一、C程序的基本组成结构

1、源文件: 后缀为.c 的文件
2、头文件：后缀为.h的文件

注意：

1. 源文件功能：实现程序功能
2. 头文件功能：函数的声明、全局变量的声明、宏定义、类型的声明
3. 一个由C语言所组成的项目中只允许有一个main函数

二、头文件

2.1 头文件的作用

1、宏定义

> #define PI 3.14

2、结构体定义

```
typedef struct student
{
    char name[32];
    int age;
}Stu;
```

3、全局变量的声明

```
extern int cnt; //注意这里是声明，而不是定义
int sum; //这里也是声明
```

4、函数的声明

```
int func(int x, char y);
int func2(int, int); //函数声明时，形参只需要指定数据类型即可
```

示例代码：

头文件：main.h

```
//宏定义
#define PI 3.14
```

```c
//结构体定义
typedef struct student
{
    char name[32];
    int age;
}Stu;

//全局变量的声明
extern int cnt; //注意这里是声明，而不是定义
int sum; //这里也是声明 编译时编译器会自动添加extern

//函数的声明
int func(int x, char y);
int func2(int, int); //函数声明时，形参只需要指定数据类型即可
```

源文件：main.c

```c
#include <stdio.h>

#include "main.h"  //包含自己实现的头文件

//定义全局变量
int cnt;
int sum;

//定义函数
int func(int x, char y)
{
    return 0;
}
//定义函数
int func2(int x, int y)
{
    return 0;
}

int main()
{

    return 0;
}
```

2.2 头文件的编译原理

1、编译器在编译的时候首先进行“预处理”，#include “main.h” 是预处理代码编译器在预处理的时候会将.c 中所包含的头文件中的内容拷贝到.c 文件中

2、为什么#include <stdio.h> 用 <>, 而#include “main.h”用“”呢?

首先编译器在编译的时候，如果为”“ ，编译器会先检索(查找)本工程目录下”“ 内对应的文件，如果该文件存在，继续检索“系统的头文件目录(标准C安装的目录)”，如果是<>，编译器只会去系统的头文件目录下检索<>内对应的文件。

疑问：是不是讲#include stdio.h 写成 #include "stdio.h"编译的时候也能找到stdio.h？是的！！

但是这样会增加编译的时间！！！

2.3 头文件中包含其他的头文件

头文件1：a.h

```
#define PI 3.14

//结构体定义
typedef struct student
{
    char name[32];
    int age;
}Stu;
```

头文件2：b.h

```
#include "a.h" //包含头文件a.h

#define max(a, b) (a)>(b)?(a):(b)
```

2.4 多个头文件的包含问题

1、在上一节中，头文件b.h中包含了头文件a.h， 假如在b.h中我们也定义一个结构体Stu，然后再main.c中包含头文件b.h，将会发生什么情况呢?

头文件1：a.h

```
#define PI 3.14

//结构体定义
typedef struct student
{
    char name[32];
    int age;
}Stu;
```

头文件2：b.h

```c
#include "a.h" //包含头文件a.h

#define max(a, b) (a)>(b)?(a):(b)

//结构体定义
typedef struct student
{
    char name[32];
    int age;
}Stu;
```

源文件：main.c

```c
#include <stdio.h>
#include "b.h"

int main()
{

    return 0;
}
```

对main.c 进行编译后报错：

```
In file included from D:\test\main.c:3:
D:\test\b.h:6:16: error: redefinition of 'struct student'
```

报错显示 struct student这个结构体被重复定义了？这是为什么呢?

因为在编译main.c时，因为main.c中包含了头文件b.h， 就会将b.h中的内容进行展开，而b.h中包含了a.h，所以在b.h中还会将a.h中的内容展开，这样b.h中的内容就变成了：

```c
#define PI 3.14

//结构体定义
typedef struct student
{
    char name[32];
    int age;
}Stu;

#define max(a, b) (a)>(b)?(a):(b)

#define max(a, b) (a)>(b)?(a):(b)

//结构体定义
typedef struct student
{
    char name[32];
```

```
    int age;
}Stu;
```

显然结构体struct student 被重复定义了。

2、如果b.h中没有包含头文件a.h，但是在main.c中同时包含了a.h 和 b.h 也会出现这种编译报错

```
#include "a.h"
#include "b.h"
```

3、解决方法如下：

```
#ifndef A_H
#define A_H

//头文件中的内容

#endif
```

注意：A_H只是一个标识而已，一般使用头文件名字的大写形式，这样所有的头文件中该标识名不会冲突

原理：在头文件中使用这种方法可以避免头文件被重复包含造成的编译报错（大部分时候会报错：类型冲突）
为什么可以做到呢？编译器在编译（预处理阶段）的时候，假如第一次包含该头文件，定义“标识”并且将该头文件中的内容拷贝至源文件中，第二次再包含该头文件的的时候，因为”标识”被定义了该头文件中的内容不会再次被拷贝

三、多个源文件

1、在实际开发过程中有如下需求：

假如A.c 文件文件中实现了函数void func();
如果需要在B.c中调用A.c中的函数func,需要在B.c中加入一条函数的声明语句: extern void func();
当然我们也经常将函数的声明语句放到头文件中

A.c :

```
void func()
{
    printf("hello world\n");
}
```

A.h:

```
#ifndef A_H
#define A_H

extern void func();

#endif
```

B.c:

```
#include "A.h"

extern void func(); //如果包含了A.h 该声明可以省略

void func2()
{
   func(); //调用A.c 中的函数func
}
```

> 练习:
>
> 分别用4个.c 文件实现四个函数
> void info_save(STU *stus, int *num); //信息录入
> void info_check(STU *stus, int num) //信息的查询
> void info_modify(STU *stus, int num) //信息的修改
> void sort(STU *stus, int num)
> // 将stus所指向的空间中的结构体按照学生的分数进行升序排序
>
> 定义一个头文件，在头文件中定义一个结构体

四、static的使用

4.1 static修饰局部变量

1、局部变量的作用域：在定义的{}内

2、局部变量的声明周期：随{}的结束而结束

3、static修饰的局部变量

- 作用域：在定义的{}内
- 声明周期：随进程（程序）的结束而结束

```
int func()
{
    int cnt1 = 0;
    static int cnt2 = 0;
```

```
    cnt1++;
    cnt2++;
    printf("cnt1: %d, cnt2: %d\n", cnt1, cnt2);
}

int main()
{
    int i;
    for (i = 0; i < 4; i++)
    func();
    return 0;
}
```

运行结果：每次调用函数func，cnt1的值都是1，cnt2的值分别为1，2，3，4

原理：

- 普通局部变量在函数定义时被分配在函数栈上，函数栈在函数调用结束后被系统自动回收，所以下一次调用函数时，普通局部变量会被重新定义和赋值！！
- 被static修饰的局部变量定义在静态存储区中，不会随函数的调用结束被系统回收
- 没有初始化的局部变量会被自动初始化为0

4.2 static修饰全局变量

1、普通全局变量的作用域：整个进程（程序），例如在A.c 中定义的全局变量**int cnt**; 在B.c 进行声明 extern int cnt； 就可以被访问到了

2、如果全局变量不想被其他文件访问到（只能在定义的文件内被访问到），可以使用static修饰该全局变量

A.c:

```
static int cnt = 10;
```

B.c:

```
extern int cnt;
void func()
{
    cnt++; //错误
}
```

2、思考：什么时候全局变量需要使用static修饰不被其他源文件访问到呢?

4.3 static修饰函数

1、普通函数的作用域：整个进程（程序），例如在A.c 中定义的函数 void func(int , int);在B.c 进行声明 extern void func(int , int)； 就可以被访问到了

2、如果A.c 中的函数func不想被其他文件访问到（只能在定义的文件内被访问到），可以使用static修饰该函数

A.c:

```
static void func(int x, int y)
{
    printf("A.c: func\n");
}
```

B.c:

```
extern void func(int , int );
void func2()
{
   func(0, 1); //错误的
}
```

原理：

- 因为A.c中的func函数被static修饰了，该函数的作用域就只在A.c文件内了，所以其他文件不能访问到A.c中的func函数
- 其他文件中可以定义函数函数名也可以叫做func，例如在B.c中定义函数 void func(int , char); 那么在B.c中调用的函数就是B.c文件中的，而不是A.c中。

思考：什么时候需要将函数使用static修饰呢?

4.4 static使用总结

1、改变作用域

- static修饰的全局变量和函数，将其作用域由原来的整个进程变成所在的文件中

2、改变生命周期

- static修饰的局部变量，将其生命周期由原来的随{}的结束而结束变成随进程的结束而结束

五、const的使用

1、被const修饰的对象是只读的

2、const常见笔试题

（a）const int * p; 该语句表示指向整形常量 的指针，它指向的值不能修改。
（b）int const * p; 该语句与b的含义相同，表示指向整形常量 的指针，它指向的值不能修改。
（c）int * const p; 该语句表示指向整形的常量指针 ，它不能在指向别的变量，但指向（变量）的值可以修改。
（d）const int *const p; 该语句表示指向整形常量 的常量指针 。它既不能再指向别的常量，指向的值也不能修改。
（e）int const *const p; 该语句与d的含义相同，表示指向整形常量 的常量指针 。它既不能再指向别的常量，指向的值也不能修改。

六、const和define的区别

1、就起作用的阶段而言： #define是在编译的预处理阶段起作用，而const是在 编译、运行的时候起作用。
2、就起作用的方式而言： #define只是简单的文本替换，没有类型检查。而const有对应的数据类型，是要进行判断的，可以避免一些低级的错误（针对C++）。
3、就存储方式而言：#define只是进行展开，有多少地方使用，就替换多少次，它定义的宏常量在内存中有若干个备份；const定义的只读变量在程序运行过程中只有一份备份
4、从代码调试的方便程度而言：const常量可以进行调试的，define是不能进行调试的，因为在预编译阶段就已经替换掉了。