一. C库的基本知识

1. 头文件和库文件：头文件为函数提供了原型，库文件包含函数的具体实现。

2. C库的三种访问方法：

①自动访问：在一些系统中，可能会自动访问相应的库文件（不推荐）

②文件包含：用#include自动包含相应的**头文件**。

③库包含：在编译或链接程序的某些阶段，可能需要指定库选项；有些不常用的函数库，必须在编译时显示指定这些库。

3. 使用库描述：（略，这里主要是教你上网查询或者在相关的在线文档或纸质文档学习库文件中函数的用法）

二. 数学库(这里有两个)

1. math.h（原书P546)

1. 在math.h中，所有涉及的角度都是以弧度为单位。

（想要弧度转角度，可以定义一个宏，替换为180/（4*tan（1））)

2. 在数学库中，类型基本上是double类型。

2. 过渡：类型变体

1. 之前有提到math.h中类型都是double，事实上，有了原型之后，也可以把float和long double类型的变量传递给那些函数。不过这样有缺陷：①float处理比double更快②传入long double时会损失精度

2. 利用_Generic()宏根据传入的参数替换。这也有两种写法：

①根据参数的类型确定使用哪个函数

②直接根据参数确定值

（这两个具体实现请见P549，因为知识用来过渡的知识，我就不具体写了）

3. tgmath.h

1. tgmath使用了**泛类型宏**（注意其中的设计思想：用函数写出相应实现，再用类函数宏+_Generic宏确定到底使用那个函数），基本上对于所有原来的math.h中的函数，①加后缀l代表使用long double类型，②加后缀f代表使用float类型。

例如（sqrtl，sqrtf）

2. 如果包含了tgmath.h, 想要调用函数而不是宏，可以用括号将函数名包起来（这一段幕后的原理书上讲得不明不白）。

三. 通用工具库(stdlib.h)

1. exit()和atexit()

1. exit()我就不具体讲了，之前已经讲过了，这里再强调一下可以传入的宏为EXIT_SUCCESS和EXIT_FAILURE。

2. atexit()可以简单地记为(at-exit())，给这个函数传入函数指针，那么在程序结束（**利用exit或程序自动结束**）的时候会自动调用这个函数。

2. qsort()

1. prototype: void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))

2. 各个参数介绍：

①base：void*（空指针类型），指向待排序**数组（不是数组还真不行）**的首元素

②nmemb: 要进行排序的元素个数

③size：每一个元素的大小（要注意之前知识传入了一个万能的空指针，和其他指针不一样，qsort还**不知道每一个元素的大小**，所以要显示使用sizeof！）

④compar：这里传入的是一个返回int的函数指针，其参数都是万能的void*类型，这个函数要自己设置，最好在函数体中用强制类型转换将void*转换为相应类型的指针类型。

3. 多说无益，来看一下一个普通的示例：

```c
/* 输入5个double类型数
 利用qsort 从小到大进行排序 */
#include<stdio.h>
#include<stdlib.h>
#define SIZE 5

// 在声明函数时参数必须要是const void的指针类型！！
int comp(const void *, const void *);

int main(int argc, char *argv[]) {
 double a[SIZE];
 for (int i = 0; i < SIZE; i++) {
 scanf("%lf", &a[i]);
 }
 // 注意，函数名就是函数指针
 qsort(a, SIZE, sizeof(a[0]), comp);
 // qsort在使用以后a数组中的元素被改变了
 for (int i = 0; i < SIZE; i++) {
 printf("%5.2lf", a[i]);
 }
 system("pause>nul");
 return 0;
```

```
}

int comp(const void *a, const void *b) {
 // 这里用到了之前说的强制类型转换
 const double * m = (const double *) a;
 const double * n = (const double *) b;
 // 利用三元运算符嵌套解决返回值的问题
 return (*m > *n)? 1 : (*m == *n)? 0 : -1;
}
```

（另外，如果是比较字符串的话，要引入string.h中的strcmp函数，这个返回值直接return，也比较简单）

四. 断言库（assert.h）

断言库的存在是用于辅助调试程序的，我们在这里一般用两个。

1. assert()

1. 需要的参数：一个常量表达式或逻辑表达式
2. 效果：能在**运行**时终止程序。
3. 用if也能达到相同的效果，但assert有以下两个好处：

①能够自动表示文件和出问题的行号。

②如果不需要assert起作用了，只要在文件开头加一句:#define NDEBUG

2. _Static_assert()

1. 需要的参数：两个，第一个是常量或逻辑表达式，第二个是生成的错误信息
2. 效果：在编译时终止（即让程序无法通过**编译**）
3. 注意：开头加下划线（唯一开头也要加下划线的就是_Generic（）了），首字母大写

总结：我们为什么要使用断言库？很大程度上就是让我们了解程序在哪边出错了，如果在写程序的时候能够在一些地方适当地添加一些必要的逻辑来防止程序出错，那么程序的调试将变得容易。

五. 再提string.h库（memcpy，memmove）

这两个函数存在的意义在于：C中是不能**对数组进行赋值**的，之前所介绍的strcpy和strncpy只能用来处理字符数组，而这里所介绍的两个可以用来进行任意数组的拷贝

1. memcpy

1. prototype: void * memcpy(void * restrict s1, const void * restrict s2, size_t n)
2. 作用：从s2中拷贝n个**字节**到s1中

3. 使用：memcpy(target, curious, 5 * sizeof(double))

2. memmove

1. prototype:void memmove(void * s1, const void * s2, size_t n)
2. 作用：从s2中拷贝n个字节到s1中
3. 使用：memmove(values + 2, values, 5 * (sizeof(int)))

乍一看这两个似乎没有什么区别，但看到restrict我们就明白了：memcpy假设两个内存区域之间没有重叠（如果硬要使用重叠的区域，行为是**未定义**的），memmove假设两个内存区域之间有重叠，所以拷贝的过程类似于先把所有字节拷贝到一个临时缓冲区，然后再拷贝到最终目的地。（也就是说，上面的示例将会覆盖后面数组的内容）

六. 可变参数(stdarg.h)

可变参数极为强大！它允许使用数量可变，类型可变的参数表

使用过程：

1. 先提供一个使用省略号的函数原型：double sum(int n, ...);
(注意这里省略号前一定要有一个代表其参数个数的变量，在这里代表n)
2. 在函数定义中创建一个va_list类型的变量：va_list ap;
3. 用va_start初始化该变量： va_start(ap, n);　　// 现在直到n的必要性了吧，用来动态开辟空间的。
4. 用va_arg逐个访问该列表中的每一项： va_arg(ap, double)； // 注意要指定每一个参数的类型同时要注意这里不会进行类型转换，如果输入了错误的类型会导致程序的失败。
5. 用va_end完成内存释放的工作：va_end(ap);

C99新增了一个用来保存ap副本的函数，这时由于ap无法再次访问已经访问过的元素的情况，这个函数用法如下：va_copy(apcopy, ap);

七. 其他

本章内容其实就是利用文件包含的方式访问库文件。同时，打着介绍库文件的名义访问头文件。还有一点知识我这里再提一下。

1. 内联函数（C99）

1. 背景：通常，函数的调用都会有一定的开销，包括建立调用，传递参数，跳转到函数代码并返回。这里介绍的内联函数，跟之前所讲的类函数宏很像，编译器**根据需要**将其他大函数中这个函数的引用**替换**为你之前函数的定义。
2. 具体的使用：(使用inline关键字)

inline static void eatline(){...}

int main(void) {...eatline()...};

3. 内联函数的效果：相当于在函数调用的位置输入函数体中的代码。

4. 注意事项：

①内联函数应该比较短小，因为长的函数执行的过程远远大于调用的过程，其优化一般。 ②内联函数要求具有内部链接（简单地说就是要在同一个文件中），最简单的方法就是包含同一个头文件。

2. _Noreturn函数（C11）

这时一个函数说明符，表明调用完成后函数**不返回主调函数，**exit就是最好的例子。

这个可以和__attribute__一起使用（详见GNU），不过要注意他不会给系统返回值，所以很可能会炸。