

一、指针

1. 什么是指针

指针其实就是一个变量，不过它的值是一个内存地址，这个地址可以是变量或者一个函数的地址

当你声明一个变量的时候，计算机会将指定的一块内存空间和变量名进行绑定；这个定义很简单，但其实很抽象，例如：`int x = 5`；这是一句最简单的变量赋值语句了，我们常说“x等于5”，其实这种说法是错误的，x仅仅是变量的一个名字而已，它本身不等于任何值的。这条statement的正确翻译应该是：“将5赋值于名字叫做x的内存空间”，其本质是将值5赋值到一块内存空间，而这个内存空间名叫做x。切记：x只是简单的一个别名而已，x不等于任何值。

`int x = 5`



0x0012

• 为什么需要指针？

不就是使用变量或者调用函数吗？难道不能直接调用吗？那么需要指针做什么呢？

实际上是可以的，但是并不是所有的情况都可以。比如：

1. 在内部函数中，可以使用指针访问外部函数中定义的某个变量x，因为它并不是声明在自己的函数范围内。
2. 指针在处理函数传递数组的时候非常高效，传递的是数组的首元素地址。
3. 我们还可以在堆内存中申请一块区域，这块区域甚至没有一个变量名称，唯一的访问方式是通过指针。
4. 可以使用你指针访问指定的内存地址（游戏修改器） 内存修改器 | 金山游侠 | CE ... xxx内存修改器

2. 声明和初始化

1. 声明指针

1.声明指针的时候要记得初始化，如果没有初始化，指针存放的将会是垃圾数据(因为根本不知道它指向何方)。如果暂时不知道指针指向何方，则可以把这个指针设置为空指针。

```
#include <iostream>
int main(){
    //初始化指针
    int a = 3 ;
    int *p0 = &a; //p0是一个指针，指向的是一个int类型的数据，这个数据是3.

    //空指针 ， 没有指向任何地方
    int *p1 = nullptr;
    int *p2 = NULL;
    int *p3 = 0 ;

    return 0 ;
}
```

2. 指针地址和大小

指针实际上也是一个变量，也会有自己的内存空间，也会有自己的长度大小。获取指针的内存地址，依然使用取地址符 `&`， 长度大小依然使用 `sizeof` 来获取

```
#include <iostream>

using namespace std;

int main(){
    int age = 88;
    int *p = &age;

    cout << "指针的地址是: " << &p <<endl;
    cout << "指针存储的是: " << p <<endl;
    cout << "指针大小是: " << sizeof(p) <<endl;
    cout << "age的大小是: " << sizeof (age) <<endl;
    return 0 ;
}
```

```
}
```

3. 两种创建指针的方式

指针接受的值只有地址，根据内存分区的划分，指针可以接收**栈内存**和**堆内存**的地址。这也就形成了指针的创建有两种方式。

- 栈内存

栈内存的空间较小，一般在这个区域存放局部变量 | 函数的参数这些数据。当函数执行结束，栈内存会自动回收这些变量所占用的空间

```
#include <iostream>

int main(){
    int age = 88;
    int *p = &age; //函数执行结束， age所占用空间会自动回收
    return 0 ;
}
```

- 堆内存

堆内存的空间相对较大，一般这个区域需要程序员采用 **new** 关键字来开辟空间，并且空间的回收也是由程序员使用 **delete** 关键字来操作

```
#include <iostream>

int main(){

    //使用new的方式在堆内存中开辟空间。
    int *p = new int(88);

    //需要手动配合delete 删除
    delete p ;
    return 0 ;
}
```

3. 指针dereference（解引用）

所谓的指针dereference就是，指针就是一个变量，存放的是一个地址。这个地址有可能是变量 a 或者是变量b的地址。有了这个地址，我们可以通过**dereference**操作符 `*` 去获取到a对应的值或者b对应的值。

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

int main(){

    //定义一个变量score，赋值100
    int score {100};

    //定义一个指针score_ptr 指向score的地址。
    int *score_ptr{&score};

    //通过指针，获取到指向位置的数据 打印100
    cout << *score_ptr << endl;

    //使用指针修改原来的score
    *score_ptr = 200 ;

    //使用指针和变量的方式打印score，结果都输出200
    cout << *score_ptr << endl;
    cout << score << endl;
    return 0 ;
}
```

二、指针与函数

1. 参数传递指针

函数的参数，除了传递普通的变量，引用之外，还可以把指针当成参数来传递

```
#include <iostream>
```

```

using namespace std;

//函数原型
void double_data(int *int_ptr);

int main(){
    int value{10};

    //修改前，输出为10
    cout << value << endl;

    //在函数内部修改value的值
    double_data(&value);

    //修改后，输出为20
    cout << value << endl;

}

void double_data(int *int_ptr){
    *int_ptr *=2;
}

```

在某些情况下，传递指针比其他方式的传递要合适得多，比如下面有一个函数负责交换传递进来的两个参数的值，此时如果不使用指针或者引用，则无法实现该功能

```

#include <iostream>
using namespace std;

void swap(int *a , int *b);

int main (){

    int x{100},y{200};

    //交换前打印 x : 100 , y : 200
    cout << x <<" = " << y <<endl;

    swap(&x , &y);
}

```

```

//交换前打印 x : 200 , y : 100
cout << x <<" = " << y <<endl;

}

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

```

2. 函数返回指针

函数平常除了返回标准的数值之外，其实也可以返回指针。

```

#include <iostream>
using namespace std;

//返回两个参数中的最大者
int *calc_largest(int *ptr1 , int *ptr2);

int main(){
    int a = 100;
    int b = 200;

    //使用指针指向最大数值
    int *largest_ptr = calc_largest(&a , &b);

    //输出: 200
    cout << *largest_ptr << endl ;
    return 0 ;
}

int *calc_largest(int *ptr1 , int *ptr2){
    //解引用获取到数据后比较 :
    if(*ptr1 > *ptr2){
        return ptr1;
    }else{
        return ptr2;
    }
}

```

注意：不要返回一个函数内部的一个局部变量指针，因为本地变量的生命周期应该只位于函数内部。一旦函数执行完毕则被释放。

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

int *do_this(){
    int size = 10;
    return &size;
}

int main(){

    int *result = do_this();

    std::cout <<"result = " <<result << std::endl;

    return 0 ;
}
```

三、引用

1. 什么是引用

引用，顾名思义是某一个变量或对象的**别名**，对引用的操作与对其所绑定的变量或对象的操作完全等价。引用在使用时，有几个要注意的地方：

1. &不是求地址运算符，而是起到标志作用
2. 引用的类型和绑定的变量类型必须相同 (指针也一样)
3. 声明引用的同时，必须对其进行初始化，否则报错。
4. 引用不是定义一个新的变量或对象，因此**内存不会为引用开辟新的空间存储这个引用**
5. 不能建立数组的引用。因为数组是一个由若干个元素所组成的集合，所以无法建立一个数组的别名。

```
#include<iostream>
#include<vector>
```

```
using namespace std;

int main(){
    //语法：类型 &引用名=目标变量名；
    int a = 3 ;
    int &b = a ;

    //此时a也会变成33
    b = 33;

    vector<string> names {"张三" , "李四" ,"王五"};

    // 这仅仅是改变了str的值，并不会改变vector里面的元素
    for(auto str:names){
        str = "赵六";
    }

    //此处不会产生新的变量，所以修改的都是vector里面的元素
    for(auto &str : names){
        str = "赵六" ;
    }
    return 0 ;
}
```

2. 左值和右值

C++的表达式要么是左值 `lvalue`，要么是右值 `rvalue` 这两个名词是从C语言继承过来的。**左值可以出现在赋值语句（=）的左侧和右侧，右值只能出现在右侧。**最常见到左值和右值的地方，是在函数的参数以及报错的日志信息里面。

不能简单的以等号的左右来判断是否是左值还是右值

判断是否是左值，有一个简单的办法，就是**看看能否取它的地址**，能取地址的就是左值。使用排除法，其他的即为右值。

左值一般就是对象、变量，右值一般是普通的数据（实实在在的数据）、运算表达式、方法的返回值。


```
//只能把左值赋值给引用，不能把右值赋给引用
int square(int &n){
    return n * n ;
}

int main(){
    int x = 1000; // x是一个左值， 而 1000 是一个右值。
    x = 1000 + 20 ; //x 是左值， 1000 + 20 是右值，

    int b = 10; // b 是一个左值， 10 是一个右值
    b = x ; //b是一个左值， 而 x依然是一个左值。

    int num = 10 ;
    square(num) ; //正确

    square(5) //错误。因为5是右值 ， 不能赋值引用。
    return 0;
}
```

3. 左值引用

平常所说的引用，实际上指的就是左值引用 `lvalue reference`，常用单个 `&` 来表示。左值引用只能接收左值，不能接收右值。**const** 关键字会让左值引用变得不同，它可以接收右值

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

//函数原型
int add(int &num1);
void print(vector<int> &scores);

int main(){

    int a = 3 ;
    int &b = a ; //ar是一个左引用，实际上可以看成是a的一个别名。

    // 这是不允许的。
```

```
// 1. 从引用层面理解的话是： 引用接收的一个变量，给某个变量起别名
// 2. 从左右值的层面理解是，这是一个左值引用，只能接收左值。 3
属于右值。
int &c = 3 ; //错误!

int a2 = 3 ;
add(a2) ; //正确
add(3) ; //错误! 参数要求的是一个左值引用，只能赋值左值 ,3 属于右值

vector<int> scores{60,70,80,90};
print(scores); //正确
print({60,70,80,90}); //错误!
}
```

4. 右值引用

为了支持移动操作，在c++11版本，增加了右值引用。右值引用一般用于绑定到一个即将销毁的对象，所以右值引用又通常出现在移动构造函数中。

看完下面的例子，左值和右值基本就清楚了，左值具有持久的状态，有独立的内存空间，右值要么是字面常量，要么就是表达式求值过程中创建的临时对象

```

int main(){

    int i = 66;
    int &r = i ; //r 是一个左引用，绑定左值 i

    int &&rr = i ; //rr是一个右引用，绑定到左值i ， 错误！
    int &r2 = i*42 ; // r2 是一个左引用， 而i*42是一个表达式，
    计算出来的结果是一个右值。 错误！

    const int &r3 = i*42; // 可以将const的引用，绑定到右值 正确
    int &&rr2 = i*42 ; // 右引用，绑定右值 正确

    return 0 ;
}

```

- 示例

```

#include<vector>
using namespace std;

//函数原型
int add(int &&num1);
void print(vector<int> &&scores);

int main(){
    int a = 3 ;
    add(a) ; //错误！ 参数要求的是一个右值引用， 只能赋值右值
    add(3) ; //正确！

    vector<int> scores{60,70,80,90};

    //print接收的是一个右值， 此处的scores是一个左值。
    print(scores); //错误！

    //{60,70,80,90} 属于右值。
    print({60,70,80,90}); //正确

    return 0 ;
}

```

