第八章 程序设计复合类型-指针

模块8.3：指针与引用

主讲教师：同济大学电子与信息工程学院 陈宇飞
同济大学电子与信息工程学院 龚晓亮

目录

- 引用变量
- 引用作函数参数
- 引用的属性和特别之处
- 何时使用引用参数

目录

- 引用变量
  - 引用类型的本质
  - 创建引用变量
  - 引用变量的作用

1.1 引用类型的本质

✓引用变量是C++新增的一种复合类型

✓引用就是某一变量（目标）的一个别名，对引用的操作与对变量直接操作完全一样

✓引用的声明方法：类型标识符 &引用名=目标变量名

```
int a; int &ra = a; //定义引用ra,它是变量a的引用，即别名
```

```
// firstref.cpp -- defining and using a reference
#include <iostream>
int main()
{
    using namespace std;
    int rats = 101;
    int& rodents = rats;   // rodents is a reference

    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;
    rodents++;
    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;

    // some implementations require type casting the following
    // addresses to type unsigned
    cout << "rats address = " << &rats<< endl;
    cout << "rodents address = " << &rodents << endl;

    return 0;
}
```

```
rats = 101, rodents = 101
rats = 102, rodents = 102
rats address = 00000007D24FF5B4
rodents address = 00000007D24FF5B4
```

```
// secref.cpp -- defining and using a reference
#include <iostream>
int main()
{
    using namespace std;
    int rats = 101;
    int& rodents = rats;   // rodents is a reference

    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;
    cout << "rats address = " << &rats;
    cout << ", rodents address = " << &rodents << endl;

    int bunnies = 50;
    rodents = bunnies;       // can we change the reference?
    cout << "bunnies = " << bunnies;
    cout << ", rats = " << rats;
    cout << ", rodents = " << rodents << endl;
    cout << "bunnies address = " << &bunnies;
    cout << ", rodents address = " << &rodents << endl;
    return 0;
}
```

```
rats = 101, rodents = 101
rats address = 0000007AA1AFFA24, rodents address = 0000007AA1AFFA24
bunnies = 50, rats = 50, rodents = 50
bunnies address = 0000007AA1AFFA64, rodents address = 0000007AA1AFFA24
```

1.2 创建引用变量

✓ 引用的声明方法：类型标识符 &引用名=目标变量名

```
int a; int &ra = a; //定义引用ra,它是变量a的引用，即别名
```

（1）&在此不是求地址运算，而是起标识作用

（2）类型标识符是指目标变量的类型

（3）声明引用时，必须同时对其进行初始化

1.2 创建引用变量

（4）引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，且不能再把该引用名作为其他变量名的别名

```
int a; int &ra = a;

ra = 1; // 等价于 a=1;
```

（5）声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，因此引用本身不占存储单元，系统也不给引用分配存储单元

故：对引用求地址，就是对目标变量求地址。&ra与&a相等

1.2 创建引用变量

（6）不能声明引用数组，但可声明数组的引用、数组元素的引用

```
int &b[3];                  //错误，不能声明引用数组

int a[5],(&b)[5]=a;//正确，引用指向整个数组

int a[5],&b=a[3];  //正确，引用指向数组元素
```

1.2 创建引用变量

（7）不能建立引用的引用，不能建立指向引用的指针

```
int n;     int &&r = n; //错误     int &*p = n; //错误
```

（8）可以建立指针的引用

```
int *p;    int *&q = p; //正确
```

编译系统把”int *“看成一体，把"&q"看成一体，即建立指针p的引用，亦即给指针p起别名q

1.3 引用变量的作用

✓引用的特点：就是在函数调用时在内存中不会生成副本

（1）在引用的使用中，单纯给某个变量取个别名是毫无意义的，引用的目的主要用于在函数参数传递中，解决大块数据或对象的传递效率和空间不如意的问题

（2）用引用传递函数的参数，能保证参数传递中不产生副本，提高传递的效率，且通过const的使用，保证了引用传递的安全性

1.3 引用变量的作用

✓ 引用的特点：就是在函数调用时在内存中不会生成副本

（3）引用与指针的区别是，指针通过某个指针变量指向一个对象后，对它所指向的变量间接操作。程序中使用指针，程序的可读性差；而引用本身就是目标变量的别名，对引用的操作就是对目标变量的操作

（4）使用引用的时机：流操作符<<和>>、赋值操作符=的返回值、拷贝构造函数的参数、赋值操作符=的参数、其它情况都推荐使用引用 （后续知识，此处了解即可）

目录

- 引用作函数参数

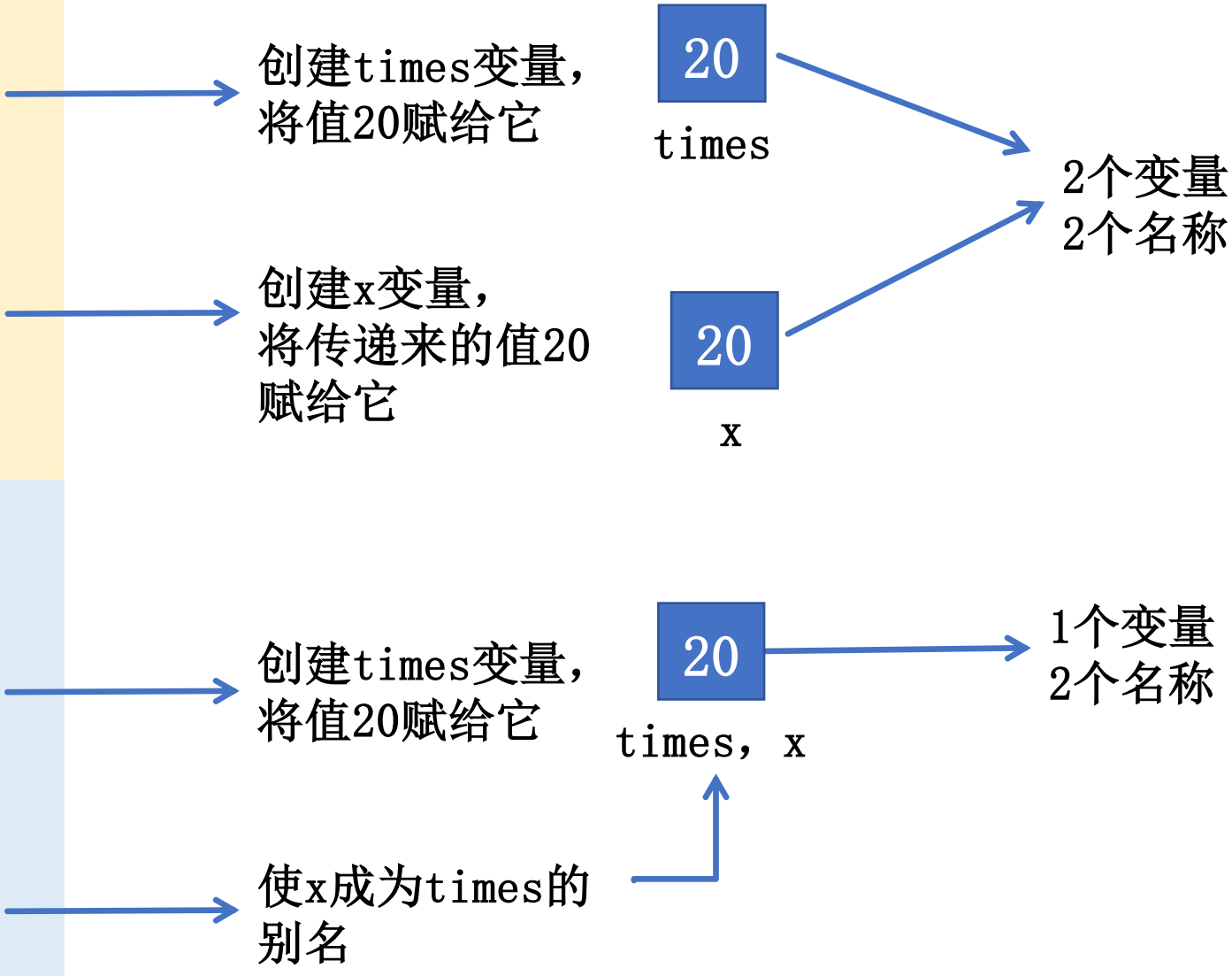
2.1 引用作函数参数

✓函数参数传递是单向传值，如果有大块数据作为参数传递的时候，采用的方案往往是指针，因为这样可以避免将整块数据全部压栈，可以提高程序的效率

✓C++中增加了一种同样有效率的选择（在某些特殊情况下又是必须的选择），就是引用

```
void sneezy(int x);
int main ()
 {
    int time = 20;
    sneezy (times) ;
    ...
}
void sneezy (int x)
 {
    ...
}
```

```
void grumpy(int &x);
int main ()
 {
   int time = 20;
   grumpy (times) ;
   ...
}
void grumpy (int &x)
 {
   ...
}
```



```
// swaps.cpp -- swapping with references and with pointers
#include <iostream>
void swapr(int& a, int& b);   // a, b are aliases for ints
void swapp(int* p, int* q);   // p, q are addresses of ints
void swapv(int a, int b);     // a, b are new variables
int main()
{   using namespace std;
    int wallet1 = 300;
    int wallet2 = 350;
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Using references to swap contents:\n";
    swapr(wallet1, wallet2);      // pass variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Using pointers to swap contents again:\n";
    swapp(&wallet1, &wallet2);    // pass addresses of variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Trying to use passing by value:\n";
    swapv(wallet1, wallet2);      // pass values of variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;
    return 0;
}
```

Part1


```
wallet1 = $300 wallet2 = $350
```

```
Using references to swap contents:
wallet1 = $350 wallet2 = $300
```

```
Using pointers to swap contents again:
wallet1 = $300 wallet2 = $350
```

```
Trying to use passing by value:
wallet1 = $300 wallet2 = $350
```

Part2

```
void swapr(int& a, int& b)    // use references
{
    int temp;

    temp = a;        // use a, b for values of variables
    a = b;
    b = temp;
}
void swapp(int* p, int* q)    // use pointers
{
    int temp;

    temp = *p;       // use *p, *q for values of variables
    *p = *q;
    *q = temp;
}
void swapv(int a, int b)        // try using values
{
    int temp;

    temp = a;       // use a, b for values of variables
    a = b;
    b = temp;
}
```

```
Using references to swap contents:
wallet1 = $350 wallet2 = $300
```

```
Using pointers to swap contents again:
wallet1 = $300 wallet2 = $350
```

```
Trying to use passing by value:
wallet1 = $300 wallet2 = $350
```

2.1 引用作函数参数

✓声明函数参数的方式不同

```
void swapr(int& a, int& b);    // a, b are aliases for ints（传递引用）
void swapp(int* p, int* q);    // p, q are addresses of ints（传递指针）
void swapv(int a, int b);      // a, b are new variables
```

✓函数调用的区别

```
swapr(wallet1, wallet2);       // pass variables（按引用传递）
swapp(&wallet1, &wallet2);     // pass addresses of variables（按地址传递）
swapv(wallet1, wallet2);      // pass values of variables（按值传递）
```

✓指针版本需要在函数使用p和q的整个过程中使用解除引用运算符*

```
void swapp(int* p, int* q)     // use pointers
{   int temp;
    temp = *p;       // use *p, *q for values of variables
    *p = *q;
    *q = temp;
}
```

2.1 引用作函数参数

（1）传递引用给函数与传递指针的效果是一样的

（2）使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好

（3）使用指针作为函数的参数在被调函数中同样要给形参分配存储单元，需要重复使用"*指针变量名"的形式进行运算，容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参

引用更容易使用，更清晰

目录

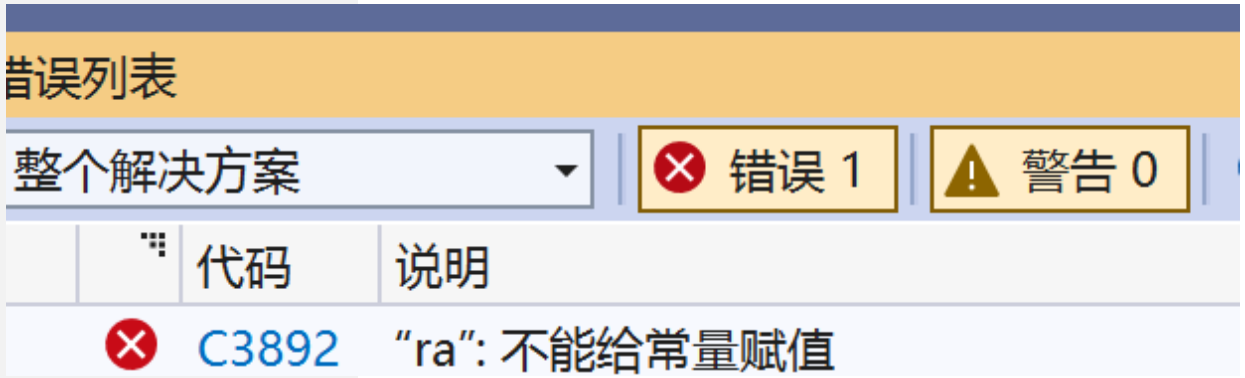
- 引用的属性和特别之处
  - 引用的属性
  - 引用的特别之处

```
// cubes.cpp -- regular and reference arguments
#include <iostream>
double cube(double a);
double refcube(double& ra);
int main()
{
    using namespace std;
    double x = 3.0;

    cout << cube(x);
    cout << " = cube of " << x << endl;
    cout << refcube(x);
    cout << " = cube of " << x << endl;
    return 0;
}
double cube(double a)
{
    a *= a * a;
    return a;
}
double refcube(double& ra)
{
    ra *= ra * ra;
    return ra;
}
```

```
27 = cube of 3
27 = cube of 27
```

```
// cubes.cpp -- regular and reference arguments
#include <iostream>
double cube(double a);
double refcube(const double& ra);
int main()
{
    using namespace std;
    double x = 3.0;

    cout << cube(x);
    cout << " = cube of " << x << endl;
    cout << refcube(x);
    cout << " = cube of " << x << endl;
    return 0;
}
double cube(double a)
{
    a *= a * a;
    return a;
}
double refcube(const double& ra)
{
    ra *= ra * ra;
    return ra;
}
```

如果要编写使用基本数值类型的函数，应采用按值传递的方式，而不要采用按引用传递的方式



3.1 引用的属性

✓按值传递的函数，可使用多种类型的实参

```
double z = cube(x + 2.0); // x + 2.0 is an rvalue,pass value
z = cube(8.0);            // 8.0 is an rvalue, pass value
int k = 10;
z = cube(k);      // k is an lvalue, convert value of k to double, pass value
double yo[3] = { 3.0, 2.0, 1.0 };
z = cube(yo[2]); // yo[2] is an lvalue 2.0, pass value
```

✓但将上述类似的参数传递给接受引用参数的函数是被限制的。引用是一个变量的别名，则实参应是该变量（表达式不是变量）

```
double z = refcube(x + 3.0);//×should not compile
```

3.2 引用的特别之处

- ✓ 如果实参与引用参数不匹配，C++将生成临时变量
- ✓ 临时变量是一种只在调用期间有效，且具有常性的变量
- ✓ 产生临时变量的三种情况：
  - ❖ 函数传值返回
  - ❖ 强制类型转换
  - ❖ 给const引用赋值

3.2 引用的特别之处

❖ 函数传值返回：

编译器会创建一个临时变量，将sum的值拷贝给这个临时变量，返回的实际上是这个临时变量的值，函数调用结束将这个临时变量的值赋值给re

```
int add(int x,int y)
{
	int sum = x + y;

	return sum;
}
int main()
{
	int a = 10;
	int b = 20;
	int re = add(a,b);
	return 0;
}
```

3.2 引用的特别之处

❖ 强制类型转换：

强制类型转换一定会产生临时变量，因为编译器并不会对原变量进行强制类型转换，而是将原变量的值强制类型转换为所转类型的值，然后将强制类型转换后的值赋值给相应类型的临时变量，不会改变原变量的类型

3.2 引用的特别之处

❖ 给const引用赋值：

➢实参的类型正确，但不是左值

➢实参的类型不正确，但可以转换为正确的类型

```
double refcube(const double& ra); // prototype
{
        return ra * ra * ra;
}
```

```
double side = 3.0;
double* pd = &side;
double& rd = side;
long edge = 5L;
double lens[4] = { 2.0, 5.0, 10.0, 12.0 };
double c1 = refcube(side); // ra is side
double c2 = refcube(lens[2]); // ra is lens[2]
double c3 = refcube(rd); // ra is rd is side
double c4 = refcube(*pd); // ra is *pd is side
double c5 = refcube(edge); // ra is temporary variable
double c6 = refcube(7.0); // ra is temporary variable
double c7 = refcube(side + 10.0); // ra is temporary variable
```

3.2 引用的特别之处

- ✓ 如果函数调用的参数不是左值或与相应的const引用参数的类型不匹配，C++将创建类型正确的匿名变量，将函数调用的值传递给该匿名变量，让参数来引用该变量
- ✓ 应尽可能将引用声明为const：
  - ❖ 可以避免无意中修改数据的编程错误
  - ❖ 可以使函数能够处理const和非const实参，否则将只能接受非const数据
  - ❖ 可以使函数能够正确生成并使用临时变量

3.2 引用的特别之处

✓ C++新增了另一种引用-右值引用（rvalue reference）。这种引用可指向右值，是使用&&声明的：

```
double&& rref = std::sqrt(36.00); // not allowed for double&
double j = 15.0;
double&& jref = 2.0 * j + 18.5;  // not allowed for double&
std::cout << rref << '\n'; // display 6
std::cout << jref << '\n'; // display 48.5
```

✓ 新增右值引用的主要目的是，让库设计人员能够提供有些操作的更有效实现

✓ 使用&声明的应用被称为左值引用

目录

- 何时使用引用参数

4.1 何时使用引用参数

- ✓ 使用引用参数的两个原因：
  - ❖ 程序员能够修改调用函数中的数据对象
  - ❖ 通过传递引用而不是整个数据对象，可以提高程序的运行速度（当数据对象较大时（如结构和类对象））

4.1 何时使用引用参数

✓ 使用传递的值而不作修改的函数：

1）如果数据对象很小，如内置数据类型或小型结构，则按值传递

2）如果数据对象是数组，则使用指针（这是唯一选择，并将指针声明为指向const的指针）

3）如果数据对象时较大的结构，则使用const指针或const引用，以提高程序的效率（这样可以节省复制结构所需的时间和空间）

4）如果数据对象时类对象，则使用const引用（传递类对象参数的而标准方式是按引用传递）

4.1 何时使用引用参数

✓ 对于修改调用函数中数据的函数：

1）如果数据对象是内置数据类型，则使用指针

2）如果数据对象是数组，则只能使用指针

3）如果数据对象是结构，则使用引用或指针

4）如果数据对象是对象，则使用引用

这只是一些指导原则，很可能有充分的理由做出其他的选择

总结

- 引用变量
  - 引用类型的本质
  - 创建引用变量
  - 引用变量的作用
- 引用作函数参数
- 引用的属性和特别之处
- 何时使用引用参数