一. 数组简介：

Ⅰ.初始化：

1. 指定数组的长度：如：float a[20];

2. 指定数组的内容：如：float a[3] = {1, 2, 3}; （使用大括号！）

注意事项：

①**当初始化列表中的值的个数少于数组元素个数时，剩下的元素都初始化为0**；

②当初始化列表中的值的个数**多于**数组元素个数时，编译器视为**错误**；

③可以让编译器自己决定数组的长度：如int days[] = {1, 2, 3};这样编译器就知道数组的长度了。　如果coder想知道数组的具体长度，可以使用：sizeof days / sizeof days[0]计算；

3. 数组的维度不能是变量，只能是常量！！（可以是字面量常量，也可以是宏定义常量，最好不要是const定义的常量，不然移植性差）

4. 初始化器（designed initializer）：可以指定特定的数组元素（类似于关键字）

格式：int arr[6] = {[5] = 212};　　(这样就指定了该数组的最后一个元素为212)

关于初始化器的一些问题 （考试专用，虽然很蠢）

①initializer后面的元素怎么办？例如a[5] = {[2] = 1, 2, 3}——这里会把index为2后面的两个元素初始化为2和3.

②有多次定义怎么办？例如a[3] = {1, [0] = 2};——按后面的来；

③如果我没显式指定数组的大小怎么办？例如a[] = {1, 2, [5] = 3 };——编译器会设置数组的大小为刚好容纳下所有元素的大小。

④没有涉及到其它元素怎么办？——统统初始化为0；

5. 一些建议：建议在显式地设置数组的长度是，最好使用符号常量而不是使用字面常量。——如#define SIZE 20

Ⅱ. 赋值：

1. C不允许把数组作为一个单元赋给另一个数组，除初始化以外也不允许使用花括号列表的形式赋值；换句话说：C中数组只能循环赋值或者普通地赋值。

2. 关于数组边界的问题：注意在C中编译器**不会检查数组下标是否使用得当**，在C标准中，**使用越界下标的结果是未定义的。**

二. 多维数组

（这里只讲二维数组）

1. 对二维数组的理解：以float rain[5][12]；为例：

看到这个数组，很自然而然的想法就是 rain是的每一个元素都是一个数组，这个数组

的每个类型又都是浮点（由此不难看出在C中不允许存在含有不同类型元素的数组）。

2. 二维数组的初始化：

```
double a[][] = {{……}, {……}, {……}};
```

三. 指针和数组

(**给指针赋值和给其他类型的变量赋值是两件完全不同的事情**)

1. 指针和数组的关系：数组名是数组首元素的地址。即a == &a[0];

2. 指针的加法运算：对于数组而言，指针加一指的是增加一个存储单元，这意味着**加一后的地址是下一个元素的地址。**

3. 许多计算机（包括PC和Macintosh）都是**按字节编址**，也就是说地址的单位是字节。

4. 思考：为什么C中的数组下标是从零开始?原来是与指针息息相关，结合1，2的知识不难看出*(a + n) == a[n]，这样就是一种非常和谐的情况，也是大家所愿意看到的。

5. &和*的优先级：**等同于**++运算符

> (这个可以详见这里：https://baike.baidu.com/item/运算符优先级/4752611?fr=aladdin#2)

四. 函数，数组和指针

（记住一个口诀：当数组被用作形参传入函数时，将会**退化为指针**；数组与指向数组的指针不同，数组还包含了具体元素的数目，这就是为什么我们需要在函数中另外传入一个维数的参数了。）

1. 有些时候我们需要向函数传递一个数组：比如搞一个函数，求各个元素之和，这时候我们应该传递的是数组第一个元素的地址。如：sum(a)；函数声明和函数头的形式为：double sum(int * a)或double sum(int a[])（**对于数组而言，后者用得较多**）

2. 如果知识仅仅传递了数组的地址，只知道每个元素的大小和首地址。为了获得整个数组，有以下两种方法：

①直接传递数组的长度：int sum(int * a, int len)

②传递数组的尾地址：这个传入尾地址是一门学问，主要有两点要考虑：一，适当利用数组的越界；二，利用运算符优先级的骚操作。

首先来看第一点：我们要传入两个地址，一个是首地址，第二个是尾地址，在这里我们采用越界一个元素的方法，end指向的元素实际上在数组最后一个元素的后面，但只要在下面的while循环中使用不等号就可以完美地解决这一个问题；注意C能保证数组的后面一个位置的指针任然是有效的指针。这样的做法更加简介，不然的话，又要减一又要等于，很麻烦。 再来看第二点，我们把循环中的做法给简化了，这里稍微分析一下：因为*的优先级和

++相同，而运算顺序（本书中称之为结合律）是从右向左，所以先把决定权交给++，++并不着急于计算，由于在本例中是后缀的形式，所以"先计算，后递增"。结果就是先将值加上去，再递增**地址**。

```
sum = sumf(a, a + SIZE);

int sumf(int * begin, int * end) {
 int total = 0;
 while (begin < end) {
 total += *begin++; //现在begin是一个移动的指针，另外一种就是数组本身
 }
 return total;
}
```

总结一下：在函数中传递一个完整的数组至少要有两个参数，共有两个方法。

五. 指针操作

1. 赋值：将地址赋给指针；地址有三种：①数组名②带&的变量名③另一个指针
注意将数组名赋给指针后，可以将该**指针作为数组名**使用:
double arr[] = {.....};　　double * pd = arr;　pd[2] = ...　　//合法
2. 取址和解引用
3. 和**整数**相加；递增；
4. 和**整数**相减；递减；
5. 指针求差：差值的单位和数组类型的单位相同。

（注意：不要讲*（指针名）作为一个普通的变量来看待，尤其是当其**未初始化**的时候。）

6. **比大小**

六. 使用const保护数据

1. 什么时候再函数中要使用指针？①程序在函数中改变数值时；②对于数组别无选择
2. 问题的引入：C一般都是按值传递数据，但唯有数组是按地址传递（俗称按**引用传递**，因为引用的是同一块内存），有时候我们不希望数组中的元素被改变，那我们应该怎么做呢？这里就要引入const的内容了。
3. const的使用场景：
①普通变量/数组：const arr[SIZE];　　注意从此以后对arr任何元素的修改，编译器都会**报错。**　　②常值指针（不能通过指针修改值）：const double * ptr;
（这种指针可以初始化为const和非const类型变量的地址都可以，因为无法修改值。与之相对的，对于一般指针，不能初始化为const类型地址，因为有可能会修改地址对应的值）

③常地址指针（不能指向其他地址）：double * const ptr;

④常值常地址指针（很显然了）：const double * const ptr;

4. 函数与指针

（注意函数在使用的过程中有一个传递（赋值可能更加形象一点）的过程。这是一个关键。） 回到问题，如果函数的意图是不修改数组中的值，那么就可以使用const关键字，不过他并不是抑制值的改变，而是在值改变时**抛出错误**。总的来说，就是将其视为常量。

函数原型和函数头的格式：int sum(const int a[], int n)

七. 指针和多维数组

1. 浅谈指针和多维数组的关系：假设有int a[3][4];

①a == &a[0]，*a == a[0]（然而a[0]存的是a[0][0]的地址，似乎没什么用）

②a[0] == &a[0][0]， *a[0] == a[0][0]

③**a == a[0][0] == *&a[0][0]

(小小总结一波:其实二维数组也只有这三对关系，很多人不知道先看哪里，打个比方：*&a[0][0],其实可以先看后面具体的数组a[0][0]，再来看前面的符号；对于高维数组也一样，不要被乱七八糟的定义混淆了。)

④用指针表示数组中的元素：*(*(a + m) + n) == a[m][n];

(这里有一个要注意，就是括号的位置，要时刻谨记**a和*a是一个地址**，只有地址才能解引用，并与整数相加)

2. 指向多维数组的指针：由于多维数组的元素本身就是数组，而且指针也有不同类型之分；那么怎么设置含有多个int（或者其他类型）数组的指针呢？——那就要引入新的方法了： 如：int (* pz)[2] //pz指向一个内含两个int类型值的数值

（注意其中的**小括号不能省**，不然就是声明一个指针数组了）

3. 向函数传递多（这里只讲2）维数组：这主要有两种方法：

①在主函数内设循环，不断调用函数，并传入一维数组：

```
for (i = 0; i < 3; i++)
 total += sum(junk[i], 4);
```

②直接传入二维数组：

```
sum_rows(junk, ROWS);

void sum_rows(int ar[][COLS], int rows) {……}
```

简单地分析一下，函数的调用比较简单，这里主要讲函数头部分：首先这个要分为两部分看：一，ar[]，这是一个**指针**,你方括号内给不给数字都无所谓，因为编译器会忽略；二，

int ar[][COLS]，这个代表这个指针的跨度是sizeof(int) * COLS。

（小贴士：使用typedef味道更佳哦）

4. 指针的兼容性：指针的兼容性很差，不同类型的变量允许隐式转换，但指针却不允许；

5. 关于高维数组的小小拓展：int sum4d(int (*a)[12][20][30], int rows)

在这里①先让指针存在，所以使用了*a（代替了a[]）②剩下的三个数字都不能省略；在这个例子　中，a**指向**的是一个12X20X30的int数组，而其本身不是一个数组。

八. 变长数组

1. 问题的引入：之前我们可以看到在写多维数组相关的函数的时候，我们只引入了一个关于列的数目的参数，这就导致我们只能处理相同行数的数组，而且，在C99之前，这是毫无办法的，对于不同行数的数组我们只能创建不同的函数。现在我们有了VLA，他允许我们用变量来代替数组的维度。

2. 函数头的形式：

int sum2d(int rows, int cols, int a[rows][cols])　　//这个rows和cols必须先定义后使用，而且缺一不可

3. 函数原型的形式：

int sum2d(int, int, int a[*][*]);　　//利用了C99中在函数原型中可以省略参数名的优势，但后面的维度必须要用*代替

4. 与普通数组的一些区别：普通数组的大小在**编译**时已经确定，而变长数组的大小延迟到程序**运行**时才确定。

（VLA解决的痛点就在于函数的某些维度的值是固定的；他还有一个含义就是，**使得C/C++中数组的长度可以不必是常量表达式或者是常量字面量或者是常量。可以用一个变量来代替数组的长度**）

九. 复合字面量

1. 字面量（literal）是除符号常量以外的常量。大家都知道5是int类型字面量，‘c’是char类型字面量，那么数组的字面量呢？这时就要引入复合字面量（compound literal）这一概念了。

2. 形式：(int [3]) {10, 20, 30}　　//当然也可以省略3，让编译器确定数组的维度

3. compound literal的优势：在信息传入函数前不必先创建数组。

十. 从复习题中得到的

1. 数组名是一种指针，也是一种常量。也就是说int a[5]; a++;不行！只能通过指针；

2. 普通变量和指针变量可以一起声明：float a, *b;

3. 不建议用数组给指针赋值，因为数组的跨度不同。可以使用类型转换强行转换，这个有点玄学：(int *) grid == &grid[0][0]

4. 给新手的坑：数组的第n个元素：数组名【n-1】

十一. 数组使用的技巧

1. 数组的各种写法：

①char * p[2]:指针数组，供3个（暗示矩阵不能超过3行）专门用来存储**字符串数组**，比如p[0]是一个字符串，p[1]又是一个字符串，由于最下面一层省略了长度，所以每个p指向的字符串长度可以不一样。注意此时p[n]++指向的是对应字符串的下一个字符。p++没意义，因为p是常量

②char （*p）[2]:跨度为2个int（暗示矩阵不能超过2列）的指针，把二维数组理解为矩阵的话，p指向的是行，所以p++指向的是下一行数（下一个数组）。

③char **p:指针的指针，这个对数组而言没什么用处，因为它省略了列和行的内容。

④char p[][5]: 这个表示多见于函数原型和函数头，它等价于char (*p)[5];

(不管是什么样子的表示方法，都可以用数组表示法表示特定地址的内容，如p[1][1])

2. 注意数组在声明的时候一定要是常量表达式，不能是变量！