一. 存储类别

Ⅰ. 基本概念

1. 对象：用来储存程序所需数据的内存叫做对象；

2. 标识符：一般而言，变量名是一种标识符，在声明变量后我们就可以通过标识符访问相应内存了。

3. 翻译单元（translation unit）：在编译时，编译器会把源代码中的预处理器指令，如#include<...>，替换为相应的头文件。所以，编译器源代码文件和所有的头文件都看成是一个包含单独信息的独立文件，这个文件被称为翻译单元。

4. 作用域：作用域描述程序中可访问标识符的区域（空间概念），作用域有以下四种：①块作用域：声明在内层块中的变量，其作用域仅局限于该声明所在的块

②函数作用域：这个与goto语句相关，这里就不提了。

③函数原型作用域：范围是从形参定义出到原型声明结束——这说明函数原型中的形参名无关紧要。

④文件作用域：从定义处到该定义所在文件的末尾均可见。这个又可细分为具有外部链接的文件作用域和具有内部链接的文件作用域；

（从上面的内容我们不难想到，本书主要讨论的是**块作用域**和**文件作用域**的变量）

5. 链接：链接这个概念通常和作用域绑定在一起，链接有以下三种：

①无链接：块作用域，函数原型作用域，函数作用域都是无链接变量。

（内部链接和外部链接的概念专门用来细分**文件作用域**的变量）

②内部链接：具有内部链接的变量只能在一个翻译单元内使用

声明：(加上static关键字)static int a = 3;

③外部链接：具有外部链接的变量可以在多个翻译单元内使用

声明：**（默认为外部链接的变量）**int a = 3;

6. 存储期：存储器描述了标识符访问对象的生存期。存储器一共有四种。

①静态存储期：所有文件作用域的变量都具有静态存储期，对于块作用域变量，可以使用关键字static。这些变量被储存在静态内存中，从程序载入到程序结束期间都存在。（注意：块作用域变量声明用static说明是**静态**的，文件作用域变量声明用static说明是**外部链接**的）

②线程存储期：用于**并发程序**设计。用关键字_Thread_local声明，从被声明到线程结束一直存在③自动存储期：当程序进入定义变量的块时，会为变量分配内存。退出块时，相当于把自动变量占用的内存视为一个**可重复**的工作区。（然而内存的分配还是按顺序）

④动态分配存储区：malloc和calloc分配的内存空间。

Ⅱ. 存储类别

1. 自动变量：

①特点：自动存储期，块作用域，无链接

②相应的存储类型说明符：auto（建议不要使用，因为和C++内涵完全不同）

③是否会初始化：不会，变量的默认值是分配的空间内的任意值；除非显式地初始化。 ④其他特性：内层块的变量会隐藏外层块的定义。（几乎每个语言都是如此，而且我们不建议使用相同的变量名）

2. 寄存器变量：

①特点：自动存储期，块作用域，无链接

②相应的存储类型说明符：register（建议不要使用，编译器会自己优化）

③其他特性：这是一种请求，而不是命令，也就是说，可能会失败；**无法获得寄存器变量的地址**

3. 块作用域的静态变量：

①特点：静态存储期，块作用域，**外部链接**

②相应的存储类型说明符：static。当声明为静态变量时，只在**编译**的时候分配内存，在执行程序的时候不执行声明语句。

4. 外部链接的静态变量：

①特点：静态存储期，文件作用域，外部链接

②相应的存储类型说明符：extern；注意extern不会引起分配存储空间；所以extern不能**定义**外部变量，只能**引用**外部的定义（可以不用，但建议使用，以免弄错）.同理，在当一个文件使用的变量定义在另一个源代码文件中，就要使用extern关键字来引用声明。

③是否会初始化：静态变量都被加载在静态内存中，并且被初始化0；

5. 内部链接的静态变量：

①特点：静态存储期，文件作用域，内部链接

②相应的存储类型说明符：static；

6. 扩展：**私有函数。**

一般我们声明的函数在其他文件中都是允许使用的，为了创建翻译单元特有的函数，我们需要在函数头面前加一个static:

static void f(void) {……}

7. 为什么我们不主张使用文件作用域的变量（这样做的好处就是我们不需要给函数传递数组参数了）？ 因为使用文件作用域变量可能会有副作用，不过我们可以使用const来避免这个问题。

二. 随机函数

Ⅰ. 了解随机函数之前，你需要了解的：

1. 伪随机数列——C中随机数的由来：随机数其实是根据一个公式（所谓的“魔术公式”）得来的，这个公式需要一个初始值（所谓“种子”）,就可以在一定范围内均匀地生成随机数。

2. 时间函数：时间函数time()来自于time.h头文件，它需要提供一个指针（一般给NULL，或者给0让编译器强行转换类型），返回一个时间值（根据时间的变化而变化）。这个时间值恰好可以用来做“种子”。

3. 总结：在C中，使用随机函数一般是使用**stdlib.h**中的**srand()**和**rand()**函数。前者为后者提供一个所谓的“种子”，后者利用种子（一般是time()的返回值）+一个随机数公式生成一个随机数。

Ⅱ. 具体的实现

由上述内容不难得出，想要得到一个随机数列，至少要两个函数（srand()和rand()），这里有一个小问题——这两个函数到底怎么共享同一个参数?

其实是这样的：只要在stdlib.h头文件中包含一个具有内部链接（防止与外面的名称冲突）的静态变量，然后把传入的参数赋给这个变量，就可以供头文件中所有的函数使用。

三. 动态内存分配

（以下的内容均包含在stdlib.h头文件中）

1. malloc():

①作用：提供一个所需的内存字节数，返回分配内存的首个元素的地址（通常是void类型，不过我们可以通过类型转换，提高代码可读性），分配失败返回NULL

②样例：ptd = (double *) malloc(30 * sizeof(double));

③应用：可以用此来创建**动态数组**（在程序运行期间决定数组的大小，就像VLA那样；而一般的数组都是静态加载的）

2. calloc():

①作用：同malloc，不过它接受两个参数：存储单元数量和存储单元大小。

②样例：ptr = (long *) calloc(100, sizeof(long));

3. free():

①作用：给定一个malloc或calloc返回的指针做参数，可以释放之前分配的内存；

②样例：free(ptr);　　//为了不改变ptr的指向，**我们不用对ptr使用自增运算符，而使用数组表示法：如ptr[5]。**

③存在的意义：自动变量会被销毁，而分配的空间依然存在，但是却无法访问。这个问题就是我们常说的”内存泄漏“，及时的free可以避免这个问题。

4. exit():

①作用：配合常量一起使用：如EXIT_FAILURE(代表程序异常终止)，

EXIT_SUCCESS（代表程序正常终止）；

②格式：exit(EXIT_FAILURE);或exit(EXIT_SUCCESS)；

③存在的意义：十分广泛，可以在malloc返回NULL（内存分配失败时）使用exit(EXIT_FAILURE);

5. 再论数组：

①VLA时动态类型变量，离开所定义的块时内存空间会自动释放（不必使用free），另一方面malloc创建的数组不必局限在一个函数内访问。

②其实free不一定要用malloc返回的原来的指针，只要指向的地址相同即可。

③不能用free释放同一块内存两次

④高级技巧：声明二维数组（其实m，n还是要**常量**；不然声明怎么办？）——

p = (int (*)[m]) malloc(n * m * sizeof(int));　　//给指针初始化

6. 再论内存：目前为止，我们已经有三种类别的内存了——

①**静态数据的内存**：在编译时确定，只要程序在运行就可以访问。

②**自动数据的内存**：进入变量定义的块时存在，离开时消失。一般用栈来处理。

③**动态分配的内存**：同样是在程序运行时确定，然而和自动数据还是有点区别，另外其访问较慢　　（看来也是不能摆脱空间换时间的规律啊！）

注意：这三种内存在总的内存中是大相径庭的，因此会有所谓的”内存碎片“（我自己定义的，来源于”磁盘碎片“的定义），所以内存类型应尽量统一。

四. ANSI C限定符

1.const

const的作用就不用多说了，由此看来const确实比define要来的灵活。const有一个很大的作用就是保护全局变量不被改写，这个在多文件作用域中我们主要有两种写法。

①在多个个源文件内：一个源文件定义，另几个源文件引用:

const float PI 3.14159;　　extern const float PI 3.14159;

②使用一个头文件，多个源文件。源文件包含头文件：

static const float PI 3.14159;　　const float PI 3.14159;

(这样做的原理与static关键字息息相关，static关键字表明该文件作用域的变量具有外部链接的属性，这样我们就不必担心要不要加extern来避免命名冲突)

2.volatile

volatile通常被用于硬件地址和在其他程序或同时运行的线程中共享数据。他常常被用于编译器的优化，编译器会把变量的值临时放在寄存器中，等到再次需要使用的时候才从寄存器上读取相应的值，以节约实践。

3.restrict

restrict关键字只能用于**指针**，表明该指针是访问数据对象的**唯一且初始**的方式。也可以进行编译器的优化。

4._Atomic

并发程序设计带来的新的挑战。这包括①如何处理不同的线程访问相同变量的情况。②不同线程所独有的变量怎么考虑。 这些问题可以通过C11的头文件stdatomic.h和threads.h提供的一些管理方法。

_Atomic关键字用来声明一个变量是一个原子变量。当一个现场对一个原子类型的对象执行原子操作时，其他线程不能访问该对象。