Cairo University
Faculty of Computers and Information
Computer Science Department

Programming-1 CS112
2017/2018

Pointers and references

Dr. Amin Allam

1 References

A reference variable is an *alias* (another name) to another variable. A reference variable is declared by appending the `&` sign to a data type. For example:

```
1 int t=7, r=8;
2 int& q=t;   // q is another name for t, any change in q affects t and vice versa
3 q=6;  cout<<q<<" "<<t<<endl;   // Prints 6 6
4 t=4;  cout<<q<<" "<<t<<endl;   // Prints 4 4
5 q=r;  cout<<q<<" "<<t<<" "<<r<<endl;   // Prints 8 8 8
6 q=3;  cout<<q<<" "<<t<<" "<<r<<endl;   // Prints 3 3 8
7 t=9;  cout<<q<<" "<<t<<" "<<r<<endl;   // Prints 9 9 8
```

Note that `q` is stated to be another name of the variable `t` in the first statement. `q` will be another name of `t` for the remainder of the code block. The statement `q=r;` does not make `q` another name of `r`, it just assigns `q` (and `t`) to the value of `r` as normal assignment statement.

Note that `q` must be stated to be another name of an existing variable in the first statement of declaration. For example, the following code is wrong:

```
1 int t=7; int& q; q=t; ✖
```

In comparison to passing by reference, the following two codes are almost equivalent:

```
1 void Fun(int& q) {q=6;}
2 int main()
3 {
4    int t=7; Fun(t);
5    cout<<t<<endl;   // Prints 6
6    return 0;
7 }
```

```
1 int main()
2 {
3    int t=7; int& q=t; q=6;
4    cout<<t<<endl;   // Prints 6
5    return 0;
6 }
```

2 Pointers

References exist only in C++. Pointers exist in C/C++ and can do the same job of references in different syntax and meaning. Pointers are useful in other jobs that can not be done by references.

Pointers are category of data types. There exist pointers to `int`, pointers to `float`, and so on. A variable `p` of type pointer to `int` is defined as `int* p`. An `int*` can hold the address of any variable of type `int`. The address of a variable is the location of the starting byte of this variable in memory. The *address-of* operator `&` takes a variable and returns its address. The *contents-of* operator `*` takes a pointer variable and returns the variable whose address is stored in that pointer.

```
1  int a=5, b=7;     // Suppose a in memory locations 1000−1003 and b in 1004−1007
2  int* p;           // Suppose p is stored in memory locations 1008−1011
3  p=&a;             // p=1000 (the address of a = the address of the first byte of a)
4  cout<<p<<endl;    // Prints 1000 (the value of p = the address of a)
5  cout<<*p<<endl;   // Prints 5 (the contents of p = the value of a)
6  cout<<&p<<endl;   // Prints 1008 (the address of p)
7  p=&b;             // p=1004 (the address of b = the address of the first byte of b)
8  cout<<p<<endl;    // Prints 1004 (the value of p = the address of b)
9  cout<<*p<<endl;   // Prints 7 (the contents of p = the value of b)
10 cout<<&p<<endl;   // Prints 1008 (the address of p)
```

The variables in the previous code are stored in memory as follow:

Variable:	a				b				p			
Address:	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011
Memory:	5				7				1004			

The following example illustrates how to use pointers to do similar effect to pass by reference. Note that the pointer itself is passed by value in this example:

```
1  void Fun(int* p)  // p will be initialized with the address of b
2  {
3      *p=6;     // Same as (*p)=6, Set the contents of p to 6, that is,
4                // Set the value of the variable whose address is stored in p to 6
5      p=0;      // Changing p will not affect q in main() because q is passed by value
6  }
7  int main()
8  {
9      int b=7;
10     Fun(&b);          // Pass the address of b
11     cout<<b<<endl;    // Prints 6
12     int* q; q=&b; b=5;
13     Fun(q);           // Pass the address of b
14     cout<<b<<" "<<*q<<endl;   // Prints 6 6
15     return 0;
16 }
```

Note that it is not possible to change the address of a variable. So, the following code is wrong:

```
1 int t=7;  &t=1000;  ✖
```

Although a pointer is actually an integer, we can not use `int` to store a pointer because:

1. The size of a pointer may be different from the size of `int`.
2. The operations we do on pointers differ from the operations we do on normal variables.
3. The contents-of operator $\star$ (and other operations which we will discuss) need to know the type of the return variable (the type of the variable whose address is stored in that pointer).

3 Dynamic allocation and pointer arithmetic

Suppose we need to allocate an array of integers, but we do not know at compile time how many integers we need to allocate. For example, the user will determine how many integers the program will allocate. Static arrays (which we studied previously) can not be used in this situation. We must use dynamic arrays instead as follows:

```
1  cin>>n;                 // Take number of integers from user
2  int* p=new int[n];      // Allocate n integers, let p the address of the first integer
3  *p=8;                   // Sets the first integer in the array to 8
4  p[0]=8;                 // Sets the first integer in the array to 8
5  *(p+1)=5;               // Sets the second integer in the array to 5
6  p[1]=5;                 // Sets the second integer in the array to 5
7  *(p+2)=9;               // Sets the third integer in the array to 9
8  p[2]=9;                 // Sets the third integer in the array to 9
9  p++;                    // Now p advances to contain the address of the second integer
10 cout<<*p++<<endl;       // Prints 5 and advances p. Same as {cout<<*p<<endl; p++;}
11 cout<<(*p)++<<endl;     // Prints 9 then increases the third integer
12 cout<<*p<<endl;         // Prints 10 (the third integer)
13 cout<<*p+5<<endl;       // Prints 15 (the third integer plus 5)
14 cout<<*--p<<endl;       // Decreases p and prints 5 (the second integer)
15 cout<<++*p<<endl;       // Increases the second integer and prints 6
16 p--;                    // Now p contains the address of the first integer
17 int* q=p+3, num;        // Now q contains the address of the fourth integer, num is int
18 num=q-p;                // num=3=the number of integers between p and q
19 if(p<q) cout<<"T";      // Prints T
20 cout<<sizeof(p)<<endl;  // Prints 8 on 64-bit systems regardless
21 cout<<sizeof(q)<<endl;  // of how many integers allocated
22 delete[] p;             // Releases the allocated memory to the operating system
```

Note that adding 1 to an `int*` actually adds `sizeof(int)` to the pointer so that it contains the address of the next integer. Similarly, adding/subtracting any number will actually multiply that number by `sizeof(int)` then adds/subtracts it to the pointer.

Subtracting two `int*` pointers returns the number of elements in-between, which is the difference divided by `sizeof(int)`.

Pointers sharing the same array can be compared, Otherwise, they can not be compared and result will be meaningless. Pointers cannot be added, multiplied, or divided.

Since `sizeof()` works in compile time, it can not detect the number of allocated elements and just returns the space of the pointer itself.

It is also possible to allocate one object, the benefit of this will be clear in other courses:

```
int* p=0;          // Initially p does not point to anything
p=new int;         // Allocate 1 integer and return its address
*p=5;              // Sets the allocated integer to 5
delete p;          // Releases the allocated memory to the operating system
```

Note that `0` is a special address: no memory location has this address. It is used as indicator that this pointer does not contain any memory address.

A pointer can be used to iterate on static array elements as follows:

```
int a[]={4,5,6};
int* p=a; // Same as int* p=&a[0]; The name of array is address of the first element
for(int i=0;i<3;i++)
   a[i]*=*p++; // Multiply each integer by itself and save the result in a[i]
for(int i=0;i<3;i++) cout<<a[i]<<" "; // Prints 16 25 36
```

A pointer can contain the address of a constant (read-only) string which can not be modified if it is not properly allocated either statically or dynamically.

```
const char* r="Hello";
*(r+1)='y';     ✖                  // The characters can not be modified
const char* words[]={"Sea", "Desert", "Air", "Ocean"};
const char* p=words[0];
cout<<*p<<endl;                    // Prints S
p=&words[1][0];
cout<<*p<<endl;                    // Prints D
p=words[2];
cout<<p<<endl;                     // Prints Air
const char** q=&words[3];          // words[3] is char*, its address stored in char**
cout<<*q<<endl;                    // Prints Ocean
cout<<**q<<endl;                   // Prints O
```

The following example combines pointers and references:

```
int x=7;
int& r=x;
int* p;
p=&x;
*p=9;
cout<<r<<endl;   // Prints 9
```

It is possible to construct a 2D array using pointers as follows:

```
int i,j;
cin>>nr>>nc;                // Get number of rows and number of columns from user
int** p=new int*[nr];       // Construct nr pointers to integers
for(i=0;i<nr;i++) p[i]=new int[nc]; // For each row, construct nc integers
for(i=0;i<nr;i++) for(j=0;j<nc;j++) cin>>p[i][j]; // Do processing
for(i=0;i<nr;i++) delete[] p[i];    // Release memory
delete[] p;                 // Release memory
```

The following is illustration of the case where `nr`=4 and `nc`=3:

p				
|				
p[0]	-->	p[0][0]	p[0][1]	p[0][2]
p[1]	-->	p[1][0]	p[1][1]	p[1][2]
p[2]	-->	p[2][0]	p[2][1]	p[2][2]
p[3]	-->	p[3][0]	p[3][1]	p[3][2]

4 Examples

Write a function that swaps two integers using references, and another function that does the same using pointers.

```
void swap1(int& a, int& b)
{
   int t=a;
   a=b;
   b=t;
}
void swap2(int* p, int* q)
{
   int t=*p;
   *p=*q;
   *q=t;
}
int main()
{
   int x=6,y=7;
   swap1(x,y);
   swap2(&x,&y);
   return 0;
}
```