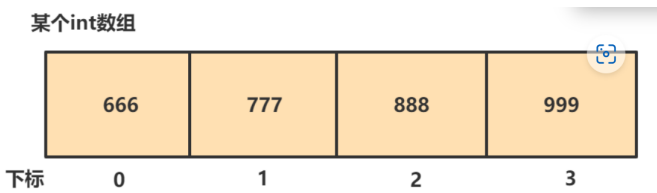
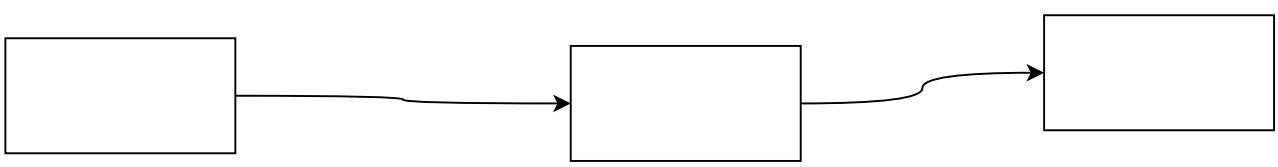
1. 逻辑结构：描述数据元素之间的逻辑关系。常见的如：
   a. 线性表：其中元素之间存在一对一的关系。
   b. 树型结构：一个层次结构的节点集合，其中n个节点（n≥0）有明确的上下级(父子)关系。
2. 物理结构：描述数据元素如何在存储介质上物理存放。例如：
   a. 数组：线性表的一种常见物理实现，要求元素在内存中连续存储。
   b. 链表：线性表另一种常见物理实现，元素在内存中可以不连续，但每个元素指向下一个，形成链式结构。

数组是线性结构的典型物理实现，它是用一片连续的内存空间来存储元素数据




链表：也是线性结构的一种典型物理实现



**数组的优缺点：**
**优点：**
**数组的最大优点是支持随机访问，访问的效率特别高，时间复杂度是0(1)**

**数组是怎么实现随机访问的?**
**简单来说是通过“寻址公式”来计算对应下标位置元素的地址，从而实现随机访问**
**实现随机访问是有前提的：**
**1.必须是连续的内存空间。数组满足**
**2.你要知道数组中元素的起始地址，基地址，也就是第一个元素的地址。（重点）在C语言中，数组名在内存中就代表数组的基地址，也就是首元素的地址。**
**3.要想进行寻址，进行连续运算，需要每个存储单元的大小保持一致。当然数组仍然满足，因为数组中只能存储单一类型变量。**

**寻址公式：**
**addr(arr[i]) = base_addr + 偏移量(offset)**
**偏移量是目标元素的地址和数组首元素的地址的字节差值，实际上它恰好等于：**
**sizeof(元素) * i**
**addr(arr[i]) = base_addr + 偏移量(offset) = base_addr + sizeof(元素) * i**

假如我们有一个长度为5的int数组，其基地址是0x0053fbd4，要访问第五个元素，即arr[4]，那么根据寻址公式就可以计算出地址：

address_arr[4] = 0x0053fbd4 + (4 * 4) = 0x0053fbe4

所以，每次你用"arr[4]"这样的语法访问数组时，程序会直接访问0x0053fbe4这个地址来获取或设置元素值，不再需要查找或遍历，效率非常高，这就是随机访问的魅力。

**数组的缺点：**
**1.对内存空间的要求非常高，必须使用一片连续的内存空间。这在很多时候是做不到的，尤其是小型机器或者内存紧张的情况下。**
**2.数组不灵活，由于它对内存空间的要求很高，导致在C语言中，普通的数组是没有办法自由扩容和缩减的。数组一旦完成初始化，大小和长度都是固定的。**
**3.数组中存储的元素的类型必须是一致的。**

**最后一个问题：**
**为什么偏移量的计算，可以直接用目标元素下标i乘以元素的大小?**
和补码统一加减法类似，这里将下标设置为从0开始，就减少了一次减法操作，大大提升了效率。

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

/*
* 一维数组
* C语言的普通数组,在声明时
* 它的长度必须是一个在编译时期就确定值的常量
* 1.字面值常量(包括它的表达式)
* 2.宏常量
* 宏常量是规范C代码最推荐使用的数组长度定义
*
* const声明的常量,不是在编译时期确定大小的,所以它不能作为数组长度
* const常量,更好的叫法是"只读变量"
*
* C语言中,不允许数组的长度是0,C语言没有空数组
*
*/
#define ARR_LEN 5
int a[5];
int main(void) {
// 一维数组的声明
   // 如果arr数组声明在函数体内部，是一个局部变量数组，那么仅有声明的数组全部元素都未初始化
   // 此时元素的取值是随机的，使用这样的数组会引发未定义行为。
   //int arr[5];

   //int len = 6;
   ////double arr2[len];    // error

   //double arr2[ARR_LEN];

   //const int len2 = 10;
   //len2 = 20;
   //char arr3[len2];

   int b[3];
   // 一维数组的初始化
   int arr[3] = { 0, 1, 2 };    // 完整初始化[0, 1, 2]
   int arr2[] = { 0, 1, 2, 3 };    // 编译器自动计算数组长度,此时长度=4
   int arr3[4] = { 0, 1 };    // 部分初始化,未给出的元素取值是0值
   int arr4[5] = { 0 };    // 数组的全部元素都是0值

   // int arr4[5] = {}; error
    //int arr4[1] = {1,2}; error

   return 0;
}
```

**二维数组的本质：**

**从数学角度看，二维数组就是数学中的矩阵。将它理解成一个有行有列的矩阵，可以方便理解二维数组的元素访问**

**arr[3][4]就是访问矩阵第3行，第4列的元素**

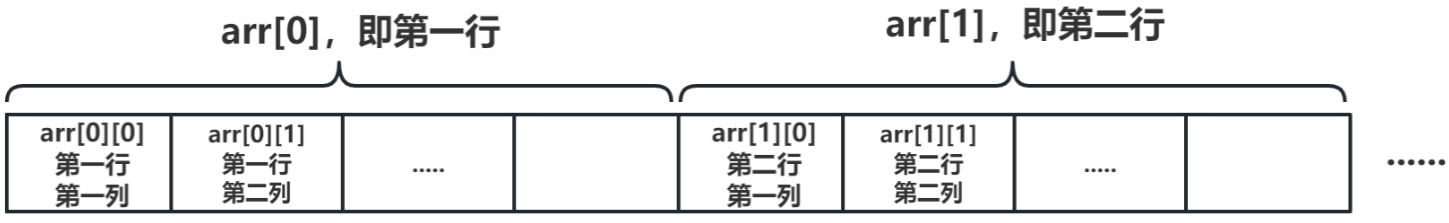
**从内存角度，二维数组是"元素是一维数组的一维数组"。如下图所示**


**上图表示一个二维数组arr[row][colum]**
**其实内存中并没有二维数组的独特结构，还是一个一维数组**
**只不过这个一维数组的元素是另一个一维数组**
**而且C语言的二维数组，仍然是连续的**
**二维数组的第一个元素arr[0]，它是一个一维数组**
**arr[1]也是一个一维数组**
**而且它们在内存中连续，先存储arr[0]，再存储arr[1].....**
**同样，二维数组的名字在内存中代表：**
**也代表首元素的地址**
**其实就是二维数组当中，第一个一维数组的第一个元素的地址**

**这种存储二维数组的方式，称之为“行主序”，就是先存一行，再去存下一行**
**二维数组的行：arr[i][j]中的i，表示二维数组中的第几个一维数组**
**二维数组的列：arr[i][j]中的j，表示每个一维数组的第几个元素**

**二维数组初始化时 行可以省略，列为什么不行呢?**
**1.“行主序”存储，第一行存完，才能存第二行，所以必须要知道一行有几列，不然存不了。**
**2.（主要）为了实现二维数组的随机访问，所以列不能省略，而且列必须时一样的。每个一维数组的长度必须是相同的**

**addr(arr[i][j]) = base_addr + (偏移量)**
**base_addr 就是第一个一维数组中首元素的地址**




**偏移量咋算?**
**arr[1][3] 咋算偏移量?**
**每个一维数组的偏移量= (i * column )* sizeof(arr[0][0])**
**再加上j的3**
**总偏移量是 =  (i * column + j )* sizeof(arr[0][0])**

**以上计算的前提是:**
**每一个一维数组的偏移量必须是一样的，不然就不能计算**

**可以理解成:**
**arr[i][j]是元素是arr[j]一维数组的二维数组，其中arr[j]的一维数组有i个**
**arr[j]就是二维数组的元素，数组中只能存储同类型元素，所以j必须确定，却所有一维数组j都是相同的**

**arr[i][j]如果它是在二维数组声明的时候使用:**
**它创建的是一个i行j列的矩阵,二维数组,这意味着二维数组当中有i个一维数组,每个一维数组的长度是j**

**arr[i][j]是在访问二维数组的元素时使用:**
**它表示访问i行j列的元素,但这个矩阵是从第0行,第0列开始的**
**它访问的是二维数组中的第(i+1)个一维数组的,第(j+1)个元素**

**在C语言中,利用[]下标访问数组元素**
**是不会自动检查数组的下标是否合法的**
**合法的数组下标应该在[0, len-1]**
**所以一旦访问非法下标,就会引发未定义行为**

**一般不会导致程序崩溃出错,会访问到一个莫名其妙的值,或者随机值**

```
返回值类型 函数名(形参列表){
   // 函数体
}
```

**C语言的函数定义中的细节问题:**
**1.返回值类型:**
**a.数组是不允许作为C语言函数的返回值类型,但是C语言完全可以返回数组作为返回值(通过指针实现)**
**b.在C90当中,C标准允许函数没有返回值类型,此时默认它的返回值是int.但C99舍弃了这种做法,现在C编程不推荐这么做**

**函数名:**
**见名知意,函数名最好用一个动词**
**getxx, handlexxx...**
**C语言当中,函数名就是同一个文件当中,函数的唯一性标识.同一个文件中,不允许出现多个相同名字的函数定义**
**但是可以出现多个相同名字的函数声明**

**形参列表:**

a. 形式参数（形参）组成的列表。形参的语法形式为："数据类型 形参名"，多个形参间用"," 逗号 分隔。

b. 形参列表可以是空的，表示函数不需要任何外部输入。在C语言中，推荐使用void取代空形参，以明确地表示函数不接受任何参数。

c. 形参列表中的数据类型，决定了调用函数时所需的实际参数的类型。例如，int a表明需要一个整数型参数。

d. 形参列表中的形参名，决定了参数传入函数后，如何去使用此参数。

e. 形参列表中的数据类型、形参个数、顺序都会影响此函数的调用，而形参名则不会影响函数调用。

## 函数的定义位置影响函数调用

**C编译器是从上到下逐行编译代码的。**

**当编译器遇到函数调用语句时，它会查找该函数的声明或定义。如果在调用处之前未找到函数的声明或定义，编译器将报错，因为它不知道这个函数的细节。**

**但在VS的MSVC平台下,编译器会特殊处理:**

**它会先假定该函数的返回值是int类型, 然后再去下面找该函数的声明或定义, 如果恰好下面有此函数的定义/声明且函数的返回值刚好是int,那么MSVC允许这种行为**

**总之，应对针对C语言的这一特性，建议采用以下策略：**

1. **始终将函数定义放在其调用之前。**
2. **在函数调用前使用函数的声明语法，事先声明此函数**

局部变量

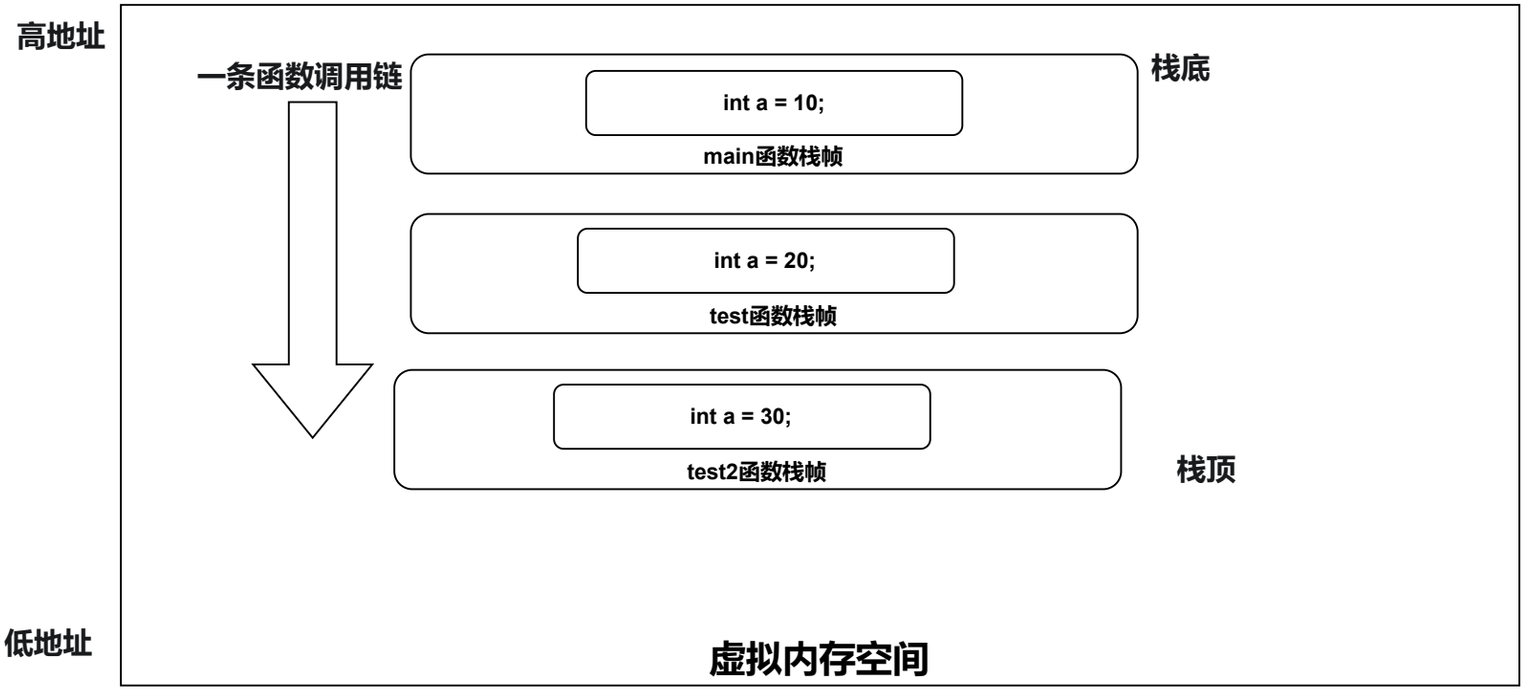
操作系统为每个执行的C程序（进程）分配虚拟内存空间，局部变量存储在一片被称之为"栈（stack）"的内存区域。

栈的工作原理是**后进先出**，这意味着最后放入栈的项是第一个被移除的。

在C程序的运行过程中，每当一个函数被调用，系统会为其创建一个**“栈帧(frame)”**来存储该函数的执行上下文，并将这个栈帧压入栈底部，这个过程称为**函数进栈（push）**。一个函数被调用，就是函数栈帧进栈的的过程。

栈帧中会存储此函数的局部变量（包括形式参数）。

当函数开始执行，对应的栈帧被压入栈顶时，局部变量得以初始化并生效。随着函数执行完毕，栈帧从栈顶中弹出，此时，函数内的局部变量也随之被销毁，这个过程称为**函数出栈（pop）**。




局部变量的生命周期:

局部变量的生命周期与包含它们的函数的生命周期是一致的：

1. 当函数被调用时，其局部变量被创建；
2. 当函数返回结束时，这些变量被销毁。

粗俗点说,局部变量就和装它的函数栈帧同生共死!

在C语言中,把这种跟随存储空间单元同生共死的生命周期,叫做"自动存储期限"
为什么叫自动?因为不手动
因为程序员不需要主动控制函数调用过程中,函数栈帧的进出栈
所以局部变量的生命周期是程序自动管理的,栈空间是自动管理的

重要:
在虚拟内存空间中,栈是非常重要的,它的"先进后出"特点,就决定了C程序函数调用的顺序
越早调用的函数越晚结束, 越晚调用的函数越早执行 完毕
最早调用的函数是主函数,它最后结束,因为它结束意味着进程退出.

处在栈顶部(靠近低地址的位置)的函数栈帧被称之为当前栈帧,也就是目前程序正在执行的函数

全局变量,也经常被称之为"外部变量"
你可以这里理解:
1.它不定义在函数体内部,所以是外部变量
2.它不仅可以在定义它的整个文件中使用,还可以在声明它的外部文件中使用,所以叫外部变量

test.c文件: extern int global_variable; // 变量的声明,但不是变量的定义
main.c文件: int global_variable = 10; // 变量的声明和定义,当然可以直接叫定义

使用全局变量,尤其是跨文件使用全局变量,要搞清楚此全局变量是在哪个文件中定义的
对于全局变量而言 声明可以有多次,但是定义只能有一次

全局变量被存储在虚拟内存空间当中，一片被称之为"数据段"的内存区域当中。

不同于局部变量随着函数的调用和返回被创建和销毁，全局变量的生命周期从程序开始执行时开始，持续到程序完全结束。

简而言之，全局变量与程序的生命周期同步：它们在程序开始时被创建并初始化，并在程序结束时被销毁。

在C语言中，这种持续存在于程序整个执行周期的生命周期特性被称为“静态存储期限”。

在C语言中,除了局部变量具有自动存储期限,其余变量都是静态存储期限,都是在整个程序的运行期间都生效

当C程序开始执行时，会在程序加载时为数据段中的变量进行初始化，包括全局变量。此时：

1. 如果全局变量已由程序员明确初始化赋值，该初始值会直接在程序启动时分配给它。
2. 若程序员未进行显式初始化赋值，系统则会为其设置默认值（也就是0值），如：整型和浮点型变量默认值为0，指针变量默认值为NULL。

注意：

1. 除非不确定此全局变量的初始值，否则建议对其进行手动初始化。
2. 全局变量的初始化有且仅有程序启动时的一次。

实际上所有的静态存储期限的变量, 都具有这种默认初始化0值的过程, 这些变量即便不手动初始化,也是可以直接用的

用的是默认0值