函数的声明

作用： 告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

函数的**声明可以多次**，但是函数的**定义只能有一次**

(在 C 中，如果不自定义函数在 main 方法之后定义，如果 main 方法调用了此函数，则 main 方法找不到此函数，因此需要提前声明)

函数中的值传递

所谓值传递，就是函数调用时实参将数值传入给形参

值传递时，如果形参发生，并不会影响实参(注意：和 Java 中不同)

```
void swap(int num1, int num2)
{
	cout << "交换前： " << endl;
	cout << "num1 = " << num1 << endl;
	cout << "num2 = " << num2 << endl;

	int temp = num1;
	num1 = num2;
	num2 = temp;

	cout << "交换后： " << endl;
	cout << "num1 = " << num1 << endl;
	cout << "num2 = " << num2 << endl;

	//return ; 当函数声明时候，不需要返回值，可以不写 return
}

int main() {

	int a = 10;
	int b = 20;

	swap(a, b);

	cout << "mian 中的  a = " << a << endl;
	cout << "mian 中的  b = " << b << endl;

	system("pause");

	return 0;
}
```

总结： 值传递时，形参是修饰不了实参的

函数的分文件编写

作用： 让代码结构更加清晰

函数分文件编写一般有 4 个步骤

1. 创建后缀名为.h 的头文件
2. 创建后缀名为.cpp 的源文件
3. 在头文件中写函数的声明
4. 在源文件中写函数的定义

示例：

```
#include<iostream>
using namespace std;

//实现两个数字交换的函数声明
void swap(int a, int b);
//swap.cpp 文件
#include "swap.h"

void swap(int a, int b)
{
	int temp = a;
	a = b;
	b = temp;

	cout << "a = " << a << endl;
	cout << "b = " << b << endl;
}
//main 函数文件
#include "swap.h"
int main() {

	int a = 100;
	int b = 200;
	swap(a, b);

	system("pause");

	return 0;
}
```

指针

指针的基本概念

指针的作用： 可以通过指针间接访问内存

内存编号是从 0 开始记录的，一般用十六进制数字表示

可以利用指针变量保存地址

指针变量的定义和使用

指针变量定义语法： 数据类型 * 变量名;

示例：

```
int main() {

	//1、指针的定义
	int a = 10; //定义整型变量 a

	//指针定义语法： 数据类型 * 变量名 ;
	int * p;

	//指针变量赋值
	p = &a; //指针指向变量 a 的地址
	cout << &a << endl; //打印数据 a 的地址
	cout << p << endl;  //打印指针变量 p

	//2、指针的使用
	//通过*操作指针变量指向的内存
	cout << "*p = " << *p << endl;

	system("pause");

	return 0;
}
```

指针变量和普通变量的区别

普通变量存放的是数据,指针变量存放的是地址

指针变量可以通过" * "操作符，操作指针变量指向的内存空间，这个过程称为解引用

总结 1： 我们可以通过 & 符号 获取变量的地址

总结 2：利用指针可以记录地址

总结 3：对指针变量解引用，可以操作指针指向的内存

指针所占内存空间

提问：指针也是种数据类型，那么这种数据类型占用多少内存空间?

示例：

```
int main() {
```

```
    int a = 10;

    int * p;
    p = &a; //指针指向数据 a 的地址

    cout << *p << endl; //* 解引用
    cout << sizeof(p) << endl;
    cout << sizeof(char *) << endl;
    cout << sizeof(float *) << endl;
    cout << sizeof(double *) << endl;

    system("pause");

    return 0;
}
```

总结：所有指针类型在 32 位操作系统下是 4 个字节，64 位操作系统为 8 个字节

空指针和野指针

空指针：指针变量指向内存中编号为 0 的空间

用途： 初始化指针变量

注意： 空指针指向的内存是不可以访问的

空指针

```
int main() {

    //指针变量 p 指向内存地址编号为 0 的空间
    int * p = NULL;

    //访问空指针报错
    //内存编号 0 ~255 为系统占用内存，不允许用户访问
    cout << *p << endl;

    system("pause");

    return 0;
}
```

野指针：指针变量指向非法的内存空间

野指针

```
int main() {

    //指针变量 p 指向内存地址编号为 0x1100 的空间
    int * p = (int *)0x1100;
```

```
    //访问野指针报错
    cout << *p << endl;

    system("pause");

    return 0;
}
```

总结：空指针和野指针都不是我们申请的空间，因此不要访问。

const 修饰指针

const 修饰指针有三种情况

1. const 修饰指针 --- 常量指针
2. const 修饰常量 --- 指针常量
3. const 既修饰指针，又修饰常量

示例：

```
int main() {

    int a = 10;
    int b = 10;

    //const 修饰的是指针，指针指向可以改，指针指向的值不可以更改
    const int * p1 = &a;
    p1 = &b; //正确
    //*p1 = 100;  报错


    //const 修饰的是常量，指针指向不可以改，指针指向的值可以更改
    int * const p2 = &a;
    //p2 = &b; //错误
    *p2 = 100; //正确

    //const 既修饰指针又修饰常量
    const int * const p3 = &a;
    //p3 = &b; //错误
    //*p3 = 100; //错误

    system("pause");

    return 0;
}
```

技巧：看 const 右侧紧跟着的是指针还是常量，是指针就是常量指针，是常量就是指针常量

指针和数组

作用： 利用指针访问数组中元素

示例：

```
int main() {

	int arr[] = { 1,2,3,4,5,6,7,8,9,10 };

	int * p = arr;  //指向数组的指针

	cout << "第一个元素:  " << arr[0] << endl;
	cout << "指针访问第一个元素:  " << *p << endl;

	for (int i = 0; i < 10; i++)
	{
		//利用指针遍历数组
		cout << *p << endl;
		p++;
	}

	system("pause");

	return 0;
}
```

指针和函数

作用： 利用指针作函数参数，可以修改实参的值(和前边形参相反)

示例：

```
//值传递
void swap1(int a ,int b)
{
	int temp = a;
	a = b;
	b = temp;
}
//地址传递
void swap2(int * p1, int *p2)
{
	int temp = *p1;
	*p1 = *p2;
	*p2 = temp;
}

int main() {
```

```
    int a = 10;
    int b = 20;
    swap1(a, b); //  值传递不会改变实参

    swap2(&a, &b); //地址传递会改变实参

    cout << "a = " << a << endl;

    cout << "b = " << b << endl;

    system("pause");

    return 0;
}
```

总结：如果不想修改实参，就用值传递，如果想修改实参，就用地址传递

指针、数组、函数

封装一个函数，利用冒泡排序，实现对整型数组的升序排序

例如数组：int arr[10] = { 4,3,6,9,1,2,10,8,7,5 };

示例：

```
//冒泡排序函数
void bubbleSort(int * arr, int len)   //int * arr  也可以写为 int arr[]
{
    for (int i = 0; i < len - 1; i++)
    {
        for (int j = 0; j < len - 1 - i; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

//打印数组函数
void printArray(int arr[], int len)
{
    for (int i = 0; i < len; i++)
    {
        cout << arr[i] << endl;
    }
```

```
}

int main() {

	int arr[10] = { 4,3,6,9,1,2,10,8,7,5 };
	int len = sizeof(arr) / sizeof(int);

	bubbleSort(arr, len);

	printArray(arr, len);

	system("pause");

	return 0;
}
```

结构体

结构体基本概念

结构体属于用户 自定义的数据类型，允许用户存储不同的数据类型

结构体定义和使用

语法： struct 结构体名 { 结构体成员列表 };

通过结构体创建变量的方式有三种：

struct 结构体名 变量名

struct 结构体名 变量名 = { 成员 1 值 ， 成员 2 值...}

定义结构体时顺便创建变量

示例：

```
//结构体定义
struct student
{
	//成员列表
	string name;  //姓名
	int age;      //年龄
	int score;    //分数
}stu3; //结构体变量创建方式 3


int main() {

	//结构体变量创建方式 1
	struct student stu1; //struct 关键字可以省略

	stu1.name = "张三";
	stu1.age = 18;
	stu1.score = 100;

	cout << "姓名： " << stu1.name << " 年龄： " << stu1.age  << " 分数： " << stu1.score << endl;

	//结构体变量创建方式 2
	struct student stu2 = { "李四",19,60 };

	cout << "姓名： " << stu2.name << " 年龄： " << stu2.age  << " 分数： " << stu2.score << endl;


	stu3.name = "王五";
	stu3.age = 18;
	stu3.score = 80;
```

```
    cout << "姓名：" << stu3.name << " 年龄：" << stu3.age   << " 分数：" << stu3.score << endl;

    system("pause");

    return 0;
}
```

总结 1：定义结构体时的关键字是 struct，不可省略
总结 2：创建结构体变量时，关键字 struct 可以省略
总结 3：结构体变量利用操作符 ''.'' 访问成员

结构体数组

作用： 将自定义的结构体放入到数组中方便维护

语法： struct 结构体名 数组名[元素个数] = { {} , {} , ... {} }

示例：

```
//结构体定义
struct student
{
    //成员列表
    string name;
    int age;
    int score;
}

int main() {

    //结构体数组
    struct student arr[3]=
    {
        {"张三",18,80 },
        {"李四",19,60 },
        {"王五",20,70 }
    };

    for (int i = 0; i < 3; i++)
    {
        cout << "姓名：" << arr[i].name << " 年龄：" << arr[i].age << " 分数：" << arr[i].score << endl;
    }

    system("pause");
```

```
    return 0;
}
```

结构体指针

作用： 通过指针访问结构体中的成员

利用操作符 ->可以通过结构体指针访问结构体属性

示例：

```
//结构体定义
struct student
{
    //成员列表
    string name;   //姓名
    int age;        //年龄
    int score;      //分数
};


int main() {

    struct student stu = { "张三",18,100, };

    struct student * p = &stu;

    p->score = 80; //指针通过 -> 操作符可以访问成员

    cout << "姓名： " << p->name << " 年龄： " << p->age << " 分数： " << p->score << endl;

    system("pause");

    return 0;
}
```

总结：结构体指针可以通过 -> 操作符 来访问结构体中的成员

结构体嵌套结构体

作用： 结构体中的成员可以是另一个结构体

例如： 每个老师辅导一个学员，一个老师的结构体中，记录一个学生的结构体

示例：

```
//学生结构体定义 struct student{    //成员列表   string name;  //姓名   int age;         //年龄    int score;     //分数};//教师结构体定义 struct teacher{     //成员列表     int id; //职工编号      string name;  //教师姓名   int age;    //教师年龄  struct student stu; //子结构体 学生};int main() {    struct teacher t1;  t1.id = 10000;t1.name = "老王"; t1.age    =    40;
    t1.stu.name = "张三";  t1.stu.age = 18;    t1.stu.score = 100;cout << "教师 职工编号：
```

" << t1.id << " 姓名： " << t1.name << " 年龄： " << t1.age << endl; cout << "辅导学员 姓名： " << t1.stu.name << " 年龄:" << t1.stu.age << " 考试分数： " << t1.stu.score << endl; system("pause"); return 0;}

总结： 在结构体中可以定义另一个结构体作为成员，用来解决实际问题

结构体做函数参数

作用： 将结构体作为参数向函数中传递

传递方式有两种:

值传递

地址传递

示例：

```
//学生结构体定义
struct student
{
	//成员列表
	string name;  //姓名
	int age;      //年龄
	int score;    //分数
};

//值传递
void printStudent(student stu )
{
	stu.age = 28;
	cout << "子函数中 姓名： " << stu.name << " 年龄： " << stu.age  << " 分数： " << stu.score << endl;
}

//地址传递
void printStudent2(student *stu)
{
	stu->age = 28;
	cout << "子函数中 姓名： " << stu->name << " 年龄： " << stu->age  << " 分数： " << stu->score << endl;
}

int main() {

	student stu = { "张三",18,100};
	//值传递
	printStudent(stu);
	cout << "主函数中 姓名： " << stu.name << " 年龄： " << stu.age << " 分数： " << stu.score << endl;
```

```
	cout << endl;

	//地址传递
	printStudent2(&stu);
	cout << "主函数中 姓名： " << stu.name << " 年龄： " << stu.age  << " 分数： " << stu.score << endl;

	system("pause");

	return 0;
}
```

总结：如果不想修改主函数中的数据，用值传递，反之用地址传递

结构体中 const 使用场景

作用： 用 const 来防止误操作

示例：

```
//学生结构体定义
struct student
{
	//成员列表
	string name;  //姓名
	int age;      //年龄
	int score;    //分数
};

//const 使用场景
void printStudent(const student *stu) //加 const 防止函数体中的误操作
{
	//stu->age = 100; //操作失败，因为加了 const 修饰
	cout << "姓名： " << stu->name << " 年龄： " << stu->age << " 分数： " << stu->score << endl;

}

int main() {

	student stu = { "张三",18,100 };

	printStudent(&stu);

	system("pause");

	return 0;
```

```
}
```