C++ Program Design
-- C to CPP

References

Junjie Cao @ DLUT

Summer 2022

https://github.com/jjcao-school/c

“引 用 ”的概念和应用
Reference variables

Three variable types

1. Normal variables, which hold values directly.
2. References variables: 类型名 &引用名 = 某变量名;
3. Pointer variables, which hold the address of another value (or null) and can be dereferenced to retrieve the value at the address they point to.

```
int i = 4;
int & ri = i;  // r引用了 i,  ri的类型是 int &
Int * pi = &i;
```

References

- A **reference** is a type of C++ variable that acts as an alias to another variable.
- 某个变量的引用，等价于这个变量，相当于该变量的一个别名。

```
int value = 5; // normal integer
int &ref = value; // reference to variable value
value = 6; // value is now 6
ref = 7; // value is now 7

cout << value; // prints 7
++ref;
cout << value; // prints 8
```

Using the address-of operator on a reference

```
int value = 5; // normal integer
int &ref = value; // reference to variable value
value = 6; // value is now 6
ref = 7; // value is now 7

cout << &value;
// prints 0012FF7C
cout << &ref;
// prints 0012FF7C
```

引用的概念

- 定义引用时一定要将其初始化成引用某个变量。
- 初始化后，它就一直引用该变量，不会再引用别 的变量了。
- 引用只能引用变量，不能引用常量和表达式。

References as shortcuts

```
struct Something
{
    int value1;
    float value2;
};
struct Other
{
    Something something;
    int otherValue;
};

Other other;
```

```
int &ref = other.something.value1;
// ref can now be used in place of other.somet
hing.value1
```

The following two statements are thus identi
cal:

```
other.something.value1 = 5;
ref = 5;
```

引用的概念

```
double a = 4, b = 5;
double & r1 = a;
double & r2 = r1;     // r2也引用 a
r2 = 10;
cout << a << endl;   // 输出 ？
r1 = b;
cout << a << endl;
```

引用的概念

```
double a = 4, b = 5;
double & r1 = a;
double & r2 = r1;     // r2也引用 a
r2 = 10;
cout << a << endl;  // 输出 10
r1 = b;                          // 解释这句指令？
cout << a << endl;
```

引用的概念

```
double a = 4, b = 5;
double & r1 = a;
double & r2 = r1;     // r2也引用 a
r2 = 10;
cout << a << endl;  //输 出 10
r1 = b;                          // r1并没有引用b
cout << a << endl;  //输 出 ？
```

引用的概念

```
double a = 4, b = 5;
double & r1 = a;
double & r2 = r1;     // r2也引用 a
r2 = 10;
cout << a << endl;  //输 出 10
r1 = b;                          // r1并没有引用b
cout << a << endl;  //输出 5
```

引用应用的简单示例

C语言中，如何编写交换两个整型变量值的函数?

引用应用的简单示例

C语言中，如何编写交换两个整型变量值的函数?

```
void swap( int * a, int * b)
{
        int tmp;
        tmp = * a; * a = * b; * b = tmp;
}
int n1, n2;
swap(& n1,& n2) ; // n1,n2的值被交换
```

引用应用的简单示例

➢有了C++的引用：

```
void swap( int & a, int &b)
{
        int tmp;
        tmp = a; a = b; b = tmp;
}
int n1, n2;
swap(n1,n2) ; // n1,n2的值被交换
```

引用作为函数的返回值

```
int n =4;
int & SetValue()   {   return n;  }
int main()
{
        SetValue() = 40;
        cout << n;
        return 0;
} //输出：      ?
```

引用作为函数的返回值

```
int n =4;
int & SetValue()   {   return n;  }
int main()
{
        SetValue() = 40;
        cout << n;
        return 0;
} //输出：  40
```

常引用

定义引用时，前面加const关键字，即为“常引用”

```
int n;
const int & r = n;
```

r 的类型是？？

常引用

定义引用时，前面加const关键字，即为“常引用”

```
int n;
const int & r = n;
```

r 的类型是 const int &

常引用

不能通过常引用去修改其引用的内容:

```
int n = 100;
const int & r = n;
r = 200;   //编译错
n = 300;  // 没问题
```

常引用和非常引用的转换

const T & 和T & 是不同的类型!!!

T & 类型的引用或T类型的变量可以用来初始化const T & 类型的引用。

const T 类型的常变量和const T & 类型的引用则 不能用来初始化T &类型的引用，除非进行强制类型转换。

References are implicitly const

- Reference to a constant variable
  - `const int x = 5;`
  - `int &ref = x; // ?` `// invalid, non-const reference to const object`
  - `const int &ref = x; // ?` `// OK`
- Const reference
  - `int value1 = 5;`
  - `int value2 = 6;`
  - `int &invalidRef;//? //invalid, needs to reference something`
  - `int &ref = value1; //?` `// okay, ref is now an alias for value1`
  - `ref = value2; // ?` `// assigns 6 (the value of value2) to value1 -- does NOT change the reference!`

References vs pointers

- *ptr and ref evaluate identically.

- Because **references** must be initialized to valid objects and can not be changed once set, references are generally much **safer** to use than pointers.

- However, they are also a bit more limited in functionality.

- If a given task can be solved with either a reference or a pointer, the reference should generally be preferred.

For-each loops

For-each loops

- C++11 introduces a new type of loop called a **for-each** loop

```
for (element_declaration : array)
    statement;
```

```
int main()
{
    int fibonacci[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
    for (auto number : fibonacci) // type is auto, so number has its type deduced from the fibonacci array
        std::cout << number << ' ';

    return 0;
}
```

For-each loops

```
int array[5] = { 9, 7, 5, 3, 1 };
for (auto &element: array) // The ampersand makes element a reference to th
e actual array element, preventing a copy from being made
{
    std::cout << element << ' ';
}
```

Rule: Use references or const references for your element declaration in for-each loops for performance reasons.

For-each doesn’t work with pointers to an array

```
int sumArray(int array[]){
    int sum = 0;
    for (const auto &number : array) // compile error, the size of array isn't known
        sum += number;
    return sum;
}

int main()
{
    int array[5] = { 9, 7, 5, 3, 1 };
    std::cout << sumArray(array);
    return 0;
}
```

Quiz

- Declare a fixed array with the following names: Alex, Betty, Caroline, Dave, Emily, Fred, Greg, and Holly. Ask the user to enter a name. Use a for each loop to see if the name the user entered is in the array.

- Sample output:
  - Enter a name: Betty
  - Betty was found.
  - Enter a name: Megatron
  - Megatron was not found.

- Hint: Use std::string as your array type