引用

范 懿

目录

引用的动因

指针的优点和不足

1. 指针在编程中发挥很大作用；
2. 指针的使用比较繁琐；

☞ 如何发挥指针的优势，克服指针的不足？

引用的三种应用场景

1. 作为函数的传入值
2. 作为函数的返回值
3. 独立使用

函数参数的传入

C 语言参数传入——一切转化为常量（数组名转为首元素地址）。

☞ 在函数体中做出了一个副本。

C++ 语言参数传入的法则

- 带“引用”时，不转化为常量
- 不带“引用”时，转化为常量——与 C 语言一致

以下将着重讨论 C++ 语言中，**带“引用”时**的参数传入法则。

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

void neg(int *i_ptr) // 传入指针值可修改外部变量
/*把i_ptr所指的整数的符号翻转*/
{
    *i_ptr = -(*i_ptr);
}

int main()
{
    int x;
    x = 10;
    cout << x << " negated is ";
    neg(&x); // 地址即指针
    cout << x << "\n";
    return 0;
}

/*函数neg()可以翻转一个整数的符号，但是代码不够直
    观。*/
```

引用的概念

传入“引用”，需要在变量前加 & 号

注意

1. 被引用的必须是一片内存。
2. 被引用的必须是变量，因此必须是左值（lvalue）。
3. 引用变量必须在声明的时候就指定**被引用者**。

定义

左值就是可以放在赋值符左边。

例

1. 变量 `n`，数组的分量 `a[2]`，指针所指的 `(*p)` 都是左值。
2. `1`, `"China"`, `0028FF1C` 不是左值。

```cpp
#include <iostream>
using namespace std;
int main(){
    int i = 10;
    int &refToI = i; // 声明i的引用
    cout << "i: " << i << endl; //取值始终相同
    cout << "refToI: " << refToI << endl;
    i++; // 一个改变另一个也跟着变
    cout << "i: " << i << endl;
    cout << "refToI: " << refToI << endl;
    refToI++; // 一个改变另一个也跟着变
    cout << "i: " << i << endl;
    cout << "refToI: " << refToI << endl;
}
/*
输出示例:
i: 10
refToI: 10
```

```
19 i: 11
20 refToI: 11
21 i: 12
22 refToI: 12
23 */
```

引用的内存本质

引用

——**指针**的简易写法，是一种可读性强、不易犯错的指针写法。

变量的访问

1. 直接访问
2. 间接访问

☞ 引用是间接访问的简易写法。

例

引用相当于联名账号，提供了多条访问途径。

声明引用

例

```
int a;  // 为变量 a 分配一片内存空间，通常为 4 个字节
int &refToA = a; // 编译器转为int const* refToA = &a;
```

例

如图，变量 a 占据 4 个字节，地址为 0028FE10。

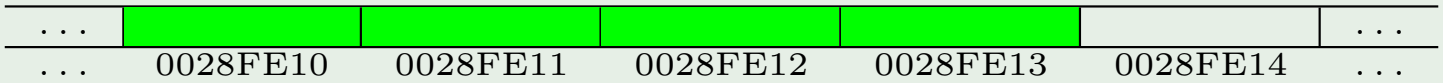

表: 变量 a 占据的内存

随后变为

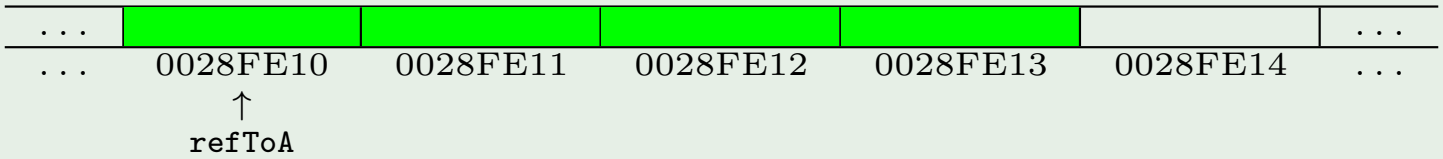

表: refToA 指向 a

声明引用

例

```
float s; // 为变量 s 分配一片内存空间，通常为 4 个字节
float &refToS = s; // 转为float const* refToS=&s;
```

例

如图，变量 s 占据 4 个字节，地址为 0028FD10。

...						...
...	0028FD10	0028FD11	0028FD12	0028FD13	0028FD14	...

表: 变量 s 占据的内存

随后变为

...						...
...	0028FD10	0028FD11	0028FD12	0028FD13	0028FD14	...
	↑ refToS					

表: refToS 指向 s

声明引用

例

```
char ch;  // 为变量 ch 分配一片内存空间，通常 1 个字节
char &refToCh=ch;  // 转为char const* refToCh=&ch;
```

例

如图，变量 ch 占据 1 个字节，地址为 0028FC12。

表: 变量 ch 占据的内存

随后变为

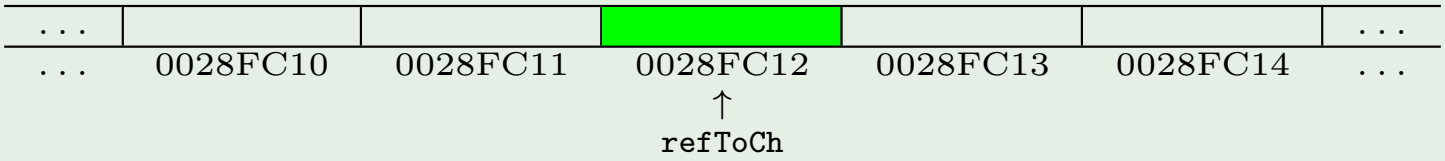

表: refToCh 指向 ch

引用的运算

例

```
int a = 5;
int &refToA = a; // 等价于int const* refToA = &a;
refToA = 8;  // 编译器自动转为 *refToA = 8;
```

例

```
float s = 1.5;
float &refToS = s; //等价于float const* refToA = &s;
refToS = 1.75; // 编译器自动转为 *refToS = 1.75;
```

例

```
char ch = 'a';
char &refToCh = ch;  //等价于char const* refToCh=&ch;
refToCh = 'b'; // 编译器自动转为 *refToCh = 'b';
```

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

class TestReferences{
public:
    int l;
    int &i;  // 编译器转换成int * i;
    int &j;  // 编译器转换成int * j;
    int &k;  // 编译器转换成int * k;
};
int main()
{
    cout<<"sizeof(int const*)="<<sizeof(int const*)<<endl;
    // 将打印32, 即3个指针和1个整型所占空间
    cout<<"size of TestReferences="<<sizeof(class TestReferences
    ) << endl;
    TestReferences testRef;
    int a = 2, b = 3, c = 4;
    //编译器转换成*testRef.i = a;*testRef.j = b;*testRef.k = c;
    testRef.i = a; testRef.j = b; testRef.k = c;
    //同上
    cout << "testRef.i, testRef.j, testRef.k: " << testRef.i <<
    ", " << testRef.j << ", " << testRef.k << endl;
    cout << "a, b, c: " << a << ", " << b << ", " << c << endl;
    return 0;
}
```

引用传入函数

函数定义

```
void neg(int &x)  // 编译器自动转为 void neg(int const* x)
{
   x = -x;  // 编译器自动转为 (*x) = -(*x);
}
```

函数调用

```
neg(n); // 编译器自动转为 neg(&n);
```

☞利用指针在函数 neg() 内部做出了 n 的引用 x——联名账号。

```
void plus_plus(int &x)
{
  x++;  // ☞这不是指针移动，因为编译器自动转为 (*x)++;
}
```

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

/*把i_ref所引的整数的符号翻转*/

void neg(int &i_ref)
// 自动翻译为void neg(int const* i_ref)
{
    i_ref = -i_ref; // 自动翻译为(*i_ref)=-(*i_ref);
}

int main()
{
    int x;
    x = 10;
    cout << x << " negated is ";
    neg(x); // 自动翻译为neg(&x);
    cout << x << "\n";
    return 0;
}

/*函数neg()可以翻转一个整数的符号，代码更直观。*/
```

```cpp
#include <iostream>
using namespace std;
void swap(int &i, int &j)
// 自动转为 void swap(int const* i, int const* j)
{
    int t;
    t = i; // 自动转为t = (*i);
    i = j; // 自动转为(*i) = (*j);
    j = t; // 自动转为(*j) = t;
}

int main()
{
    int a, b, c, d;
    a = 1; b = 2; c = 3; d = 4;
    cout << "a and b: " << a << " " << b << "\n";
    swap(a, b); // 自动转为swap(&a, &b);
    cout << "a and b: " << a << " " << b << "\n";
    cout << "c and d: " << c << " " << d << "\n";
    swap(c, d); // 自动转为swap(&c, &d);
    cout << "c and d: " << c << " " << d << "\n";
    return 0;
}
```

返回引用

☞这使得函数调用可以发生在赋值运算符的左边。

函数的定义

```
int& item(int *array, int k)
// 编译器转为 int *item(int *array, int k)
{
  return array[k];  // 编译器转为 return &array[k];
}
```

函数的调用

```
item(a, 3) = 8;  // 自动转为 *item(a, 3) = 8;
```

☞指针在函数 `item()` 外做出了 `array[k]` 的引用 `item(array, k)`——联名账号。

函数内部声明的变量，它们的引用不能被返回，因为生存域受限。

```cpp
#include <iostream>

using namespace std;

char &replace(int i);
// 返回引用，自动转为char* replace(int i);

char s[80] = "Hello There"; // 全局变量，放在static空间

int main()
{
    // 把X赋给Hello之后的字符空间
    replace(5) = 'X'; //编译器自动转为*replace(5) = 'X';
                      //replace(5)做成了s[5]的引用
    cout << s;
    return 0;
}

char &replace(int i)
// 自动转为char* replace(int i)
{
    return s[i]; // 编译器自动转为return *s[i];
}
```

引用的补充事项

注意以下提到的非法操作

- 不能引用另一个引用变量
- 不能获取引用变量的地址
- 不能构造引用变量组成的数组
- 不能构造指向引用变量的指针
- 不能引用一个比特域（bit-field）

引用必须被初始化，除非它是

- 类的成员
- 函数的参数
- 或者返回值

引用和指针的书写风格

以下两种写法等价

- `int* p;`
- `int *p;`

以下两种写法等价

- `int& r;`
- `int &r;`

某些 C++ 程序员习惯把 * 和 & 与类型名写在一起。

优缺点

- * 和 & 与类型名一起，有利于把指针和引用看做特定的类型
- 但是可能会引起误导

例

```
int *p, q; // 程序员容易误以为 p 和 q 都是指针类型
```

用引用来隐式返回两个或以上的值

例

编写一个函数，使得输入一个正整数，并判断它的平方能否写成两个正整数的平方和。如果能，返回真并给出一种可能，否则返回假。

既要返回真假值，又要给出分解方案，单用`return`语句不够。

☞ 在传入值中使用引用即可

```cpp
#include <stdio.h>
/*一个函数返回多个值*/

/*以下自动转为int divide(int c, int *a, int *b)*/
int divide(int c, int &a, int &b)
/*把一个正整数c进行分解，写成c^2=a^2+b^2，
若成功，则返回真，并给出a和b的值，否则返回假*/
{
    int x, y;
    for(x = 1; x * x <= c * c; x++)
    {
        for(y = 1; x * x + y * y <= c * c; y++)
        {
            if(x * x + y * y == c * c){
                /*以下自动转为*a=x,*b=y*/
                a = x; // 把值保存在外面
                b = y; // 函数结束后，x和y将丢失
                return 1; // 分解成功
            }
        }
    }
    return 0; // 分解失败
}
```

```
24
25 int main()
26 {
27     int n;
28     printf("Please input an integer:\n");
29     scanf("%d", &n); // 输入要分解的值
30     int a, b;
31     /*以下自动转为if(divide(n, *a, *b))*/
32     if(divide(n, a, b)) // 传入引用以便进行修改
33     {
34         printf("%d^2 = %d^2 + %d^2\n", n, a, b);
35     }
36     else
37     {
38         printf("cannot be divided\n");
39     }
40     return 0;
41 }
```