C++基础入门

7 指针

7.1 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

7.2 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;`

```
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;
    cout << p << endl;

    //2、使用指针
    // 可以通过解引用的方式来找到指针指向的内存
    // 指针前面加 * 代表解引用，知道指针指向的内存中的数据。
    cout << "*p = " << *p << endl;
    retrun 0;

}
```

7.3 指针所占内存空间

- 在32位操作系统下，指针是占4个字节空间大小，不管是什么数据类型
- 再64位操作系统下，指针是站8个字节空间大小

7.4 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访问的 （0 ~ 255 内存编号，系统占用内存，不可以直接访问）

野指针： 指针变量指向非法的内存空间

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

7.5 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

```
int main()
{
    int a = 10;
    int b = 20;

    // 常量指针： const 修饰的是指针，指针指向（即内存地址）可以修改，但是指针指向的值（即内存地址存储的值）不能修改
    // 常量指针：指针指向的值（内存地址所存储的值）是常量，不可修改。
    const int *p1 = &a;
    p1 = &b; //正确，指针p1指向的内存地址 从a的地址变为b的地址
    // p1 = 100; // 错误，指针指向的值被修改了

    //指针常量 ：const 修饰的是常量，指针指向（即内存地址）不能修改，但是指针指向的值（即内存地址存储的值）可以修改
    //指针常量 ：指针指向（即内存地址）是常量，不可修改
    int * const p2 = &a;
    //p2 = &b; //错误，指针p2指向的内存地址被修改
    *p2 = 100; //正确，指针p2指向的值可以被修改

    return 0;
}
```

7.6 指针和数组

作用： 利用指针访问数组中元素

7.7 指针和函数

作用： 利用指针作函数引用，可以修改实参的值

- 值传递：值传递不会修改实参的值，因为会创建临时变量来进行操作，不会对实参造成影响
- 地址传递：地址传递可以修改实参的值，因为会修改指针指向的值（注意指针指向，即内存地址是不变），对实参造成影响

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

void swap1(int a,int b) //值传递
{
```

```
    int temp = a;
    a = b;
    b = temp;
    cout << "swap1中的a=" << a << ", b=" << b << endl;
}
void swap2(int* p1, int* p2) //地址传递
{
    cout << "swap2中交换前的c地址=" << p1 << ", d地址=" << p2 << endl;
    int tmep = *p1;
    *p1 = *p2;
    *p2 = tmep;
    cout << "swap2中的c=" << *p1 << ", d=" << *p2 << endl;
    cout << "swap2中交换前的c地址=" << p1 << ", d地址=" << p2 << endl;
}
int main()
{
    int a = 10;
    int b = 11;
    int c = 20;
    int d = 21;
    cout << "交换前的a=" << a << ", b=" << b << endl;
    swap1(a, b);
    cout << "swap1交换后的a=" << a << ", b=" << b << endl;
    swap2(&c, &d);
    cout << "swap2交换后的c=" << c << ", d=" << d << endl;
    system("pause");
    return 0;
}
```

8 结构体

8.1 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

8.2 结构体定义和使用

语法： `struct 结构体名 {结构体列表};`

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}`
- `定义结构体顺便创建变量`

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

8.3 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] = { {},{},...,{}};`

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

struct student
{
    string  name;
    int age;
    int socre;
};

int main()
{
    student s1[3] =
    {
        {"张三",21,87},
        {"李四",22,34},
        {"王五",23,76}
    };

    s1[2].name = "赵六";
    s1[2].age = 26;
    s1[2].socre = 60;

    for (int i = 0; i < 3; i++) {
        cout << "姓名：" << s1[i].name
             << "年龄：" << s1[i].age
             << "分数：" << s1[i].socre << endl;
    }
    system("pause");
    return 0;
}
```

8.4 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

8.5 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

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

struct student
{
```

```
    string name;
    int age;
    int score;
};
struct teacher
{
    int id;
    string name;
    int age;
    struct student stu;
};


int main()
{
    student s1 = { "张三",18,90 };
    student * p = &s1;
    cout << "姓名：" << p->name
         << " 年龄：" << p->age
         << " 分数：" << p->score << endl;
    teacher t1;
    t1.id = 10001;
    t1.name = "王老师";
    t1.age = 45;
    t1.stu.name = "张三";
    t1.stu.age = 15;
    t1.stu.score = 70;
    cout << "编号：" << t1.id
         << " 姓名：" << t1.name
         << " 年龄：" << t1.age << endl;
    cout << "姓名：" << t1.stu.name
         << " 年龄：" << t1.stu.age
         << " 分数：" << t1.stu.score << endl;
    return 0;
}
```

8.6 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

```
include<iostream>
using namespace std;

struct student
{
    string name;
    int age = 0;
    int score = 0;
};
```

```
1、值传递
void struct_print1(student s)
{
    s.age = 100;
    cout << "(In Sub1 Function)\nname: " << s.name << " ,age: " << s.age << "
,score: " << s.score << endl;
}

2、地址传递
void struct_print2(student *p)
{
    p->age = 200;
    cout << "(In Sub2 Function)\nname: " << p->name << " ,age: " << p->age << "
,score: " << p->score << endl;
}

int main()
{
    student s1;
    s1.name = "zhangsan";
    s1.age = 25;
    s1.score = 98;
    cout << "(In Main Function)\nname: " << s1.name << " ,age: " << s1.age << "
,score: " << s1.score << endl;
    struct_print1(s1);
    cout << "(In Main Function)\nname: " << s1.name << " ,age: " << s1.age << "
,score: " << s1.score << endl;
    struct_print2(&s1);
    cout << "(In Main Function)\nname: " << s1.name << " ,age: " << s1.age << "
,score: " << s1.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

8.7 结构体中const使用场景

作用： 用const防止误操作

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

struct student
{
    string name;
    int age = 0;
    int score = 0;

};

void struct_print( const student *s)
{
    //s->age = 150;
```

```
    cout << "(In Sub Function)\nname: " << s->name << " ,age: " << s->age << "
,score: " << s->score << endl;
}

int main()
{
    struct student s1 = { "zhangsan",15,70 };
    struct_print(&s1);
    cout << " ,age: " << s1.age << endl;
    system("pause");
    return 0;
}
```