这是一篇关于c语言的教程 大致描述c的常用语法和一些技巧 为了保持精简 我会提供一些搜索的关键词以保证可以搜索到其他没有讲解的问题

由于c语言标准有几个 包括但不限于c89 c99 c11 默认情况下 这里讲解c99标准 而c11与c99语法上大致相同 主要注意c89与c99的区别 具体的不同会被指出

另外 为了打字方便 这里统一使用空格分隔句子 不会出现任何标点符号

目录

快速入门

快速入门会提到大部分常用的语法 会略去一些细节 只给出一些最常用的语法 应注重程序的编写而不是语言的细节上

编译器 编辑器 集成开发环境

这是编写程序的必要准备 编写程序的过程中 我们需要编辑器 其中 你可以选择一款自己喜欢的编辑器 比如 Sublime 或者NotePad++ 以及其他的编辑器

编写好的程序 我们需要编译器将代码翻译成机器语言 这里 只使用 gcc

由于gcc是linux的编译器 linux系统一般自带gcc 而windows上运行gcc 可以下载TDM64

或者Mingw-w64 搜索之后下载即可

集成开发环境[IDE] 是既可以编辑也可以编译的软件 比如 Codeblocks ,Visual Studio ,Eclipse

由于是c语言的编写 推荐codeblocks

由于集成开发环境直接使用比较简单 这里不介绍如何使用 只讲如何用gcc编译编写的代码

linux下直接打开bash 输入

```
gcc test.c
```

这样会产生一个a.out的可执行文件 运行只需输入

`./a.out`

windows下的cmd并不好用 而cygwin过于庞大 可以使用较小的 git bash for windows 本身是给git使用的 不过编译c语言绰绰有余

打开git bash后 输入和linux下相同 这是生成的文件是a.exe 因为windows下 可执行文件的后缀必须是.exe 而linux并无限制 运行方式同linux下相同

`./a.exe`

如果想指定生成的可执行文件的名字 可以输入

`gcc test.c -o yourname.exe`

通过-o指定生成的名字

关于gcc的使用方式 可以参考相关博客 这里有一个常用的gcc编译选项整理

c程序的基本结构

一个最小的c程序 是这样的

```
int main()
{

}
```

main函数是一个程序的入口 int是main函数的返回值 该返回值是返回给操作系统的 一般返回0表示程序正常运行 如果不指定返回值 默认返回0 如上例程序就是用默认返回值

这个程序什么都干不了 我们让程序输出一段话 显示在终端上

```
#include<stdio.h>
int main()
{
    printf("Hello World\n");
}
```

`#include<stdio.h>` 表示包含一个头文件 其中 std是standard的缩写 表示标准 io表示输入输出

包含该头文件是因为使用了printf用于输出`Hello World\n`

其中" "括起来的部分被称为字符串 \n表示换行

注意 c语言的一条语句需要用`;`结尾

基本的数据表示

一个程序至少要能计算 而计算就需要数 c语言提供了一些数据类型用于表示数据

char 用于表示字符

int 用于表示整数 [注意是有范围的 现代的PC机上大致是-20亿到+20亿之间 具体的范围后序讲解]

double 用于表示浮点数[并不是小数 是一种对小数的近似的表示 并不精确 注意整数是精确的]

有了数据后 我们便可以进行一些数值计算了

```c
#include<stdio.h>
int main()
{
    int a=10,b=20;
    printf("a+b=%d\n",a+b);
}
```

`int a=10,b=20;` 表示定义两个int型的变量 名字为a和b 并分别初始化为10和20

最后输出a+b的值 `%d` 表示输出的是int类型

浮点数的使用 与int类型相似 只是输出的时候用`%f` 代替`%d`

注意 二者不能相互代替 是什么类型就要对应相应的符号

字符类型需要使用`%c` 表示

`char c='A';` 我们要输出c 则`printf("%c",c);` 这样终端上会显示一个A

' '用于将相应的字符转码 注意 ' '之间只能存放一个字符 多个字符需要使用字符串 后序讲解

程序的控制结构

理论和实践证明，无论多复杂的算法均可通过顺序、选择、循环3种基本控制结构构造出来。

c语言最基本的执行顺序就是从一个函数开始自上而下地顺序执行

循环语句可以控制一个语句或语句块执行若干次

而选择语句可以控制一个语句或语句块是否执行

在循环语句和选择语句中 必须判断某条件是否成立

c语言中条件成立使用**非0**表示 条件不成立使用**0**表示

循环

暴力求解是计算机最擅长做的事情 他的计算速度极快 例如 我们考虑一下高斯小学遇到的问题

1+2+3+...+100=?

利用for循环可以很容地编写这个式子

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

定义一个sum用于保存求和结果 for循环的形式是

`for(exp1;exp2;exp3)`

进入for循环首先执行exp1 之后将不再执行 然后执行exp2 此处一般放置条件判断 如果条件成立 则执行for循环下面的语句 执行完成后 会执行exp3 此处i++表示对i自增1 相当于`i=i+1` 之后再执行exp2以确定是否继续执行循环

注意`sum=sum+i`此处的`=`是赋值 注意与初始化区别开来 只有在定义的时候的`=`是初始化 初始化会与赋值有一些区别 具体细节后序描述

另外`=`表示的是赋值 而不是数学中的相等 相当于把`=`右侧的计算结果赋值给左侧

所以不能写成`sum+i=sum` 这样会编译错误

能写在`=`左侧的值叫做左值

另一个循环表达式是`while`

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

他比for的形式更为简单`while(exp)`

只要exp的求值结果不为0 就会一直执行下面的语句块

用`{}`括起来的叫做语句块 这里有两条语句[有几个分号就有几条语句] 所以必须使用`{}`

否则 只能像上例的for 后面只能接一条语句[即一个`;`]

下面看一个用`while`循环计算最大公约数的例子

由欧几里得辗转相除法公式

函数 gcd(a,b)可以计算a和b的最大公约数

其中 gcd(a,b)=gcd(b,a%b) 并且gcd(a,0)=a [其中`%`在c语言中表示求模 即取余数 如 10%3=1]

由以上公式和性质 我们可以把问题迭代至最后一个公式以化简问题的复杂度

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

int main(int argc,char *argv[])
{
    int a=15,b=10;
    while(b!=0)
    {
        int c=a%b;
        a=b;
```

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

在求完a%b之后 需要把原来的a设置成b 再把b 设置成a%b

其中!=代表不等于 如果不相等成立 则会返回一个非0值 表示循环可以继续 [!和=必须紧挨在一起 是一个整体]

如果要表示相等 则用==也是紧挨在一起的 [一个=表示赋值或初始化 它会修改位于它左侧变量的值]

条件判断

实际情况中 往往有一些在某一条件成立下 才执行的动作 比如求一个数的绝对值

如果这个数大于0 则不作任何动作 而小于0 需要求他的相反数

```c
#include<stdio.h>
int main()
{
    int a=-10;
    if(a<0)
        a=-a;
    printf("%d\n",a);
}
```

考虑一个更复杂的问题 一些学校往往会根据成绩划分等地 那么该怎么做呢

```c
#include<stdio.h>
int main()
{
    int score=89;
    if(score<60)
        printf("failed\n");
    else if(score<70)
        printf("pass\n");
    else if(score<80)
        printf("mean\n");
    else if(score<90)
        printf("nice\n");
    else
        printf("excellent\n");
}
```

通过if else if else 可以组合成一系列的判断 其中一个成立 那么就不再执行剩余的 注意 执行顺序是从上至下的 如果条件比较简单else if不是必须的

自定义函数

c语言允许自定义的函数 可以很方便的调用 例如 输出调用printf一样

一个函数的结构是这样的

```
返回类型  自定义函数名(参数列表)
{
    执行相关动作
}
```

看一个具体的例子 这个函数计算(x+y)*(x-y)

```
int fun(int x,int y)
{
    return (x+y)*(x-y);
}
```

函数定义之后 如何调用他呢

```
#include<stdio.h>

int fun(int x,int y)
{
    return (x+y)*(x-y);
}

int main()
{
    int a=20,b=10;
    int result=fun(a,b);
    printf("%d",result);
}
```

注意参数一一对应

有一个值得注意的地方是 **c语言的函数只有值传递 即换入函数的参数都是一份副本 对副本的改变不会影响到原来的变量**

另外 c语言的函数比数学的函数功能更多一点 他不仅可以返回一个值 还可以执行一些动作 但这些需要其他的一些东西

指针

c语言的指针是一种复合类型 这种类型的变量本身也保存一个值 他保存的值是一个内存的地址

内存的编址一般以8个bit为一个单位 称为一个字节

每一个字节都有一个唯一的地址 就像班级的每一个学生都有一个唯一的学号

指针的形式如下

```
int a=20,b=10;
int *ptr1=&a,*ptr2=&b;
```

`int *`表示一个指向int类型的指针 这里注意ptr2前面也有`*`这是因为指针是复合类型 所以必须再写一次 a之前的&表示获取a的首地址

`ptr1`与`ptr2`存储的是`a`和`b`的地址 那么如何获取到他们存储的值呢

```c
#include<stdio.h>

int main()
{
    int a=20,b=10;
    int *ptr1=&a,*ptr2=&b;
    printf("a=%d b=%d",*ptr1,*ptr2);
}
```

只需在指针变量名字之前加上`*`即可获取到他们的

是访问指针指向的内容之前 指针必须初始化为或被赋予合法的地址 合法的地址即程序可用的地址 包括变量的地址 申请的内存的地址 合法地址之外的地址的访问 往往会导致程序出错或崩溃

由于变量存储的位置是不变的 即地址不变 那么只需传递给函数他们的地址 就能在另一个函数中修改调用者的变量

例如 现在要交换a和b的值

```c
#include<stdio.h>

void swap(int *a,int *b)
{
    int tmp=*a;
    *a=*b;
    *b=tmp;
}

int main()
{
    int a=20,b=10;
    printf("%d %d\n",a,b);
    swap(&a,&b);
    printf("%d %d\n",a,b);
}
```

`void`表示函数不返回任何值

这里主函数也没有显式地返回任何值 但是主函数比较特殊 不标明返回何值 则默认返回0

这是其他函数所没有的

一定要注意的是 **指针必须指向类型一致的对象** `int*`的指针对象必须指向`int` 否则 不保证程序正确性

数组

有时候 需要使用大量的同一类型的变量 不可能一个一个地定义 这时候就可以使用数组

数组表示一组同一类型的变量 形式如下

`int arr[1024];`这样 就定义了一个长度为1024的`int`类型数组

数组的下标从`0`开始 所以不能使用`a[1024]` 这样就越界了 即访问了不合法的地址空间

假设现在有一组数据 需要输入 之后按递增顺序输出

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

```c

void swap(int *a,int *b)
{
    int tmp=*a;
    *a=*b;
    *b=tmp;
}

int find_min(int a[],int begin,int end)
{
    int min=begin;
    for(int i=begin+1;i<=end;i++)
    {
        if(a[min]>a[i])
            min=i;
    }
    return min;
}

void select_sort(int a[],int n)
{
    for(int i=0;i<n;i++)
    {
        int index=find_min(a,i,n-1);
        swap(&a[i],&a[index]);
    }
}

int main()
{
    int a[1024];
    int n;
    scanf("%d",&n);
    for(int i=0;i<n;i++)
        scanf("%d",&a[i]);
    select_sort(a,n);
    for(int i=0;i<n;i++)
        printf("%d ",a[i]);
}
```

这是一个简单的选择排序 每一次都找到未排序部分的最小值 然后把他放在第一个

`scanf` 表示输入 形式上与 `printf` 类似 稍微有一点区别 %d表示输入的是int类型的数

注意 由于传入函数的参数都是副本 所以只能传入地址来修改本地的值 所以 `scanf` 里的参数需要 `&`

c语言没有内置的字符串类型 所以只能通过字符数组表示

```c
#include<stdio.h>

int main()
{
    char str[1024]="this is a string\n";
    printf("%s",str);
}
```

字符数组的空间有限 如果字符个数超出字符数组长度再减去1 那么会发生错误

字符数组使用`'\0'`表示字符串结束 所以存储空间会占用掉一个

`"this is a string\n"`可以被称为字符串 不过他是只读的 不可修改 同样 他也是以`'\0'`结束

```
int main()
{
    char str[1024];
    str="this is a string\n";
    printf("%s",str);
}
```

str是一个字符数组 但是由于初始化和赋值的区别 上述代码是**错误**的

结构体

数组表示了一类相同类型的集合 实际情况中 往往还有一些不同类型的信息集合

例如 一个学生的信息 一般有 名字 学号 成绩

这类由不同数据类型结合起来的集合可以由结构体表示

```
#include<stdio.h>

struct student
{
    char name[20];
    int id;
    int score;
};

int main()
{
    struct student John={"John",17,85};
    struct student Lee={.id=10,.name="Lee",.score=89};
    printf("%s id:%d\n",John.name,John.id);
    printf("%s id:%d\n",Lee.name,Lee.id);
}
```

`struct`关键字表示结构体 用于定义一个结构体 注意 结尾有个`;`

在定义完类型后 可以使用`struct student`定义一个变量

上例中分别使用了两种初始化 第一种需要一一对应 第二种可以指定初始化

注意 只有初始化才能写成如上两种形式

赋值时不可使用 但可以这么写

`Jack=(struct student){"Jack",9,84};`

`=`右边的部分为一个`struct student`类型的临时变量 将他赋值给Jack

`struct`关键字必须出现在`student`之前 不可省略

但可以使用`typedef`来定义一个别名

`typedef struct student stu;`

这样 之后可以用`stu`来定义

例如`stu Pikachu`

在访问结构体内部的变量是 使用`.`来指定具体的内容

如果是指向结构体的指针访问 则有两种方式

```
#include<stdio.h>

struct student
{
    char name[20];
    int id;
    int score;
};

int main()
{
    struct student John={"John",17,85};
    struct student *ptr=&John;
    printf("%s id:%d\n",(*ptr).name,ptr->id);
}
```

通过指针解引用来访问 注意`.`运算符的优先级大于`*`故括号不能省略

为了简化这种写法 提供了`->`运算符

基本数据类型

c语言的基本数据类型分为整型和浮点型 整型表示整数 浮点数用于近似表示小数

整型又分为有符号和无符号 有符号整型的类型大致如下

1. `signed char`
2. `short`
3. `int`
4. `long`
5. `long long`

以上只有`char`前面必须加`signed`关键字才能表示有符号 其余不加默认为有符号

[虽然这很奇怪 但是标准并没有规定`char`是有符号还是无符号 不过大部分编译器都实现为有符号]

另外`short` `long` `long long`三个关键字后面都可以加`int`也可以省略不写 就像上面的表一样

例如`long long int`

下面给出无符号整型 无符号整型前面必须有`unsigned`关键字

1. `unsigned char`
2. `unsigned short`
3. `unsigned int`
4. `unsigned long`

5. `unsigned long long`

尽量不使用无符号类型运算 如果对无符号数的规则不是很清楚 使用无符号数容易出错

在大部分情况下 应使用有符号数 [java语言没有无符号数 这不妨碍他是最受欢迎的语言之一]

为什么c语言使用这么多类型来表示整数呢 一个原因是机器运算时 都是固定位数的二进制运算 这有利于计算速度的提升 而c语言本省 就是对机器的简单抽象 基本上 c语言能做的事 和机器基本相同

这么多类型 表示的数值范围是不一样的 关于c语言标准的描述 这里不进行描述 因为c标准并没有具体规定数值的范围 甚至没有规定应该使用补码 反码或者是其他编码 [这部分内容参考计算机组成原理或者数字电路]

由于大部分机器都使用补码 所以这里直接给出常用的**数据模型**

Data Type	ILP32	ILP64	LP64	LLP64
char	8	8	8	8
short	16	16	16	16
int	32	64	32	32
long	32	64	64	32
long long	64	64	64	64
pointer	32	64	64	64

表格中给出的是各类型占用的二进制位数 pointer是指针类型

一般情况下 windows系统与linux系统32位编译器使用ILP32数据模型

如果是64位的编译器 那么windows使用LLP64模型 linux系统使用LP64模型

[编译器位数指的是编译器生成的可执行文件的位数 不是编译程序本身的位数]

如果对二进制的数据表示不清楚 可查阅相关博客或资料

为了解决无法确定类型长度的问题 c99标准提供了一个stdint.h头文件 使用该头文件 即可使用固定位数的数据类型

例如

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

int main()
{
    int8_t x=100;
    int32_t y=-60;
    uint32_t z=65535;
    uint64_t w=1024;
}
```

`int`开头的为有符号数 `uint`开头的为无符号数 后面的数字只能取8 16 32 64

整型常量表示方法除了普通的写法之外 还有其他一些写法

```c
void f()
```

```
{
    int x=123; //十进制写法
    int y=012; //八进制写法 0开头便是八进制表示
    int z=0xf; //十六进制写法 0x开头

    long a=0L; //L后缀 代表long类型 不使用的话 0为int类型 之后会拓展成long
    long long int b=0LL; //LL后缀 代表long long int
    unsigned int c=0U;  //U后缀 代表无符号
    unsigned long long int d=1LLU; //LLU后缀 代表unsigned long long int
    short int e=1S; //S后缀 代表short

    float f=11.25f; //f后缀 代表float
    double g=12.5E12; //表示12.5乘10的12次方
}
```

上述的后缀均可换成小写或大写字母 `0x` 也可以用 `0X` 表示

小数的表示有以下几种类型

1. `float`
2. `double`
3. `long double`

其中 `float` 使用32位二进制表示 `double` 使用64位二进制表示 `long double` 由具体实现而定

一般来说 最常用的是 `double` 他的有效数字在15位左右[十进制下的有效数字]

`float` 的有效数字较少 大概是6位左右 一般情况下 除非有存储限制 应使用 `double`

`long double` 基本上不使用 这里只做介绍

一个值得注意的问题是 浮点数是不精确的 例如

```
int main()
{
    float x,y;
    x=123456789;
    y=123456788;
    printf("%f",x-y);
}
```

上例中 输出结果基本上不为正确的1 因为 `float` 的有效数字在6位左右 而上例中数字达到9位

如果使用 `double` 那么结果就没有问题

由于这个特性 计算顺序有时候也会影响最后的计算结果

运算过程中的数据类型与类型提升

不同数据类型之间运算时 有一定的规则

1. 不足 `int` 的 如 `char` `short` 等 提升为 `int`
2. 不小于 `int` 且位数相同的有符号与无符号计算 有符号提升为无符号
3. 短的整型与长的整型计算 短的提升为长的

4. 整型与浮点型计算 整型会提升为浮点型
5. `float`与`double`计算 `float`提升为`double`

注意有符号向无符号转的问题 这也是无符号经常带来的问题 例如

```
int main()
{
    int x=-1;
    unsigned int y=5;
    printf("%d",y>x);
}
```

正常情况下 y>x 是直觉上正确的 按c的规则 应返回一个非0值 然而 实际输出却是0 原因在于`int`会被提升为`unsigned int` 即-1的补码会被解释成无符号数 导致x被机器认为是一个很大的数

c语言还有一个基本类型 `void` 但他不能单独使用 必须作为复合类型的一部分或者是函数返回值

声明、定义与作用域

由于历史原因 c语言的一个名字[包括变量名 函数名 结构体名等]在使用前 必须在前面声明

一般情况下 定义了一个名字后 在后面即可使用它 即定义的时候会同时表示声明

定义表示的是从无到有地创建了一个名字 前面所有的例子中 都有大量的定义 例如 定义了一个int类型的变量 或者是定义一个函数 他与声明是不同的 声明并不创建新的名字

作用域是表示一个名字在哪里有效 出了作用域的范围 那么这个名字将不再有效

c语言分为**局部作用域**和**全局作用域**

全局作用域是指定义在所有函数之外 所有结构体之外的变量 [形式上说 是没有在`{}`内部的变量]

局部作用域是指定义在函数内部的变量 [形式上说是定义在`{}`内部的变量]

文字说明不够直观 这里直接给出例子

```
#include<stdio.h>

int x=10; //全局作用域

int main()
{
    printf("x=%d\n",x);
    int x=0; //局部变量隐藏全局变量
    printf("x=%d\n",x);
    {
        int x=20; //局部变量隐藏上一个局部变量
        printf("x=%d\n",x);
    }
}
```

`//`表示注释 只能注释一行

`/**/`表示多行注释 注释范围从第一个`/*`开始 直到遇到第一个`*/`结束 注意并不允许嵌套

注释是在预处理阶段进行的 并不发生在编译期

作用域也不仅仅是由`{}`决定的 例如

```
int fun(int x)
{
    int x=10; //wrong!
    return x+1;
}
```

上例会引发一个重名的编译错误 因为这两个x的作用域是相同的

另一个例子就是`for`

`for(int i=0;i<n;i++){}`中的`i`的作用域与`{}`内相同

定义变量时可以选择是否初始化 如果不进行初始化 局部变量和全局变量会有区别

全局变量会进行默认初始化[c语言中一般就是所有位为0] 局部变量则不进行任何操作

```
int x; //默认初始化 值为0

void fun()
{
    int x; //不初始化 值不确定
    int y=20; //指定初始化 值为20
}
```

以上结果似乎并没有显示出声明有什么用处 因为定义的时候即已经声明了 下面给出必须使用声明的例子

```
#include<stdio.h>

extern int add(int x,int y); //声明有add()这个函数

int main()
{
    int x=add(10,20);
    printf("%d",x);
}

int add(int x,int y)
{
    return x+y;
}
```

在main函数中 使用了add()函数 但此时add并没有定义 所有在此之前 必须说明有这个函数

`extern`关键字表示外部的 他可以引用其他文件中的全局变量

对函数的声明`extern`关键字是可以省略的 但对全局变量的声明 他是不可省略的 否则会被当成是定义变量 从而导致重名错误

[这里提一下 c语言有个历史问题 如果一个函数在使用前没有定义或声明 那么编译器会自动生成一个声明形式为`int function_name()`的声明 所以上例其实是可以编译通过的 但会有警告 不过 不是所有的函数的形式都是如此 如果对应不上 会引发一些程序正确性的错误 `()`内空表示可以接受任意个参数 这也是一个比较容易出错的问题]

变量和函数的声明与定义还可以添加一些修饰前缀

例如 需要使用一个不能被修改的量 使用`const`修饰 `const`原来是常量的意思 但在c语言中准确地说他是**只读[即 read-only]**

```c
#include<stdio.h>

int add(const int x,const int y)
{
    x=10; //wrong
    return x+y;
}

int main()
{
    const int x=100;
    int z=add(x,20);
    printf("%d",x);
}
```

`const`更多地是用在指针类型上 后序再介绍 不过定义一个全局的const变量还是很有用的

运算符与求值顺序

c语言提供了一些运算符 包括算数运算符 判断运算符 逻辑运算符 位运算符 赋值运算符等

首先说明算数运算符 有 + - * / % ++ -- 分别是加 减 乘 除 求模 自增 自减

其中++ --又分为前加和后加 这里直接给出例程

```c
int main()
{
    int x=1,y=2;
    int a=x+y; //a=3
    int b=x-y; //b=-1
    int c=x+x*y; //c=3
    int d=x/y; //d=0 注意 整数除以整数还是整数 向0取整
    int e=x%y; //e=1 在c99标准中 相当于取余数 只能对整型使用
    int xx=x++; //x自增1 并返回原值1 即xx=1
    int yy=++y; //y自增1 并返回增加后的值 即yy=3
}
```

其中乘法 除法 求模 优先级高于加法和减法

++ --由于会改变自身的值 优先级高于乘除和求模

判断运算符 用于处理逻辑条件

```
>  大于
<  小于
>= 大于等于
<= 小于等于
== 等于
!= 不等于
```

如果条件成立 那么他们会返回非0值[代表真] 否则 返回0值[代码假]

这些运算符**一次只能处理两个操作数** 不能像数学上的a>b>c

为了处理这类情况 可以使用逻辑运算符

```
&& 并
|| 或
!  条件取反
```

例如 a>b>c 应写成`a>b && b>c`

如果是a>b或者c>b 应写成`a>b || c>b`

在使用条件语句和循环语句时经常会用到这两个运算符

!用于逻辑取反 例如 ab的相反事件是 `!(ab)`括号不能去掉 因为!的优先级很高 [这里直接用!=`更好]

关于运算符的使用 这里不做过多的描述 自行搜索相关内容

这里主要描述的是更重要的**运算符优先级与结合性**

对优先级的理解 与数学中相似 如 *的优先级高于+

而结合性主要是相同优先级的运算符那个先处理的问题 类似与1+2-3是先算加法还是减法 并没有意义 因为结果相同

结合性主要是给一元运算符使用 例如++

这里先提一个指针的例子

```c
#include<stdio.h>

int main()
{
    int *p;
    int a[2]={0,1};
    p=&a[0];
    printf("%d",*p++);
}
```

指针可以进行算术加减[只能加减 不能乘除] 表示他指向位置的左右偏移

这里 指针p指向a[0] `*p++` 中 `*` 代表解引用 且他的优先级和++相同

这里就是一个相同优先级该先与哪个运算符结合的问题 单目运算符均从右至左结合 即先++ 再*

由于后置的++返回原值 即还是a[0]的地址 故解引用后访问到的值就是a[0]的值

求值顺序

诸如 `int x=(a+b)*(c+d);` 直观上讲 肯定是先求a+b 再求 c+d 然而c语言并没有规定求值顺序

编译器可以自由地选择先算哪一部分 因为上例先算哪个结果都相同 这也不破坏优先级和结合性

但是 如果是带有副作用的表达式 就会产生问题

例如 `int x=(a+b)*(++a);` 显然 先算左边和先算右边 求得的结果不同 但c语言没有规定到底先算哪边 所以 该表达式是一个未定义行为[undefined behaviour] 结果不确定

这类带有副作用的表达式 应尽量避免

另外 在一条语句中[即一个`;`] 函数调用顺序也是不确定的 例如`int x=f(1)+f(2);`

为了保证结果正确 函数f()返回的结果应与调用次序无关

不过 c语言还有几个运算符规定了求值顺序

`&&`和`||`运算符会从左往右依次计算 不过 `&&`左边的表达式值一旦是`0`则不再计算 `||`左边的表达式值一旦不为`0`也不再计算

`,`运算符也规定从左往右计算 不过 像函数的参数分隔符也是用`,` 但它不是逗号运算符 注意区分 函数参数的求值顺序也是不确定的

`?:`运算符会先求第一个表达式的值 再决定求第二个或者第三个 这两个只有一个会被求值

赋值运算符

赋值运算符`=` 虽然与初始化时的等号相同 但二者是不同的

另外 为了方便 c语言还提供`+=`之类的运算符 例如`x+=a+b;` 他与`x=x+(a+b)`相同

关于赋值问题 这里还有一个未定义行为 即一条语句中 不能对同一个变量同时赋值两次以上

例如`x=x++;`由于++会修改x的值 相当于赋值 这里同时赋值两次 结果是不确定的

另外一个经典的问题

```c
#include<stdio.h>

int main()
{
    int i=5;
    int x=(i++)+(i++)+(i++);
    printf("%d",x);
}
```

这里对i同时赋值了3次 其结果也是不确定的 较新的编译器 如gcc7.2 会对此类错误警告[但不报错]

最后附上c语言运算符的优先级与结合性表 越靠前的运算符优先级越高 相同行的运算符优先级相同

() [] -> .	从左至右
! ~ ++ -- + - * (type) sizeof	从右至左
* / %	从左至右
+ -	从左至右
<< >>	从左至右
< <= > >=	从左至右
== !=	从左至右
&	从左至右
^	从左至右
|	从左至右
&&	从左至右
||	从左至右
?:	从右至左
= += -= *= /= %= &= ^= |= <<= >>=	从右至左
,	从左至右

语句

c语言提供选择语句 循环语句 和跳转语句

选择语句包括

1. if语句
2. switch语句

循环语句包括

1. while语句
2. for语句
3. do while语句

跳转语句使用`goto`

选择语句例程

```
void fun(int a,int b)
{
    if(a==b)
    {
        a=1;
        b=2;
    }
    else
    {
        a=2;
        b=1;
    }
}
```

另外一个是switch语句

```
void f(int x)
{
    switch(x)
    {
    case 1:
        printf("first");
        break;
    case 2:
        printf("second");
        break;
    case 3:
        printf("third");
    default:
        printf("default");
    }
}
```

如果x匹配到某个值 那么就会跳转到那个位置去执行 `break` 表示结束switch语句 否则 不管有没有匹配都会继续向下执行 知道switch语句结束或者是遇到下一个 `break`

switch不能范围匹配 他只能匹配一个确定的值 并且必须是整型类型

循环语句例程

之前以有for语句和while的例程 这里不再给出

do while 语句与上述两个循环不同的是 他至少执行一次

```
void fun(int a,int b)
{
    do
    {
        printf("a-b=%d\n",a-b);
    }while(a>b);
}
```

由于该语句第一次循环时不进行条件检查 实际使用中是容易发生问题的 故不推荐使用do while语句

`break` 与 `continue` 关键字

在循环中 可能需要用到某一条件达成后立即结束循环 或者是某一条件成立时 直接进行下一次循环

```c
void fun(int a,int b)
{
    while(a<b)
    {
        if(a+b==10)
            break;
        if(a==10)
            continue;
        printf("process once\n");
    }
}
```

`break` 执行后会直接结束循环 [只能退出一重循环]

`continue` 执行后会立即回到循环的条件判断位置 然后开始下一轮循环

此处如果 `a+b==10` 成立 那么循环结束

如果 `a==10` 成立 那么下面的 `printf` 将不会执行 但循环继续

由于 `break` 和 `continue` 会破坏执行结构 所以应尽量避免使用 尤其是 `continue`

`goto` 语句

跳转语句会直接跳转到指定的位置执行[限制在一个函数内 不能在函数间跳转]

由于其任意性 一般是不会使用的 常用的地方一般是错误处理或者是多重循环退出

考虑以下问题 是否存在某个三位的数 abc 他等于a^3+b^3+c^3 [`^` 这里代表幂 c语言中他是**异或**]

只要找到一个就行

```c
int main()
{
    int result=-1;
    for(int i=1;i<10;i++)
    {
        for(int j=0;j<10;j++)
        {
            for(int k=0;k<10;k++)
            {
                if(i*i*i+j*j*j+k*k*k==i*100+j*10+k)
                {
                    result=i*100+j*10+k;
                    goto END;
                }
            }
        }
    }
END:
    printf("%d",result);
}
```

此处的三重循环 使用 `break` 是不能跳出的 使用 `goto` 则很方便

另一个作用是错误处理 方便统一管理

无论何种使用方式 `goto` 都是往后跳转 如果往前跳转 会导致程序逻辑混乱

一般情况下 避免使用 `goto`

指针与数组

指针和数组都是对内存资源的一种使用 二者相似又有区别 要理解指针与数组 就需要理解资源是如何存储在内存上的

典型的内存一般是每个字节[8个bit]都有一个唯一的地址例如

地址	**0**	**1**	**2**	**3**	**4**	**5**	**6**
内容	10	20	17	32	78	255	0

现有一个指针 `int8_t *p=3;` 则p的值为3 他存储的是一个地址 在解引用时 *p 可以获取到3号地址中的内容 即32

以上只是指针的一个简单例子 但运行在操作系统之上的程序 内存由操作系统管理 程序是无法直接获取真实的物理地址 而且资源由操作系统管理 使用内存必须向操作系统申请 否则访问的既是非法地址 操作系统会终止程序 并报告段错误

除了指针要指向合法地址外 还必须指明被指向的对象的类型 例如 上例中 被指向的对象的类型是 `int8_t`

void指针

c语言提供一个通用的指针 他可以指向任意类型 也可以被赋值给指向任意类型的指针 例如

```c
#include<stdio.h>

int main()
{
    int a=10;
    void *ptr_void=&a;
    int *ptr_int=ptr_void;
    printf("value is %d,address is %p",*ptr_int,ptr_int);
}
```

`void*` 只能指向对象 他不能解引用 因为他丢失了被指向对象的类型 必须重新使用另一个指针来赋予被指向对象的类型

`%p` 用于输出地址信息

在c++中 `void*` 代表的是空指针 而c语言中 他代表的是通用指针 这也是c和c++不兼容的一个地方

上述代码使用c++编译器编译 会引起一个类型不一致的编译错误

使用指针指向申请的内存

在函数上的变量 存储在一个叫做栈的地方 他在函数调用结束后会被释放 此时再访问他是非法的 并且 栈的容量是有限的 一般地 windows程序的栈空间是2MiB 而linux的栈空间一般是8MiB

[MB MiB的区别自行查阅]

为了长时间使用一块内存 需要申请内存 被申请的内存存储在堆上 他的生命周期不收函数调用的影响 直到程序员手动释放为止

下面看一个例子

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

int* fun()
{
    int a=100;
    int *p=malloc(sizeof(int));
    double *q=malloc(sizeof(*q));
    *p=200;
    *q=10.24;
    printf("%f\n",*q);
    free(q);
    return p;
}

int main()
{
    int *p=fun();
    printf("%d",*p);
    free(p);
}
```

`malloc`函数用于申请内存 并返回该内存块的首地址 他接受一个内存块大小的参数[即指明需要多少字节内存]

`sizeof`运算符 [虽然他很像一个函数 但是他是运算符] 用于计算类型的占用内存的字节数 或者计算指定对象占用的内存字节数[二者其实是一样的 `sizeof`通过推导被指定对象的类型来计算字节数 上例中`*q`的类型是`double`]

被申请的内存块不会自动释放 需要使用`free`函数释放 注意 `free`函数的参数必须是申请内存的首地址 而且 被释放后就不能再次释放 否则会引发错误

`malloc`和`free`函数需要包含头文件`stdlib.h`才可使用

一般情况下 申请内存使用最多的地方会涉及结构体和数组

指向`const`类型的指针

指针可以指向`const`类型 例如

```c
int fun()
{
    int x=1024;
    const int *p=&x;
    *p=10; //wrong!
    printf("%d",*p);
}
```

如果`p`指向`const int`那么一点问题都没有 因为类型描述是一致的

但是 用`const`修饰被指向对象的类型的指针 可以指向不被其修饰的对象

因为x是可以修改的 用一个不能被指针修改的类型没有什么问题 这样可以防止误修改

上述的`*p=10`是错误的 因为`*p`的类型是`const int`他是不可修改的

这在函数参数传递过程中更为有效 例如

```c
#include<stdio.h>

struct stu
{
    char name[20];
    int  id;
};

void print_stu(const struct stu *s)
{
    printf("%s , id: %d\n",s->name,s->id);
}

int main()
{
    struct stu Jack={"Jack",21};
    print_stu(&Jack);
}
```

对于打印函数 是绝对不会修改对象信息的`const`关键字可以防止这种事情的发生

这类指针也被称为**常量指针** 他是个指针 被指向的对象是常量[相当于控制指针访问权限]

不可修改的指针

指针本身也是一个对象 他存储在内存上 同样 他也可以像基本类型一样 添加`const`以防止其被修改[指针本身的值不被修改 而指向的对象是否可以修改 需要上述的`const`]

```c
int main()
{
    int x=10,y=20;
    int *const p=&x;
    p=&y; //wrong!
    printf("%d",*p);
}
```

上述的`p`本身就是一个常量 他需要初始化 一旦初始化后就不能被修改 但他可以修改被指向对象 因为被指向对象的类型是`int` 可以被修改

这个指针通常被称为**指针常量** 他是一个常量 并且是个指针

当然 也可以构造一个常量指针常量 `const int *const p=&x;`

另外 `*`与`const`不必紧挨在一起 中间也可以用空格分离 如 `int * const p=&x;`

数组

数组用于表示含有n个元素的集合 这n个元素的类型是相同的 例如

`int a[10];` 代表10个`int` 类型的元素

`struct stu a[100];`代表100个`struct stu`类型的元素

c语言的数组下标是从0开始的 所以合法的访问范围是0到元素个数-1

越界访问数组是一种未定义行为 往往会引发程序错误

定义数组长度的量必须是常量 但c99中允许**局部数组**的长度为变量

```
int size=1024;
const int size_else=1024;
int arr[size]; //错误 全局数组长度必须是常量
int tab[size_else]; //错误 c语言中const代表只读 故也不是常量
int list[1024]; //正确 1024是常量

int f(int n)
{
    int a[10];
    int b[n]; //只有支持c99的编译器可以 并且c++不允许
    a[2]=20;
    a[10]=9; //错误 访问越界
    a[0]=a[2];
}
```

数组的初始化

局部数组不会默认初始化 全局数组会默认初始化 例如

```
int arr[1024]; //默认初始化 全为0
int tab[]={0,1,2,3,4,5}; //长度自动推导 为6 并且6个元素的值依次如表中所示

int f(int n)
{
    int a[10]; //不初始化 每个元素的值不确定
    int b[20]={1,2,3}; //前三个元素依次为1 2 3 剩余的执行默认初始化 为0
}
```

局部数组一旦初始化 没有指定初始化的值会被默认初始化 c语言中 相当于所有位置0

c99的数组可以指定初始化

```c
void f()
{
    int a[10]={[5]=2,[4]=3};
    printf("%d",a[5]);
}
```

上例中 对数组a初始化`a[4]`和`a[5]` 剩余没有指定的元素执行默认初始化

遗憾的是 标准没有规定范围初始化 如果想让整个数组的元素都初始化为1 那么只能用循环编写

字符数组与字符串

c语言中使用字符数组代表字符串

```c
void f()
{
    char s[]="this is a string.\n";
    char t[]={'o','t','h','e','r','\0'};
    printf("%s",s);
    printf("%s",t);
}
```

字符串有两种初始化方式 常用的是第一种 第二种其实就是数组的初始化方式 注意 字符串的末尾有`'\0'`用于代表字符串结束 第一种初始化方式会自动添加 第二种需要手动添加

如果忽略了`\0` 在输出时会发生访问越界的错误 因为没有这个标记 就不知道字符串何时结束

字符数组`[]`中的长度也可以指定 但必须大于初始化长度 [注意`\0`也占一个位置 要算进去]

获取数组长度

数组也是一种类型 同样可以使用`sizeof`运算符获取数组的字节数

```c
void f()
{
    int a[1024];
    int size=sizeof(a); //数组a的字节数
    printf("%d",size);
}
```

下面给一个通过`sizeof` 求数组元素个数的例子

```c
void f()
{
    int a[]={1,2,3,4,5,6,7,8,9};
    for(int i=0;i<sizeof a/sizeof a[0];i++)
        printf("%d ",a[i]);
}
```

此例中很好地体现了`sizeof`并非是函数 而是运算符 他的优先级高于`/`

不过为了清楚起见 一般还是写作`sizeof(a)/sizeof(a[0])`

对于类型而言 括号是必须的 必须写作`sizeof(int)`

通过求得数组的字节数和数组每一个元素的字节数 通过除法 就可以计算出总用有多少个元素

上例中 数组的长度是自动推导的 故可以只添加数组元素而不做其他任何改动

对字符串使用`sizeof`

```
void f()
{
    int size=sizeof("abcd");
    printf("%d",size);
}
```

字符串`"abcd"`的类型是`const char[5]`所以求得的字节数便是5

注意最后末尾的`\0`

指针与数组的联系与区别

使用上的联系

指针和数组在使用的形式上似乎是共通的 例如

```
void f()
{
    int a[]={0,1,2,3,4,5,6,7};
    int *p=a;
    printf("%d %d %d %d %d",a[1],*p,*(p+5),p[4],*(a+6));
}
```

上述的几种访问方式 无论是指针还是数组 都可以混用

指针+数字 表示相对于指针指向的首地址的偏移

数组的`[]`运算符实际上是对这种偏移写法的简化

参数传递时的联系

数组传递给函数时 数组类型会退化成指针[只退化一级 多维数组再讲解]

例如`int a[10];`传递给函数`void fun(int arr[],int n);`时

调用函数`fun(a,10);`此时 arr的类型实际上是`int*` 上例中的写法实际上是对`void fun(int *arr,int n)`的一种简化 二者是等价的

同样的 可以传递给一个申请的内存块给该函数

`int *ptr=malloc(10*sizeof(int));` `ptr`指向的内存块也可以包含10个`int`类型的数据

调用时 同样的`fun(ptr,10);`

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

void fun(int a[],int n)
{
    for(int i=0;i<n;i++)
```

```
        a[i]=i;
}

void print_arr(int a[],int n)
{
    for(int i=0;i<n;i++)
        printf("%d ",a[i]);
}

int main()
{
    int a[]={0,1,2,3,4,5,6,7,8,9};
    int *ptr=malloc(10*sizeof(*ptr));
    print_arr(a,10);
    printf("\n");
    fun(ptr,10);
    print_arr(ptr,10);
}
```

上例中 `fun()` 函数用于初始化数组 `print_arr` 用于打印数组中的元素

数组与指针在类型上的区别

虽然数组和指针在使用上有相似之处 尤其是处理内存块时 二者几乎等价

但是 二者在类型上是不同的

数组类型具有几个特征

1. 数组名是常量 不可改变
2. 数组名本身不占据内存空间 他不需要在别处额外存储
3. 数组类型包含数据块的长度信息

对比指针 指针有这几个特征

1. 指针名不是常量 可以修改 [const指针仅仅是只读]
2. 指针本身需要在别处额外存储 占据一定的内存空间
3. 指针只包含指向内存块的类型信息 他不知道指向的是内存块 还是一个对象的内存

例如 在地址 `0x00001000` 处开始 有一个 `int a[16];` 的数组 `a` 是不占据内存空间的 真正占据空间的是数组的16个元素

而指针则是 在地址 `0x00002000` 处存储一个指针 里面的内容是 `0x00001000` 即 该指针指向了数组 `a`

指针本身占用了内存 占用的内存用于存储地址信息 表示该指针到底指向了哪块内存 具体存数据的地方则是被指向的那块内存

由此 既然二者差别如此之大 为何数组和指针在使用形式上如此相似

其中的原因便是 c语言**规定**数组名本身的值要和数组的首地址相同 即 `&a` 的值与 `a` 的值是相同的

```
int main()
{
    int a[]={1,2,3,4,5};
    printf("a=%p &a=%p &a[0]=%p",a,&a,&a[0]);
}
```

并且 和数组的第一个元素的地址也是相同的

而指针则不具备这一特征 可以发现

```
int main()
{
    int a[]={1,2,3,4,5};
    int *ptr=a;
    printf("ptr=%p &ptr=%p &ptr[0]=%p",ptr,&ptr,&ptr[0]);
}
```

`ptr`的值与`&ptr`的值是不同的 原因在于指针本身需要存储

当然 言归正传 数值相同不代表类型相同 `a`的类型是`int[5]` 而`&a`的类型是`int(*)[5]`即指向数组的指针类型

该形式比较奇怪 在详细讲解c语言的类型之后 将会理解为什么是这种形式

多维数组

对于矩阵来说 一维的数组在使用上是比较麻烦的 c语言支持多维数组

二维数组的定义`int a[3][3];`三维数组的定义`int a[3][3][3];`更高维的依次类推

在使用时 注意下标也和一维的一样从0开始 例如 访问第二行第一列 `a[2][1]` 注意不要用`,`那是错误的

多维数组在存储上其实是一维数组 这个格式与使用无关 你可以将第一个[]当做行 也可以将第二个[]当做行

虽然使用上无关 但要编写高效率的代码 需要了解这种格式 因为与**局部性原理**有关

多维数组在传参时需要注意一点 数组退化成指针只退化一级 故`void f(int a[][],int m,int n)`这类传参方式是错误的 多维数组必须指明维数大小 除了最高的那一维 例如 这里应写成`void f(int a[][3],int n);`

但如果各维数不确定怎么办 似乎也没有什么好的办法 一种解决办法是 将多维数组转化成一维数组

```
#include<stdio.h>

void print_matrix(int *a,int m,int n)
{
    for(int i=0;i<m;i++)
    {
        for(int j=0;j<n;j++)
        {
            printf("%d ",a[i*m+j]);
        }
        printf("\n");
    }
}

int main()
{
    int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
    print_matrix(&a[0][0],3,3);
```

```
}
```

`a[i*m+j]` 涉及多维数组存储格式 实际上 例如 `int v[3][5];` 就是把内存块分成3份 这每三份又分成5个元素存储空间

以2*3矩阵为例

一维顺序	0	1	2	3	4	5
访问	`a[0][0]`	`a[0][1]`	`a[0][2]`	`a[1][0]`	`a[1][1]`	`a[1][2]`

根据这种存储结构 可以将二维数组当成一维数组访问

但是 传递参数时 **不能传入数组名 因为会引发一个类型不一致的错误**

另一种看法 其实多维数组本身不是多维数组 更应该将其看做数组的数组

例如 `int a[2][3];` 可以看到 `a` 存储了2个长度为3的一维数组 即 `a[0]` `a[1]` 都是数组 并且长度为3 而 `a` 是存储这两个数组的数组

多维数组的初始化

```
int main()
{
    int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
    int b[3][3]={1,2,3,4,5,6,7,8,9};
    int c[3][5]={[1]={1,2},{[2]=9},};
}
```

多维数组初始化建议使用第一种 第二种与存储格式顺序相同 第三种使用了指定初始化

在初始化元素不足时 最后的 `,` 是可以添加的 这在一维数组也是如此 用于方便后序添加元素

初始化一部分后 剩余的执行默认初始化

申请多维数组

使用 `malloc()` 申请一个多维数组 例如 申请一个行为10 列为20的多维数组

```
int main()
{
    int **p;
    p=malloc(10*sizeof(*p)); //p=malloc(10*sizeof(int*));
    for(int i=0;i<10;i++)
        p[i]=malloc(20*sizeof(*p[i])); // *(p+i)=malloc(20*sizeof(int));
}
```

使用二维指针申请一个多维数组 与二维数组的区别是 这需要额外的10个空间用于存储指针 注意的是 这种分配方式的空间不一定是连续的 他的使用方式直接 `p[x][y]` 即可 传递参数时使用 `int **p`

他实际上是先申请了10个指针 这些指针用于指向10个长度为20的数组（或者叫内存块）

c99的多维数组参数传递

在c99中 可以使用一种方式传递不同行列数的多维数组

```
void print_matrix(int m,int n,int a[m][n])
{
    for(int i=0;i<m;i++)
    {
        for(int j=0;j<n;j++)
        {
            printf("%d ",a[i][j]);
        }
        printf("\n");
    }
}
```

其中 `int a[m][n];`中的`m`是可以省略的 另外 `m` `n`的定义必须放在前面 以使`int a[m][n]`时这两个量已经定义了 否则 会发生变量未定义先使用的错误

函数

将代码全部集中于`main`函数不是一个好的主意 单个模块的代码越长 意味着出错的可能性加大 更重要的是 这不利于代码的维护和除错 使代码更易理解 这是非常重要的

c语言通过函数 可以实现对模块的拆分 一般来说 一个函数实现一个功能 也可以多个函数实现一个功能

不过 一个函数实现多个功能也是可以的 但这并不是个好主意

函数的构成

一个函数 需要有

1. 函数名
2. 返回类型
3. 参数列表
4. 函数体

一个简单的例子就是交换两个变量

```
void swap(int *a,int *b)
{
    int tmp=*a;
    *a=*b;
    *b=tmp;
}
```

`swap`是函数名 `void`是返回类型 `(int *a,int *b)`是参数列表 `{...}`是函数体

这几个部分缺一不可 参数列表可以为空 写作`void fun(void){}` 这虽然也很奇怪 但由于历史原因 `void fun(){}`的写法虽然更像是参数列表为空 但实际上他表示可以接受任意个参数[但是无法使用到传进去的参数如何获取？]

函数也可以有返回值 例如 编写一个二分查找的函数 找到则返回下标 未找到则返回-1

```
```

```c
#include<stdio.h>

int bfind(int a[],int n,int key)
{
    int begin=0,end=n-1;
    while(begin<=end)
    {
        int center=(begin+end)/2;
        if(key<a[center])
            end=center-1;
        else if(key>a[center])
            begin=center+1;
        else
            return center;
    }
    return -1;
}

int main()
{
    int a[]={1,3,5,7,9};
    int index=bfind(a,sizeof(a)/sizeof(a[0]),5);
    printf("%d",index);
}
```

`return`关键字用于返回函数的返回值 如果返回值为空 则写作`return;`即可

返回值的类型要相匹配 类型不一致且不存在隐式类型转换 则是错误的

函数一旦执行`return`函数也便结束了

函数声明

函数在使用前必须声明 定义函数的同时也会声明 声明后的函数可以在下方使用 其中 一个好的声明方式就是直接使用函数定义去除掉函数体的那一部分

```c
#include<stdio.h>

int bfind(int a[], int n, int key); //声明
int bfind(int[],int,int); //可以重复声明 并且 声明时可以去除参数列表中的名字 但定义时不能去

int main()
{
    int a[]={1,3,5,7,9};
    int index=bfind(a,sizeof(a)/sizeof(a[0]),5); //在上面已经声明
    printf("%d",index);
}
int bfind(int a[],int n,int key) //定义函数 在他下方的函数可以直接使用 但上方的函数不可见 需要声明
{
    int begin=0,end=n-1;
    while(begin<=end)
    {
        int center=(begin+end)/2;
        if(key<a[center])
```

```
            end=center-1;
        else if(key>a[center])
            begin=center+1;
        else
            return center;
    }
    return -1;
}
```

参数调用顺序

函数参数的调用顺序是不确定的 例如`f(exp1,exp2,exp3);`先求解哪个表达式是不确定的 所以需要保证表达式求值结果与调用顺序无关 大部分编译器是从右往左求值 但不保证一定会是这个顺序 下面看一个例子

```
int f()
{
    static int x=0;
    return x++;
}
int g()
{
    int x=0;
    return x++;
}
void test(int a,int b,int c)
{
    //do something
    ...
}
void start()
{
    test(f(),f(),g())
}
```

上述例子就是一个未定义行为 因为函数`f()`的返回值与调用顺序有关 会引发二义性问题

递归函数

函数调用自己即称为递归 递归非常有用 他常常可以简化程序的编写

递归函数有两个要求

1. 必须有个终止条件 即递归出口
2. 每次调用后 问题都会缩小以趋近于终止条件

递归与数学上递归函数类似 例如 求解最大公约数问题

已知gcd(a,b)=gcd(b,a%b) ,gcd(a,0)=a

首先 第一个等式每次都可以将问题缩小 第二个等式表明递归的终止条件 因为已经可以得到结果了

所以可以很容易地编写这个函数

```
int gcd(int a,int b)
{
    if(b==0)
        return a;
    return gcd(b,a%b);
}
```

不用去考虑递归函数到底是如何运行的 只需要知道 每次递归调用 最终将问题归结于递归出口

如果问题不能简化到递归出口 就会陷入无限递归 这会引发栈溢出的错误 程序将崩溃

递归问题的理解 其实就是将一个大问题化简成若干个小问题 小问题小到一定程度 就可以直接给出答案 而每一个小问题和大问题的解法又是一样的 考虑一个求幂算法 用于求解x^y 注意二者都是正整数

简单的求解x的y次幂 即做y次乘法 实际上 这里存在重复计算 例如 2^6=64 可以转化成(2^3)*(2^3) 这样只需4次乘法 对于计算速度上来说 提升是显著的 显然 2^6和2^3的求解是同一个解法可以求出的

下面给出求幂的递归版本 只需要注意如何处理奇数次幂

```
int qpow(int x,int y)
{
    if(y==0)
        return 1;
    else if(y==1)
        return x;
    else
        return qpow(x,y/2)*qpow(x,y-y/2);
}
```

`y/2` 如果除不尽 整型类型会向零取整 这里的递归出口有两个 可以证明 最终都会以这两个为终止条件

这是也是一个分治法 每次将一个大问题分解为两个小问题 最终再将两个小问题合并 小问题也是如此处理 直到小问题足够小

递归函数是否总能高效 答案是否定的 实际上递归要比迭代的版本慢一些 但是编写极其容易 不容易出错

下面考虑一个递归版本会十分慢的情况

对于斐波那契数列的定义为 f(n)=f(n-1)+f(n-2) ,f(0)=0 ,f(1)=1

这个数列的定义很明显 就是这个数是前两个数之和 他的递归函数写法为

```
int fib(int n)
{
    if(n==0)
        return 0;
    else if(n==1)
        return 1;
    else
        return fib(n-1)+fib(n-2);
}
```

很明显 这里存在大量的重复计算 例如 fib(10)=fib(9)+fib(8) 而fib(9)又会计算一次fib(8) 实际上这已经计算过了 这会引起大量的重复计算 大大浪费计算资源

上一个求幂的函数也是如此 考虑qpow(2,10)会分解为qpow(2,5)*qpow(2,5) 这就需要计算两次2^5

通过改写 可以得到快速幂的版本

```
int qpow(int x,int y)
{
    if(y==0)
        return 1;
    else if(y==1)
        return x;
    else if(y%2==0)
    {
        int r=qpow(x,y/2);
        return r*r;
    }
    else
    {
        int r=qpow(x,y/2);
        return r*r*x;
    }
}
```

这里的版本比上一个高效不少 因为去除了重复计算的地方 即它不会两次计算同一个参数

这是分治法中递归的一种应用 利用递归可以很容易地编写分治算法

上一个二分查找的例子也是分治法 即从一个有序的数列中找到一个数 先看中间的元素 如果小于他 那么他就在左边 如果大于他 那么就在右边 每次都可以将问题缩小一半

```
int bfind(int a[],int begin,int end,int key)
{
    if(begin<=end)
    {
        int center=(begin+end)/2;
        if(key<a[center])
            return bfind(a,begin,center-1,key);
        else if(key>a[center])
            return bfind(a,center+1,end,key);
        else
            return center;
    }
    return -1;
}
```

编写递归函数 刚开始可能会有些困难 实际上 递归的使用是很方便的 有时候往往递归比迭代更好编写

递归和迭代可以相互转化 能用递归编写的算法 一定可以用迭代编写 反之亦然

编写递归函数的关键 在于能否化为子问题求解 并且子问题同原问题的处理相同 同时 化简的子问题一定会向递归出口靠拢

函数也可以相互递归 考虑一个不太好的例子 用于判断一个数的奇偶性

这里不使用求模运算 而是使用一个较慢的版本

1. 定义0是偶数
2. 如果比他小1的数是偶数 那么他是奇数
3. 如果比他小1的数是奇数 那么他是偶数

```
int is_odd(int x);
int is_even(int x)
{
    if(x==0)
        return 1;
    if(is_odd(x-1))
        return 1;
    else
        return 0;
}

int is_odd(int x)
{
    if(x==0)
        return 0;
    if(is_even(x-1))
        return 1;
    else
        return 0;
}
```

尾递归及其优化

当一个函数调用自身只出现在函数末尾 且仅仅调用它自身[即仅仅出现 `return fun();` 而不是 `return fun()+1;`] 那么称之为尾递归 以上递归的例子中 只有第一个求解最大公约数是尾递归 其他的例子均不是

从尾递归的形式上看 可以添加一条 `goto` 语句来优化 以最大公约数为例

```
int gcd(int a,int b)
{
BEGIN:
    if(b==0)
        return a;
    int r=a%b;
    a=b;
    b=r;
    goto BEGIN;
}
```

可以看到 当参数不满足递归出口条件时 改写参数后重新回到函数开始处继续执行即可

编译器会自动优化尾递归 上例只是补充编译器是如何优化尾递归的

函数指针

指针可以指向一个函数 指向函数的指针被称为函数指针

```
int fun(int x)
```

```
{
    return x*x;
}

int main()
{
    int (*p1)(int)=fun;
    int x=(*p1)(2);
    int y=p1(2);
    int (*p2)(int)=&fun;
    typedef int (*FUN_TYPE)(int);
    FUN_TYPE p3=fun;
}
```

指针指向函数后 他的调用方式可以和原来函数一样 不需要解引用 而对函数取不取地址也相同 二者都返回函数的首地址

需要注意的是 这里的类型声明方式 函数指针的类型 在原来函数名的地方 改成`(*)` 而指针名写在`*`之后

如此奇怪的定义方式 与c语言的语法分析器算法有关 利用`typedef`定义了一个类型别名 新类型名的位置也和前者相同 使用新类型名即和基本数据类型相同

那么函数指针有什么用处呢 考虑一个标准库函数的例子

对于数据排序 可能会对整型变量按非递减排序 也可以是对字符串按字典序排序 如何实现一个通用的排序函数呢

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

void print_arr(int a[],int n)
{
    for(int i=0;i<n;i++)
        printf("%d ",a[i]);
    printf("\n");
}

int cmp1(const void *a,const void *b)
{
    return *(int*)a-*(int*)b;
}

int cmp2(const void *a,const void *b)
{
    const int *x=a;
    const int *y=b;
    return *y-*x;
}

int main()
{
    int a[9]={9,4,8,7,1,3,2,6,5};
    int b[9]={3,2,6,7,1,9,4,8,5};
    qsort(a,9,sizeof(a[0]),cmp1);
    qsort(b,9,sizeof(b[0]),cmp2);
    print_arr(a,9);
    print_arr(b,9);
```

```
}
```

`qsort` 接受四个参数 第一个参数为数组首地址 第二个参数为数组元素个数 第三个参数为单个元素的字节长 第四个为比较函数的函数指针

其中 比较函数返回0表示相等 小于0表示小于 大于0表示大于 上述两个例子直接使用减法来表示这种结构

比较函数传入两个`const void*`指针 使用时需要类型转换 第一个使用了强制类型转换 而第二个使用了`void*`作为通用指针可以赋给任意其他类型的指针 但是`const`是不可丢弃的

这样就对两个数组分别实现从小到大排序和从大到小排序

c标准库有个`signal.h`头文件 它用于接受某个信号后执行相关动作 一般来说 各个信号都有默认执行的动作 例如 除零错误的信号 程序外部强制终止的信号等 可以通过`signal`函数来改写执行的动作

例如 [该例程需要在linux系统上运行]

```c
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>

void handler(int signum)
{
    fprintf(stderr,"catch signal %d\n", signum);
    exit(0);
}

int main()
{
    signal(SIGFPE,handler);
    int x=0;
    int y=10/x;
    printf("%d",y);
}
```

在遇到除零错误时 会给进程发送一个除零错误信号 此时会调用`handler`

`exit`函数用于直接退出进程 其中的参数表示返回给系统的值 `fprintf`用于向错误流输出信息 他保证输出结果一定会显示在屏幕上

内联函数

`inline`关键字用于修饰函数 以表示该函数为内联函数 内联函数可以消除调用函数带来的开销 而使用时与函数相同 一般来说内联函数很短 以及足够得简单 内联函数并非总能内联

例如 `swap`函数 该函数可以实现内联

```c
inline void swap(int *a,int *b)
{
    int tmp=*a;
    *a=*b;
    *b=tmp;
}
```

内联函数会在调用时被展开 实际上编译后它已经不是一个函数了 内联函数一般写在头文件中

可变参数列表

诸如`printf`与`scanf`函数 它们可以接受任意个参数 实际上是使用了可变参数列表 它使用`...`表示

例如`printf`的函数原型是`int printf(const char *format,...);`

下面给出一个用于求n个数和的例子

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

int sum(int n,...)
{
    va_list ap;
    va_start(ap,n);
    int r=0;
    for(int i=0;i<n;i++)
    {
        int num=va_arg(ap,int);
        r+=num;
    }
    va_end(ap);
    return r;
}

int main()
{
    printf("%d\n",sum(5,1,2,3,4,5));
    printf("%d\n",sum(3,1,2,3));
}
```

使用可变参数列表必须使用`stdarg.h`头文件按固定的规则使用即可 注意 参数的类型和个数编译器无法检查 需要程序员自己指定 一旦出错 调试比较困难

另外 可变参数列表必须写在所有参数后面 而且至少要有一个有名的参数

传递到可变参数列表时 会发生一些隐式类型转换 例如 不足`int`的基本类型会拓展成`int`， `float`会被拓展成`double` 这也是为什么无论是`float`还是`double`在`printf`中都使用`%f`的原因[`scanf`不适用 它传入的是指针]

数组参数

数组在传递给函数时 数组类型会退化成指针类型 例如

```
int f(int a[],int n);
int g(int *a,int n);
```

二者是等价的 但这并不是说指针与数组是等价的 只有作为函数参数时 他们才是等价的 而且 数组退化成指针只退化一级 二维数组是不会退化成二维指针的 只会退化成指向数组的指针

早期的c语言只有第二种写法 第一种是后来加进去的 其实也只是为了更直观 不过也引入了一些新的问题

结构、联合、枚举

结构体

`struct`关键字用于定义结构体 例如

```
struct complex
{
    int re;
    int im;
};
```

定义了一个名为`struct complex`类型的结构 这里是为了表示一个复数

结构中也可以包含其他结构 例如

```
struct number
{
    int size;
    struct complex num;
};
```

结构不能包含自身 因为这样会引发一个递归定义 从而陷入无限递归 但可以定义一个指向该结构类型的指针 常见的例子既是链表

```
struct list
{
    int data;
    struct list *next;
};
```

结构定义后可以使用默认的`=`进行赋值 但是类型要一致 数组是不能进行赋值的 但是结构中的数组是可以的

```
struct arr
{
    int a[30];
};

int main()
{
    struct arr a={{1,2,3,4,5,6,7}};
    struct arr b=a;
    struct arr c;
    c=b;
}
```

但是结构体赋值仅仅是值拷贝 如果带有指针指向的资源时 要十分注意 它是不会拷贝指向的资源的 仅仅是拷贝了指针值 例如

```
struct X
```

```
{
    int size;
    int *ptr;
};

int main()
{
    struct X x,y;
    x.size=10;
    x.ptr=malloc(x.size*sizeof(int));
    y=x;
    x.ptr[5]=123456;
    printf("%d",y.ptr[5]);
}
```

访问结构中的内容 可以使用`.`或者`->` 前者用在普通对象上 后者用在指针类型上 例如

```
struct T
{
    int a,b;
};
void f()
{
    struct T a={1,2};
    struct T *p=&a;
    printf("%d %d",a.a,p->b);
}
```

结构体类型只认结构名 即相同内容不同名字的结构体不是同一类型的 例如

```
struct A
{
    int a;
    int b;
};
struct B
{
    int a;
    int b;
};
```

这两个类型是不同的 尽管他们内部结构相同 他们之间不能相互赋值

c99支持结构体的指定初始化

```c
struct T
{
    int a;
    int b;
};
void f()
{
    struct T x={.a=1,.b=2};
}
```

c99也支持**复合字面量** 只需指定列表的类型

形式为 `(类型){列表内容}` 列表内容可以使用指定初始化的形式 例如

```c
struct A
{
    int a;
    int b;
};

int main()
{
    struct A x;
    x=(struct A){.a=2,.b=9};
    int *p=(int[]){1,2,3,4,5};
    int y=(int){10};
}
```

虽然大部分类型都可以指定 不过一般只用在结构中 他相当于建立一个临时变量

结构名的`struct` 不可去掉 但可以通过`typedef`定义一个类型别名 `typedef`相当于是一种声明 可以在类型未定义之前即使用 例如

```c
typedef struct list List;
struct list
{
    int data;
    List *next;
};
typedef struct list Node;
typedef Node* Node_ptr;
typedef int Data_Type;
```

结构体中的名字空间是独立的 不用担心名字冲突的问题

使用结构的主要目的是对数据进行一定程度的封装 用以简化程序的思路 便于看懂

联合体

联合体又称共用体 或者共用的名字起得更好 定义方式与结构体类似 但区别在于内部元素的存储

`union`关键字用于定义共用体 定义共用体的主要原因是 某些数据 存储了其中一个后 另一个就不需要存储了 而他们的类型又不一致 如果把他们都定义了 浪费资源 例如

```c
union T
{
    int a;
    float b;
};
void f()
{
    union T x;
    x.a=10;
    printf("%d",x.a);
    x.b=12.25;
    printf("%f",x.b);
}
```

由于联合体一定程度上破坏了类型系统 使用时其实是容易出错的 因为一个内存块的类型需要由程序员自己去确定 编译器无法检查出此类错误 根据经验来看 联合体的使用频率是极低的

联合体的使用最好对什么类型赋值 就只用那个类型 有一种常见的用法

```c
union T
{
    int a;
    float b;
};
void f()
{
    union T x;
    x.b=12.25;
    int i=x.a;
    printf("%x",i);
}
```

用此种方式试图获取浮点数12.25的二进制表示 但是 不能保证`int`和`float`的字节数相同 这段代码是不可移植的

尤其是字节数不同的情况下 错误访问的行为是不确定的

枚举

`enum`关键字用于定义枚举 c语言中 常常用枚举来定义常量 例如

```c
enum{ONE=1,TWO,THREE};
enum CHAR{ALPHA,BETA};
void f()
{
    int x=THREE;
    enum CHAR y=BETA;
}
```

枚举可以指定一个值 也可以不指定 不指定的 其值比前一个大1 如果第一个不指定 那么其值为0

如 以上`TWO`的值为2 `ALPHA`的值为0

上述两个例子分别为无名枚举和有名枚举 一般常用匿名枚举定义常量

编译过程

c语言的编译过程分为 预处理、编译、汇编、链接 其中分别使用预处理器、编译器、汇编器、链接器

一般情况下 整个过程均称之为编译

这四个过程中 会分别把源代码处理成 预处理之后的文件、编译后的汇编文件、目标文件和可执行文件

预处理

预处理器会删除注释 展开头文件 展开宏定义 处理条件编译

编译器并不需要注释 注释是用于描述代码的 方便维护 所以预处理之后的代码是不包含注释的

除了处理注释 预处理器还会展开头文件 例如`#include<stdio.h>`会将`stdio.h`文件内容**原封不动地拷贝**到包含他的文件中 位置处于原来的`#inlclude`处

头文件包含分为两种`#include<...>`与`#include"..."`两者略有不同 一般来说 前者包含标准库头文件 后者包含自定义的头文件 [前者只往库文件夹中搜索 后者先往当前目录搜索 如果没有找到 则再往库文件夹中搜索]

宏定义与条件编译在预处理过程中比较重要

```
#define MAX_SIZE 100
#define max(a,b) (a)>(b)?(a):(b)
#define DEBUG
#undef DEBUG
```

最简单的宏定义如`#define DEBUG`它表明 宏定义了`DEBUG`而它什么都表示 一般用于条件编译中

定义常量的另一种方式是`#define MAX_SIZE 100`预处理器在遇到`MAX_SIZE`这个名字之后 会原封不动地使用后面的内容替换 这也带来的一些问题 考虑以下这类情况

```
#define NUM -100

void f()
{
    int x=5-NUM;
}
```

按宏展开的策略 它会展开成`int x=5--100;`这时 按最长匹配原则`--`和在一起作为运算符 即自减 显然这是不对的 所以 一般来说 宏定义都会使用`()`包围常量 写作`#define NUM (-100)`

宏也可以模仿函数 一般用在避免调用函数开销的地方 **不过有内联函数之后 它的作用就小了许多**

同样的 它也是基于**文本展开**的 以上的例子中

```
#define max(a,b) (a)>(b)?(a):(b)

void f()
{
    int x=10,y=20;
    printf("%d",max(x,y));
}
```

`max(x,y)` 将会展开成 `(x)>(y)?(x):(y)` 加括号的原因是为了防止出错 因为宏只是文本展开 而不是调用函数

考虑以下例子

```
#define mul(x,y) x*y

void f()
{
    int x=mul(2+3,4+5);
}
```

将宏展开 得到 `int x=2+3*4+5;` 该表达式显然与原意图不一致 而加上括号后 就没有问题了

```
#define mul(x,y) (x)*(y)

void f()
{
    int x=mul(2+3,4+5);
}

#define max(a,b) (a)>(b)?(a):(b)
void g()
{
    int x=10,y=20;
    int z=max(x++,y);
}
```

加上括号也不能保证万无一失 将代码展开 `int z=(x++)>(y)?(x++):(y);` 这里出现了对x的两次自增操作

条件编译在头文件机制中是不可避免的 它用于防止重复包含 一般的头文件 都有这样的一种格式

例如test.h文件

```
#ifndef TEST_H
#define TEST_H
...
#endif
```

条件编译指令有 `#if` `#endif` `#ifdef` `#ifndef` `#else` `defined` 等

在条件不成立的情况下 预处理器会将其中的代码删除

上面的例子中 可以这样解释

```
如果没有宏定义 TEST_H
宏定义 TEST_H
...
结束上一个if宏
```

这是头文件中必须要用的条件编译指令

另外一点 在去除代码时 一般不会直接删除 采用 `/*...*/` 的方式注释代码也有可能出现问题 可以采用条件编译

```
#if 0
void g();
void f();
...
#endif
```

gcc可以使用`gcc -E test.c > test.i`指令来查看预处理后的代码

编译

编译过程将预处理后的代码翻译成汇编 优化的过程也是在这完成的

可以通过`gcc -S test.c`来生成编译后的汇编代码

例如

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

这段代码将会被翻译成

```
    .file   "test.c"
    .text
    .globl  f
    .def    f;  .scl    2;  .type   32; .endef
    .seh_proc   f
f:
    pushq   %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe   %rbp, 0
    .seh_endprologue
    movl    %ecx, 16(%rbp)
    movl    %edx, 24(%rbp)
    movl    16(%rbp), %edx
    movl    24(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
    .seh_endproc
    .ident  "GCC: (tdm64-1) 5.1.0"
```

这是未优化的结果 优化后

```
    .file   "test.c"
    .section    .text.unlikely,"x"
.LCOLDB0:
    .text
.LHOTB0:
    .p2align 4,,15
    .globl  f
    .def    f;  .scl    2;  .type   32; .endef
```

```
    .seh_proc   f
f:
    .seh_endprologue
    leal    (%rcx,%rdx), %eax
    ret
    .seh_endproc
    .section    .text.unlikely,"x"
.LCOLDE0:
    .text
.LHOTE0:
    .ident  "GCC: (tdm64-1) 5.1.0"
```

可以看到 函数`f`中的指令大大减少

汇编

将汇编代码汇编成机器码 这一过程会生成目标文件 虽然他是二进制文件 但还不能执行

通过`gcc -c test.c -o test.o`可以生成目标文件

链接

链接器将目标文件链接成可执行文件 可能会链接其他目标文件 或者链接库文件等

一般来说 编译器是默认链接标准库的

多文件编译

实际的项目中 不会只有一个单一的文件代码 常常是多个文件

将项目拆分成多个文件 有几个好处

1. 尽量让每一个代码文件做同一类的事 之后再合并 方便维护
2. 某个文件代码的修改不需要重新编译其他未改动的文件

例如

`mian.c`

```c
#include<stdio.h>

extren int add(int,int);//声明add函数 至于在哪 交给链接器
int main()
{
    int x=add(10,20);
    printf("%d",x);
}
```

`add.c`

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

再将两个文件编译成目标文件

`gcc -c main.c`

`gcc -c add.c`

再链接两个文件

`gcc main.o add.o -o test.exe`

当然 也可以直接写成`gcc main.c add.c -o test.exe`

这样 有两个代码文件可以生成一个可执行文件

集成开发环境[IDE]中 通过建立项目 并添加这两个文件后编译 也可以实现多文件编译

另外 像`extern int add(int,int);`这类声明的语句 一般是放在头文件中的 例如建立

`add.h`

```
#ifndef ADD_H
#define ADD_H

extern int add(int,int);

#endif
```

将他们放在同一个文件夹中 便可以使用了

`test.c`

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

int f()
{
    printf("%d",add(10,20));
}
```

多文件编译的外部变量引用

全局变量可以被其他文件引用 只要全局变量没有被`static`修饰

`static`关键字修饰全局变量和函数时 表明 他们只能被当前文件使用 对其他文件来说是不可见的

`static`修饰函数内部的变量时 表明他是静态变量 相当于全局变量 但只能在函数内部访问

引用其他文件中的全局变量 只需声明即可 例如

`main.c`

```
#include<stdio.h>
extern int x; //声明x
int main()
{
    printf("%d",x);
}
```

`foo.c`

```
int x=1024; //定义x
```

将他们编译链接后 即可使用

链接过程由链接器完成 也就是说 链接时类型已经丢失 链接器只负责名字相同 如果类型不一致 将会引发程序错误 而不被编译器检查到

一个常见的错误 也是对数组和指针的误解 数组和指针并不是相同的

`main.c`

```
#include<stdio.h>
extern int *A;
int main()
{
    A[0]=1024;
}
```

`foo.c`

```
int A[1024];
```

编译链接后 将会发生错误 程序将崩溃

正确的声明方式是`extern int A[];` 不需要指明长度 只需说明他是数组 而不是指针

出错的原因在于 数组名和数组的首地址是相同的 而指针的访问 它是将存储在那个位置的值读出来后 认为是个地址 再对这个地址进行寻址 例如

地址	**0x1000**	**0x1004**	**0x1008**	**0x100c**
值	0	0	0	0

例如数组首地址是0x1000 名字为`A` 那么他按0x1000开始寻址 [数组名的值规定与它的首地址相同]

而现在误认为`A`是个指针 那么将读取`0x1000`处的变量 得到`0` 然后再按`0x0`开始寻址 而这块地址是非法的 故程序会崩溃

除了类型不一致的问题 引用外部变量还有一个更容易出错的问题 这里先说明**强符号与弱符号**

1. 初始化的全局变量和定义的函数都是强符号
2. 未初始化的全局变量是弱符号

链接器有这几个规则

1. 不允许有多个强符号
2. 如果有一个强符号和多个弱符号 那么选择强符号
3. 如果有多个弱符号 那么任意选择一个

如果不小心起了相同名字的弱符号 那么在引用时就可能会出错 而这个问题又难以检查 例如

`main.c`

```c
#include<stdio.h>
int x;
int y;
extern int f();
int main()
{
    printf("%d\n",x);
    f();
    printf("%d\n",y);
}

```

`foo.c`

```c
int x=1024; //强符号
int y;
int f()
{
    y=2048;
}
```

正常情况下 未初始化的全局变量默认初始化为0 而运行结果 会是超出预期的1024和2048

第一个是由于使用了强符号 而第二个是不可预测的 故对全局变量的使用要十分小心 如果不希望其他文件能访问到该全局变量 应用`static`修饰 例如

`static int y;`

静态库与动态库

像c标准库 链接的时候不是全部将所有函数链接 而是用到哪个函数链接哪个部分 这个普通的目标文件是无法实现的 目标文件会将全部代码和内容链接 而静态库只链接用到的

通过`ar -cr libfoo.a foo.o`即可创建一个静态库

链接静态库使用`gcc -static main.o -lfoo`其中`-static`表示强制链接静态库

链接静态库的一个缺点是 如果修改静态库 必须重新链接一次库 重新发布新程序

而动态库只需重新发布动态库 不需要重新链接 因为动态库的调用是在调用到库中的内容后才会真正加载 然后再执行

`gcc -fPIC foo.c -o foo.o`生成位置无关代码

`gcc -shared foo.o -o libfoo.so`产生动态库

如果该库不放在默认位置 链接时需要指定动态库目录

通过`gcc -Wl,-rpath=dir main.o -lfoo`来生成最终程序`dir`表示的具体的目录 当前目录用`.`表示

运行过程

字符串

输入输出

异常处理

gcc拓展语法

数据的表示与运算

优化程序性能

类型系统

程序的机器级表示

利用c语言实现面向对象编程

常见的问题