Transfer Bridge – C++

Qi Zhao
9/4/2019
Outline

• Function
• Pointer
Parameter passing methods

• In all the function prototype and definition examples we have seen thus far, parameters are specified as if they were simple variables, and the process of parameter passing is comparable to the process of assigning a value to a variable:
  – Each parameter is assigned the value of its corresponding argument
  – Although the value of a parameter may change during the course of a function, the value of the corresponding argument is not affected by the change to the parameter
Passing by value

• The process described on the previous slide, and illustrated in all examples thus far, is called **passing by value** - this is the default method for parameter passing.

• When arguments are passed by value:
  – a **copy** of each argument **value** is passed to its respective parameter
  – the parameter is stored in a separate memory location from the storage location of the argument, if the argument has one
  – Any valid expression can be used as an argument
Functions: Passing parameters

By value

```cpp
double squareRoot( double x )
{
    return sqrt( x );
}
```

Caller

```cpp
... double v;
cin >> v;
cout << squareRoot( v );
...
```

Local `x`  

Local `v`

Two different contexts or scopes!
Example

The program below illustrates what happens when arguments are passed by value. A tracing of the changes in the program’s variables is shown on the right.

```c
int multiply (int, int);

int main()
{
    int a, b, c;
    a = 2;
    b = 5;
    c = multiply(a, b);
    a = multiply(b, c);
    return 0;
}

int multiply (int x, int y)
{
    x = x * y;
    return x;
}
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>10</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When the program ends, the variables remaining in memory have the values shown in red
Limitations of pass by value

• Recall that a function can have either one return value or no return value
• If we want a function’s action to affect more than one variable in the calling function, we can’t achieve this goal using return value alone – remember, our options are one or none
• The next example illustrates this problem
Example – swap function

Suppose we want to write a function that swaps two values: that is, value a is replaced by value b, and value b is replaced by the original value of a. The function below is an attempt to achieve this goal.

```c
void swap (int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

The function appears to work correctly. The next step is to write a program that calls the function so that we can test it:

```c
int main()
{
    int a=2, b=6;
    cout << “Before swap, a=” << a << “ and b=” << b << endl;
    swap(a,b);
    cout << “After swap, a=” << a << “ and b=” << b << endl;
    return 0;
}
```

Output:
Before swap, a=2 and b=6
After swap, a=2 and b=6
What went wrong?

- In the swap function, parameters \( x \) and \( y \) were passed the values of variables \( a \) and \( b \) via the function call \( \text{swap}(a, b) \);
- Then the values of \( x \) and \( y \) were swapped
- When the function returned, \( x \) and \( y \) were no longer in memory, and \( a \) and \( b \) retained their original values
- Remember, when you pass by value, the parameter only gets a copy of the corresponding argument; changes to the copy don’t change the original
Building a better swap function: introducing reference parameters

- C++ offers an alternative parameter-passing method called **pass-by-reference**
- When we pass by reference, the data being passed is the **address** of the argument, not the argument itself.
- The parameter, rather than being a separate variable, is a reference to the same memory that holds the argument – so any change to the parameter is also a change to the argument.
Functions: Passing parameters

By reference

bool squareRoot( double x, double& s )
{
    if( x < 0 )
        return false;

    s = sqrt( x );
    return true;
}

double r;
double v;
cin >> v;
if(squareRoot( v, r ))
cout<< r;
else
cout<< "Imaginary";

Local x  Local s  Local r  Local v
Revised swap function

We indicate the intention to pass by reference by appending an ampersand (&) to the data type of each reference parameter. The improved swap function illustrates this:

```c
void swap (int& x, int& y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

The reference designation (&) means that x and y are not variables, but are instead references to the memory addresses passed to them.

If we had the same main program as before, the function call:

```c
swap(a,b);
```

indicates that the first parameter, x, is a reference to a, and the second parameter, y, is a reference to b.
How pass-by-reference works

• In the example on the previous slide, x and y referenced the same memory that a and b referenced
• Remember that variable declaration does two things:
  – Allocates memory (one or more bytes of RAM, each of which has a numeric address)
  – Provides an identifier to reference the memory (which we use instead of the address)
• Reference parameters are simply additional labels that we temporarily apply to the same memory that was allocated with the original declaration statement
• Note that this means that arguments passed to reference parameters must be variables or named constants; in other words, the argument must have its own address
#include <iostream>

using namespace std;

void copy (int& a, int& b, int& c)
{
    a *= 2;
    b *= 2;
    c *= 2;
}

int main ()
{
    int x = 1, y = 3, z = 7;
    copy (x, y, z);
    cout << "x = " << x << " , y = " << y << " , z = " << z;
    return 0;
}
#include <iostream>

using namespace std;

void fun(int &x)
{
    x = 20;
}

int main()
{
    int x = 10;
    fun(x);
    cout << "New value of x is " << x;
    return 0;
}
```cpp
#include <iostream>

using namespace std;

long factorial (long a)
{
    if (a > 1)
        return (a * factorial (a + 1));
    else
        return (1);
}

int main ()
{
    long num = 3;
    cout << num << "! = " << factorial (num);
    return 0;
}
```
#include <iostream>
using namespace std;

int add(int a, int b);
int main()
{
    int i = 5, j = 6;
    cout << add(i, j) << endl;
    return 0;
}

int add(int a, int b)
{
    int sum = a + b;
    a = 7;
    return a + b;
}
#include <iostream>

using namespace std;

void Sum(int a, int b, int & c)
{
    a = b + c;
    b = a + c;
    c = a + b;
}

int main()
{
    int x = 2, y = 3;
    Sum(x, y, y);
    cout << x << " " << y;
    return 0;
}
Pointer

- It is a method of locating variables (data) in your program (in memory)
- Pointer stores address in memory
  - not the value
Reference operator

• Operator & does
  – Returns address of a given variable name
  – With every program run this address changes
Reference operator

ted = &andy;
Reference operator

andy = 25;
fred = andy;
ted = &andy;
Dereference operator

• Gets a real value of a variable by a given address

• Dangerous operation
  – Leads to many bugs
  – Type of the variable must match the type of the variable at the specified address.
Dereference operator

\[ \text{beth} = *\text{ted}; \]
Summary

& is the reference operator and can be read as "address of"

* is the dereference operator and can be read as "value pointed by"
Reference and Pointer

By reference

```cpp
bool squareRoot( double x, double& s )
{
    if( x < 0 )
        return false;
    s = sqrt( x );
    return true;
}
```

```cpp
double r;
double v;
cin >> v;
if(squareRoot( v, r ))
cout<< r;
else
cout<< "Imaginary";
```
Reference and Pointer

• `int &n = m`
• `int *n = &m`

• Declaration & Initialization
• NULL
• Modification
void Func1(int x)
{
    x = x + 10;
}

...  

int n = 0;
Func1(n);
cout << "n = " << n << endl;
void Func2(int *x)
{
    (* x) = (* x) + 10;
}

...  

int n = 0;
Func2(&n);
cout << "n = " << n << endl;
void Func3(int &x) {
    x = x + 10;
}

... 

int n = 0;
Func3(n);
cout << "n = " << n << endl;
Declaring a pointer

- It is not the same thing to point to a char as to point to an int or a float.
Examples

- `int * number;`
- `char * character;`
- `float * greatnumber;`
* Means different things

• Similar to arrays
  – [] in variable declaration means size
  – [] in code execution means position in array

• With pointers
  – * in variable declaration means it is a pointer
  – * in code execution means get value by address
```cpp
#include <iostream>
using namespace std;

int main ()
{
    int firstvalue, secondvalue;
    int * mypointer;

    mypointer = &firstvalue;
    *mypointer = 10;
    mypointer = &secondvalue;
    *mypointer = 20;
    cout << "firstvalue is " << firstvalue << endl;
    cout << "secondvalue is " << secondvalue << endl;
    return 0;
}```
#include <iostream>
using namespace std;

int main ()
{
    int firstvalue = 5, secondvalue = 15;
    int * p1, * p2;

    p1 = &firstvalue;
    p2 = &secondvalue
    *p1 = 10;
    *p2 = *p1;
    p1 = p2;
    *p1 = 20;

    cout << "firstvalue is " << firstvalue << endl;
    cout << "secondvalue is " << secondvalue << endl;
    return 0;
}
Pointers and arrays

```c
int numbers [20];
int * p;
p = numbers;  // ?
```
Pointers and arrays

a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed by (a+5) = 0
Pointers and arrays

```c++
int main ()
{
    int numbers[5];
    int * p;
    p = numbers;    *p = 10;
    p++;    *p = 20;
    p = &numbers[2];    *p = 30;
    p = numbers + 3;    *p = 40;
    p = numbers;    *(p+4) = 50;
    for (int n=0; n<5; n++)
        cout << numbers[n] << "", ";
    return 0;
}
```
Pointer arithmetic

- \(*(p++)*
- \((*p)++\)
Pointer arithmetic

• *(p++) – increases pointer to the next memory location
• (*p)++ - increases the value stored in memory
Pointers and arrays

```cpp
int a[5] = {1, 2, 3, 4, 5};
cout << a[4] << endl;  // print out the last element
cout << *(a+4) << endl; //?

int *p = a;
cout << p[4] << endl; //?
cout << *(p+4) << endl; //?
```

The operations of 1-D array and pointer are almost the same.
Traversing An Array

```c
int main()
{
    int a[5];
    int *p;
    for(p = a; p < a + 5; p++)
        *p = 0;
}
int main()
{
    int a[5];
    int *p = a;
    for(int i = 0; i < 5; i++)
        p[i] = 0;
}
```
Passing Arrays or Portions to A Function

```c++
void myFun(int a[], int size){
    cout << *++a << endl;
}

void myFun(int *a, int size){
    cout << *++a << endl;
}

int main()
{
    ...
    int array[5] = {1,2,3,4,5};
    myFun(array, 5);
    myFun(array+2, 5);
    ...
}
```
Returning Pointers, nullptr

```c
int* findTheFirstNegative(int *a, int size)
{
    for(int *p = a; p < a + size; p++) {
        if(*p < 0)
            return p;
    }
    return nullptr;
}
```

- nullptr is a special location in memory, every type of pointers can point to that location.
  - `p = nullptr`
- If a pointer points to nullptr, *p is not allowed.