloc(unsigned long size);
```

函数功能：随时随地动态分配内存,并且分配的内存是连续的只要不调用 free 函数释放内存,它分配的内存一直存在并且分配的内存里面的数据是随机数。

形参：

size:传递要分配的内存大小,单位是字节,随意分配

返回值：返回分配的内存的首地址,跟类型无关，如果分配失败,返回 NULL。

(2)free 函数原型

```
void free(void *p)
```

函数功能：释放 malloc 分配的内存归还给操作系统

形参：

p：传递 malloc 分配的内存的首地址

(3)memset 函数原型

```
void memset(void *p, int data, int len)
```

函数功能：将内存中的数据设置为指定的数据

形参：

p:要设置的内存的首地址
data:要设置的内存新值
len:要设置的内存大小,单位为字节
例如：memset(p, 0, 512);

(3) 综合演练：malloc1.c,malloc2.c

```
1  #include<stdio.h>
2  #include<string.h>
3  #include<stdlib.h>
4  typedef struct student{
5        char name[20];
6        int age;
7        int id;
8  }stu_t;
9  stu_t *get_student_info(void){
10       stu_t *p = (stu_t *)malloc(sizeof(stu_t));
11       if(p == NULL){
12        return NULL;
13       }
14       memset(p,0,sizeof(stu_t));
15       strcpy(p->name,"guanyu");
16       p->age=18;
17       p->id=66;
18       return p;
19 }
20 int main(int argc,char **argv)
21 {
22       stu_t *p= get_student_info();
23       printf("%s %d %d\n",p->name,p->age,p->id);
24       free(p);
25       p = NULL;
26       return 0;
27 }
```

三十九、文件操作相关的标准库函数:fopen/fclose/fread/fwrite/fseek/rewind

1.fopen 函数原型

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

功能：打开文件

参数：

filename:指定要打开的文件名,建议用绝对路径

例如："/home/tarena/helloworld.txt"

mode:指定打开文件的方式,相关如下：

"r":以只读的方式打开,前提是该文件必须存在

"r+":以读和写方式打开,前提是文件必须存在

"w":以只写方式打开,如果文件存在,文件会被清空，如果文件不存在,创建该文件

"w+":以读或者写的方式打开,如果文件存在,文件会被清空，如果文件不存在,创建

该文件。

"a":以附加方式打开文件,属性只写,如果文件存在,要写入的新数据会追加文件的尾部,文件不存在,创建该文件。

"a+":以附加方式打开文件,属性可读可写,如果文件存在,要写入的新数据会追加文件的尾部,文件不存在,创建该文件。

返回值:返回文件指针,FILE 是自定义的数据类型,返回的指针将来代表文件本身如果文件打开失败,返回 NULL。

2.fclose 函数原型

void fclose(FILE *fp)

功能:关闭文件

参数 fp:传递文件打开以后的文件指针,就是 fopen 的返回值

3.fwrite 函数原型

unsigned long fwrite(const void *p, unsigned int size,
unsigned long count, FIlE *fp)

函数功能:向文件写入数据,就是将内存中数据保存到文件中(硬盘上)

参数:

p:传递保存数据的内存首地址,将来这些数据要写入到文件中

size:传递要写入的单块数据的大小,例如:单块数据为 4 字节

count:传递要写入的数据块的个数,例如:传递 4 个数据块

结论:总共写入了: 4*4=16 字节

fp:传递文件指针

返回值:如果写入失败返回-1,写入成功返回实际写入的数据块个数

4.fread 函数

unsigned long fread(void *p, unsigned int size,
unsigned long count, FIlE *fp)

函数功能:从文件(硬盘)读取数据保存到内存中

参数:

p:传递保存数据的内存首地址,这些数据来自文件

size:传递要读取的单块数据的大小,例如:单块数据为 4 字节

count:传递要读取的数据块的个数,例如:传递 4 个数据块

结论:总共要读: 4*4=16 字节

fp:传递文件指针

返回值:如果读取失败返回-1,读取成功返回实际读取的数据块个数

5.fseek 函数

int fseek(FILE *fp, long offset, int fromwhere)

功能:定位文件指针,改变文件指针的位置

成功返回 0,失败返回-1。

fp:文件指针

offset:从文件的哪个地方开始(字节数)

fromwhere:需要指定以下三个宏:

SEEK_SET:从文件的开头开始

SEEK_END:从文件的结尾开始

SEEK_CUR:从文件的当前位置开始

File.c

```c
/*文件操作的标准库函数*/
#include <stdio.h>
int main(void)
{
    //目标:向文件写入数组
    int a[] = {1,2,3,4,5,6,7,8};
    int len = sizeof(a) / sizeof(a[0]);
    int size = 0;

    //1.打开文件
    FILE *fp = NULL; //好习惯
    fp = fopen("/home/tarena/stdc/day13/a.bin", "w+");
    if(fp == NULL) {
        printf("文件打开失败.\n");
        return -1; //程序结束
    }
    //2.把数组中的数据(内存)写入到文件中(硬盘)
    size = fwrite(a, sizeof(int), len, fp);
    if(size == -1) {
        printf("文件写入失败.\n");
        fclose(fp);
        return -1;//程序结束
    } //结果:执行完毕,fp 文件指针现在指向文件的末尾
    printf("实际写入的数据块个数是%d\n", size);

    //3.将 fp 文件指针定位到文件的开头
    rewind(fp);

    //4.从文件中读取数据到内存
    int b[8] = {0}; //暂存读取到的数据
    size = fread(b, sizeof(int), 10, fp);
    if(size == -1) {
        printf("文件读取失败.\n");
        fclose(fp);
        return -1;
    }
    printf("实际读取了%d 个数据块.\n", size);
    //打印读取的数
    for(int i = 0; i < size; i++)
        printf("b[%d] = %d\n", i, b[i]);

    //5.从文件的开头往后数 8 个字节作为文件指针的开始
    //1 2 3 4 5 6 7 8
```

```
44      //      fp
45      int c[2] = {0};
46      fseek(fp, 8, SEEK_SET);
47      fread(c, sizeof(int), 2, fp); //从文件读取 8 字节数据放到数组 c 中
48      //1 2 3 4 5 6 7 8
49      //           fp
50      printf("%d %d\n", c[0], c[1]); //3 4
51
52      //从当前位置开始往后移动 8 个字节作为文件指针的开始
53      //1 2 3 4 5 6 7 8
54      //                 fp
55      fseek(fp, 8, SEEK_CUR);
56      fread(c, sizeof(int), 2, fp); //从文件读取 8 字节数据放到数组 c 中
57      printf("%d %d\n", c[0], c[1]); //7 8
58
59      //从结尾开始往前移动 12 个字节作为文件指针的开始
60      //1 2 3 4 5 6 7 8
61      //               fp
62      fseek(fp, -12, SEEK_END);
63      fread(c, sizeof(int), 2, fp); //从文件读取 8 字节数据放到数组 c 中
64      printf("%d %d\n", c[0], c[1]); // 6 7
65
66      //6.关闭文件
67      fclose(fp);
68      return 0;
69  }
```