


CHD_ACM第二次讲课：

C语言基础(二)

- 
1. 数组
 2. 函数
 3. 结构体
 4. 时间/空间复杂度

1.数组定义：

同类元素的集合，称为数组

一维数组的定义形如 `T Name[size]`

- T: 类型名，如int型，double型，指明数组中存的数据的类型
- Name: 数组名
- size : 常量表达式，表示数组的大小

```
int a[10];    //定义含有10个元素的一维数组，数组保存数据类型为int型，数组名为a
```

```
double b[5]; //定义含有5个元素的一维数组，数据类型为double型，数组名为b
```

```
char c[20];  //定义一维字符数组，数据类型为char
```

数组赋值：

1. 初始化：

```
int a[3]={0,1,2};
```

```
int b1[]={0,1,2};
```

```
int b2[5]={1,2,3};
```

```
char s []="Alan";
```

注意：只有在定义数组时才能整体赋值

2. 逐个赋值：

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

```
int a[10];  
for(int i=0;i<10;++i) //逐个赋值  
    a[i]=i;  
  
for(int i=0;i<10;++i) //接受输入的数据给数组元素  
    scanf("%d",&a[i]);
```

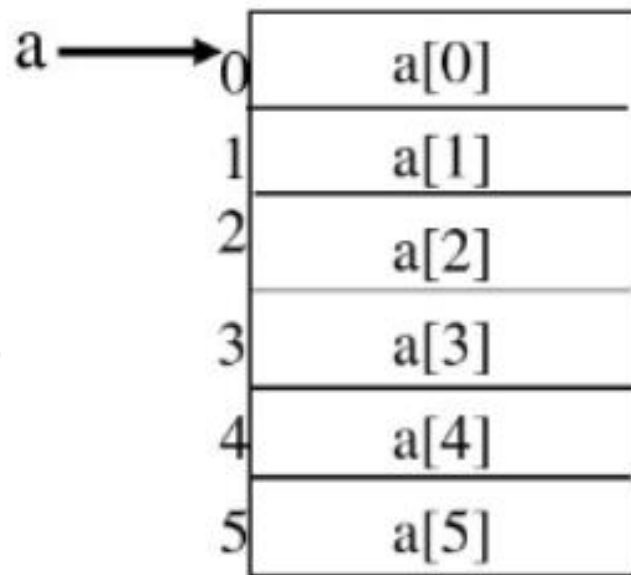
访问数组元素：

一维数组的下标

数组的下标实际上是关于数组第一个元素的偏移量。
所以数组元素为 $a[0] \sim a[\text{size}-1]$

```
int a[6];
```

- 数组大小固定，在访问数组元素时要注意防止数组下标越界（下标值溢出），即访问到数组合理范围以外的值。如定义 $a[10]$ ，访问 $a[100]$ ，就会发生数组越界。



访问数组元素：

一维数组的遍历：

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

    int a[5] = { 0,1,2,3,4 };

    for(int i=0;i<5;++i)
    {
        printf("%d",a[i]);
    }
    return 0;
}
```

多维数组：

二维数组的定义：

数据类型 数组名[常量表达式][常量表达式]

```
int a[2][3];
```

二维数组的初始化：

```
int a[2][3]={ {2,3,5} , {6,1,2} };
```

下标	0	1	2
0	2	3	5
1	6	1	2

多维数组：

二维数组的遍历：

用for循环遍历二维数组，先遍历第一行元素，再按列依此次遍历每一行

```
for(int i=0;i<2;++i){  
    for(int j=0;j<3;++j){  
        printf("%d",a[i][j])  
    }  
}
```


字符数组：

字符数组的定义和初始化：

数组中的各个元素均为字符变量，每个元素只存放一个字符

字符数组结尾以 `'\0'` 结尾，`'\0'` 要占一位。

```
char c[5];           //定义字符数组
char ch[6]={"Hello"}; //字符数组的初始化,下面是数组内具体的
char ch[6]="Hello";
char ch[]="Hello";
```

H	e	l	l	o	\0
ch[0]	ch[1]	ch[2]	ch[3]	ch[4]	ch[5]

说明：

- 字符串在内存中，系统会自动加上 `'\0'` 作为字符串结束标记，字符串所占字节数为串中字符个数+1
- 程序中系统根据 `'\0'` 判断字符串是否结束，而不是通过数组长度。

字符数组：

二维字符数组的定义和初始化：

```
char fruit[][7]={"Apple","orange","Grape","pear"};
```

fruit[0]	A	p	p	l	e	\0	\0
fruit[1]	O	r	a	n	g	e	\0
fruit[2]	G	r	a	p	e	\0	\0
fruit[3]	P	e	a	r	\0	\0	\0

字符数组：

字符数组的输入输出：

- 输入/输出逐个字符时用格式符 `"%c"`

```
char c[8];  
for(int i=0;i<8;++i)           //逐个输入同其他类型数组  
    scanf("%c",&c[i]);  
for(int i=0;i<8;++i)           //逐个输出  
    printf("%c",c[i]);
```

- 输入/输出整个字符串时用格式符 `"%s"`

将数组当作字符串来输入，遇到空格，回车结束。

```
char c[8];  
scanf("%s",c);                  //必须是数组名，不用加&取地址符  
printf("%s",c);                 //必须是数组名，不能写成数组元素  
  
scanf("%s",c+1);                //从c[1]开始存字符串  
printf("%s",c+1);
```

- 用gets()函数输入，一次输入一整串，遇到回车结束

```
char str[10];  
gets(str);
```

- 用puts()函数输出，一次输出一整串

数组作为函数参数：

知道数组的首地址和长度就可以唯一确定一个数组，所以数组作为函数参数时的形参表要有有数组名(确定首地址)，和数组长度。

```
#include<stdio.h>

int a_sum(int a[],int num)           //计算数组元素和，数组作为函数参数
{
    int sum=0;
    for(int i=0;i<num;++i){
        sum+=a[i];
    }
    return sum;
}

int main()
{
    int a[3]={1,2,3};
    printf("%d\n",a_sum(a,3));       //调用a_sum()函数时传入数组名，也就是首地址
    return 0;
}
```

函数：

- 简单来说，**函数就是一段实现某个功能的代码。**
- 函数的存在实现了编程的模块化，更易于调试和维护
- 提高效率，减少重复编写程序的工作量，而且程序的可读性也很好。程序思路很清楚！！

举例：输入a,b,c三个int型的值，分别输出它们的阶乘

```
#include<stdio.h>
int main()
{
    int a,b,c;
    scanf("%d%d%d",&a,&b,&c);

    int ans=1; //a!
    for(int i=2;i<=a;++i){
        ans*=i;
    }
    printf("%d\n",ans);

    ans=1; //b!
    for(int i=2;i<=b;++i){
        ans*=i;
    }
    printf("%d\n",ans);

    ans=1; //c!
    for(int i=2;i<=c;++i){
        ans*=i;
    }
    printf("%d\n",ans);

    return 0;
}
```

```
#include<stdio.h>
int main()
{
    int a,b,c;
    scanf("%d%d%d",&a,&b,&c);

    get_fact(a); //a!

    get_fact(b); //b!

    get_fact(c); //c!

    return 0;
}
```

函数的定义，声明和调用：

函数定义：

```
返回类型 函数名(参数类型 参数名, 参数类型 参数名,...)
{
    若干条语句 (函数体)
}
```

```
void get_fact(int x)
{
    int ans=1;
    for(int i=1;i<=x;++i){
        ans*=i;
    }
    printf("%d\n",ans);
}
```

```
int Max(int x,int y)
{
    if(x>y)
        return x;
    else
        return y;
}
```

函数的定义，声明和调用：

函数的声明：

- 如果函数定义在前，调用在后，调用前可以不声明。
- 如果函数定义在调用之后，则必须在调用前声明函数的类型，名称和参数。
- 相当于要让编译器知道你有这个函数
- 声明格式：<类型> <函数名> (<参数列表>);

```
int max(int a, int b);
```


函数的定义，声明和调用：

函数的调用：

- 要调用一个函数，就要按照形参表传入形参表规定类型的实参，即调用上面 get_fact()函数，就要传入一个int 型的实参，对应int 型形参int x。
- 调用上面的max()函数就要传入两个int 型的数据。

```
printf("%d",max(5,10));
```

```
int a=5,b=10;
```

```
int c=max(a,b);
```

函数的

```
#include<stdio.h>

int max(int a,int b);           //函数声明
int get_fact(int x);

int main()
{
    int x=10,y=5;
    printf("%d\n",max( get_fact(5) , get_fact(10) ) ); //输出5!和10!的最大值
    return 0;
}
//下面是函数定义，在函数调用后面所以前面要声明
int max(int a,int b)           //求a,b间最大值函数
{
    if(a>b)
        return a;
    else
        return b;
}

int get_fact(int x)           //计算阶乘 x!
{
    int ans=1;
    while(x>1){
        ans*=x;
        x--;
    }
    return ans;
}
```

函数的参数传递：

形参和实参：

主调与被调函数间有数据传递关系，这就是有参函数，定义函数时 () 中的变量名为**形式参数**（形参），调用函数时 () 中的变量名为**实际参数**（实参）。

- 形参在调用函数前只是形式，不占用内存空间，在函数被调用结束时，形参所占内存单元也被释放。因此形参只有在该函数内有效。
- 实参可以是常量，变量，表达式，函数等。但无论是哪一种类型的量，在进行函数调用时它都必须具有确定的值。

```
int f(int x)
{
    return x*x+2*x+3;
}
```

$$f(x) = x^2 + 2x + 3$$

函数的参数传递：

值传递：

- 把主程序调用过程的具体数值（实参）复制给函数的参数（形参）。
- 实参和形参是不同的内存单元，形参数值的改变不会影响到实参。

引用传递：

- 在c中没有引用的概念，引用是c++的概念。引用相当于给原变量起了一个别名。
- 将引用作为形参，在执行主调函数中的调用语句时，系统自动用实参来初始化形参，这样形参就成为实参的一个别名，对形参的任何操作也就直接作用于实参。
- 实参与形参是同一变量，有相同的地址和内存空间。|

函数的参数传递:

```
#include<stdio.h>
void swap_1(int x,int y)
{
    int t=x;
    x=y;
    y=t;
}

void swap_2(int &x,int &y)
{
    int t=x;
    x=y;
    y=t;
}

int main()
{
    int a=5,b=10;
    swap_1(a,b);
    printf("%d %d\n",a,b);
    a=5,b=10;
    swap_2(a,b);
    printf("%d %d\n",a,b);
    return 0;
}
```

```
5 10
10 5
```

```
Process returned 0 (0x0)   execution time : 0.995 s
Press any key to continue.
```

函数嵌套与递归函数：

函数嵌套： 在一个函数中可以嵌套使用其他函数，遇到函数时先执行函数，返回值后再继续执行原主调函数

```
#include<stdio.h>

int Max(int x,int y)
{
    return x>y?x:y;
}

int Max_three(int x,int y,int z)    //求三个数中的最大值
{
    int max1=Max(x,y);              //在Max_three()函数中调用 max()函数
    int max2=Max(x,z);
    return Max(max1,max2);
}

int main()
{
    int a=1,b=2,c=3;
    printf("%d\n",Max_three(a,b,c));
    return 0;
}
```

函数嵌套与递归函数：

递归函数：

采用递归调用可以解决的问题是有限的，这些具有以下特点：

- 原问题可以分解为一个子问题，解决子问题要用到解决原问题的方法，按这一特点分解下去，每次出现的新问题都是原问题的简化的子问题（规模不断变小，解决形式不变）
- 最终分解的子问题是具有已知解的问题（这样递归规模是有限的，递归调用才能结束）

```
#include<stdio.h>
int get_fact(int x)
{
    if(x==1)                //最终子问题 1!=1
        return 1;
    else
        return x*get_fact(x-1); //5!可以分解为计算5*4!...
}

int main()
{
    printf("%d\n",get_fact(5) );
    return 0;
}
```

结构体：

实际应用中数据并不是都是同一类型的，更多时候是多个不同类型的数据成组出现。

如：一个学生的学号 (int /字符串型)，姓名 (字符串型)，性别 (char型)，年龄 (int 型)，成绩 (double 型) 等数据属于同一个学生。如果将它们单独定义为互相独立的变量，不能很好的反映这些数据之间的内在联系。希望有一种数据类型可以将这些信息存储在一个单元，就像统计信息时每个学生信息保存在一张表中，而不是所有学生名字一张表，学号一张表 ... 。

为了解决这一问题，c支持自定义类型，我们可以把这些数据组成一个组合数据 `sutudent_1`，在这个变量中包含 `num`，`name`，`sex`，`age` 等项。c++中还可以通过定义类来自定义新的数据类型。

num	name	sex	age	score
10010	XXX	M	18	87.5

结构体的定义与初始化：

结构体的定义：

```
struct 结构体名
{
    成员列表;
    ...
    ...
}变量列表;
```

```
struct Date
{
    int year;
    int month;
    int year;
}d1,d2;
```

//定义一个学生的结构体类型

```
struct student
{
```

```
    int num;
    char name[10];
    char sex;
    int age;
    double score;
```

```
}stu_1,stu_2;
```

```
student stu_3;
```

//结构体名：student

//下面就是成员列表，列出这个结构体中有哪些类型的变量。

//也就是student中有哪些属性

//这里是定义的变量列表，即student类型的变量。

//也可以单独定义，就像定义int类型变量一样，用结构体名去定义

结构体的定义与初始化：

结构体的初始化：

初始化：<结构体类型>变量名={表达式1, 表达式2, ..., 表达式n}

```
Date d2={2019,10,8};
```

```
student stu_4={1001,'XXX','M', 18 , 87.5} //定义时初始化stu_4
```

结构体的访问：

- 一般用成员成员运算符 `.` 来访问结构体类型变量的成员，它是所有运算符中优先级最高的。
- 对结构体变量的成员的操作和对变量的操作一样。

```
Date d1,d2;|
d1.year=2019;           //访问结构体变量成员
d1.month=10;
d1.day=8;
scanf("%d %d %d",&d2.year,&d2.month,&d2.day); //C 输入
printf("%d %d %d ",d2.year,d2.month,d2.day);  //C 输出

cin>>d2.year>>d2.month>>d2.day;              //c++输入输出
cout<<d2.year<<d2.month<<d2.day;
```

结构体数组：

定义结构体数组：

```
Date d[10];
```

- 通过循环结构和数组的使用可以方便地对大量结构体类型数据进行操作

```
for(int i=0;i<10;++i){  
    printf("%d %d %d\n",d[i].year,d[i].month,d[i].day);  
}
```

结构体类型作为函数参数:

值传递

```
数据类型 函数名(结构体名 形参名, 其他形参表)
{
    //函数体
}
```

引用传参

```
数据类型 函数名(结构体名 &形参名, 其他形参表)
{
}
```

结构体类型作为函数参数:

```
#include<stdio.h>
struct Date
{
    int year;
    int month;
    int day;
};

void date_print(Date d)
{
    printf("%d %d %d ",d.year,d.month,d.day);
}

int main()
{
    Date d2={2019,10,8};
    date_print(d2);
    return 0;
}
```

时间/空间复杂度分析

时间复杂度：

用来表示一个算法的理论上的耗时时间

(1) 时间频度：一个算法中的语句执行次数称为时间频度，记作 $T(n)$

(2) n 为问题规模，当 n 不断变化时，时间频度 $T(n)$ 也会不断变化。但有时我们想知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。一般情况下，算法中基本操作重复执行的次数是问题规模 n 的某个函数，用 $T(n)$ 表示，若有某个辅助函数 $f(n)$ ，使得当 n 趋近于无穷大时， $T(n)/f(n)$ 的极限值为不等于零的常数，则称 $f(n)$ 是 $T(n)$ 的同数量级函数。记作 $T(n)=O(f(n))$ ，称 $O(f(n))$ 为算法的渐进时间复杂度，简称时间复杂度。

时间复杂度的计算：

- 计算出时间频度
- 计算出 $T(n)$ 的数量级
 - 忽略常量，如果运行时间是常数量级，则用1表示
 - 忽略低次幂
 - 忽略高次幂系数

时间/空间复杂度分析

$O(1)$ 常数算法

```
int a=1,b=2,c=3;
```

$O(n)$ 线性算法

```
int sum=0;
for(int i=0 ; i<n ; ++i){
    sum+=i;
}
```

$O(n^2)$

```
int num1,num2;
for(int i=0; i<=n ;++i){
    num1 += 1;
    for(int j=1; j<=n; ++j){
        num2 += j;
    }
}
```

$O(\log^n)$

```
int i=1;
while(i<=n){
    i*=2;
}
```


时间/空间复杂度分析

$$O(1) < O(\log_2^n) < O(n) < O(n \log_2^n) < O(n^2) < O(n^3) < \dots < O(n^k) < O(2^n) < O(n!)$$

空间复杂度:

- 空间复杂度作为算法所需存储空间的度量, 记作 $S(n)=O(f(n))$
- 这里所说的存储空间不是指程序本身、常数和指针所需要的存储空间, 也不是指输入数据所占用的存储空间, 而是指**解题过程所需要的辅助空间**。

数据类型范围：

unsigned int 0~4294967295 4e9
int -2147483648~2147483647 2e9

unsigned long 0~4294967295 4e9
long -2147483648~2147483647 2e9

long long的最大值：9223372036854775807 9e18
long long的最小值：-9223372036854775808
unsigned long long的最大值：
18446744073709551615 //20位 1e19

__int64的最大值：9223372036854775807 9e18
__int64的最小值：-9223372036854775808
unsigned __int64的最大值：
18446744073709551615 1e19

代码风格与缩进

群文件



浅谈编码风格.pdf

```
#include<stdio.h>
int main()
{int a,b;
int ans=1;
scanf("%d%d",&a,&b);
for(int i=1;i<=a;++i)
{ans*=i;}
printf("%d\n",ans);
return 0;
}
```

```
#include<stdio.h>
int main()
{
    int a,b;
    int ans=1;
    scanf("%d%d",&a,&b);
    for(int i=1;i<=a;++i)
    {
        ans*=i;
    }
    printf("%d\n",ans);
    return 0;
}
```

谢谢！