

MH1402 Algorithms & Computing II

Lecture 10 Pointers and References

Wu Hongjun

Overview

- **Data in Memory**
- **Pointer**
 - new, delete
 - Pointer Arithmetic
 - Dynamic array
- **Reference**

Data in Memory

- **When a program is running (called a process), all the data in the program are stored in random access memory (RAM)**
 - **For example, when we execute xyz.exe, the program xyz.exe is loaded from hard disk into memory (RAM), then the program gets executed. After execution, the program is cleared from computer memory.**
 - **Accessing (Read/Write) RAM is about 1000 times faster than accessing harddisk (If the RAM size is not enough for running your programs, part of the harddisk may be used as virtual memory, then the computer becomes very slow)**

Data in Memory

- **Computer memory size**
 - The memory size of today's desktop/notebook computer is around 4GB~16GB
Note that 1GB (Gigabytes) is roughly 1 billion bytes (precisely, 2^{30} bytes).
 - iPad mini 3 RAM size is 1GB (16/32/64/128 GB storage)
- **The computer memory is a sequence of bytes**
 - The address of the 1st byte is 0 (zero-based indexing)
 - The address of the 2nd byte is 1
 - The address of the 3rd byte is 2
 -
 - The address of the nth byte is n-1

Data in Memory

- **When a program is running, every data is stored at an address in memory**
 - **For example, every variable we declared in C++ is stored at some address in memory**
 - **In C++, we can retrieve the address of a variable**
 - **In C++, we can access the data at any address location in memory (if we have the permission to access that memory location)**

Data in Memory

- **How to find out the memory address of a variable in C++?**

- We can apply the reference operator & to a variable.

Example:

```
int x;
```

```
cout << &x;    // &x gives the address of variable x in memory
```

```
                // the address is given in hexadecimal (base 16) format
```

- On our lab computers, the memory address is a 32-bit unsigned integer
On 64-bit machine with 64-bit OS, the memory address is a 64-bit integer

Data in Memory

- **Hexadecimal format (base 16)**
 - Similar to binary format (it groups 4 bits into a hexadecimal digit), but shorter form
 - In hexadecimal format, there are 16 hexadecimal digits (0 to 9, a to f) (there is no difference between lower and upper cases)
0 is 0, 1 is 1, 2 is 2, ... , a is 10, b is 11, c is 12, d is 13, e is 14, f is 15
Example: Binary number: 0001 0011 0101 1011
Hexadecimal: 0x135b
 - In C++, 0x is prefixed to a number to indicate that it is in hexadecimal format
Example: 0x1234 0x4567 0x2adef9

Data in Memory

- **How to access the data at a given memory address?**
 - we can apply the dereference operator ***** to an address

Example:

```
int x = 5;  
cout << *(&x); // it prints out 5;  
                // &x gives the address of x,  
                // *(&x) gives the data stored at the address &x
```


Pointer

- **Pointer is a special data type used to store the memory address.**
 - Recall that the memory address on our lab computer is a 32-bit unsigned integer
- **Pointer is declared by using the dereference operator ***

Example:

```
int* foo;  
// declare pointer foo;  
// foo is a memory address to store an integer (using 4 bytes);  
// foo is not initialized here(it is a random address);  
  
double* bar;  
// declare pointer bar;  
// bar is a memory address to store a double; (using 8 bytes)
```

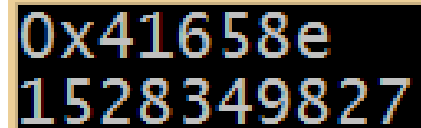
Example 1

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

int main()
{
    int* foo;    // foo is a pointer, it is a memory address;
                // it is a random address

    cout << foo << endl;    //print out the address foo
    cout << *foo << endl;    //print out the data at foo(random value)

    return 0;
}
```



```
0x41658e
1528349827
```

Example 2

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

```
int main()
```

```
{
```

```
    int* foo;           // foo is a pointer, it is a memory address;
```

```
                        // it is a random address
```

```
    *foo = 23;          // store integer 23 at address foo
```

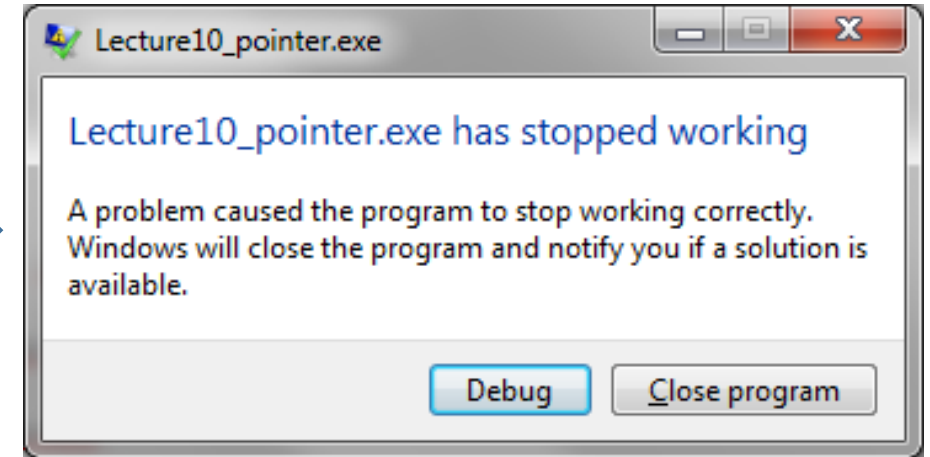
```
    cout << foo << endl; // print out the address foo
```

```
    cout << *foo << endl; // print out the data at foo(random value)
```

```
    return 0;
```

```
}
```

Run-time error



Pointer

- **Why run-time error in the previous slides?**
 - it happened since the program is not allowed to write to the address foo
 - On our computer, some memory locations are protected by the operating system, so a user's program is not allowed to write or read those addresses
Reasons: for the security and reliability of the computer

Pointer

- If we want the operating system to allocate some memory space to a program so that the program can read/write at that address, we can use the **new** operator

Example:

```
int* foo = new int;
```

```
// the computer allocates a memory space to foo (occupying 4 bytes)
```

```
// the program can write/read an integer at that address
```

Pointer

- Example 3 :

```
#include <iostream>
using namespace std;
int main()
```

```
{
    int* foo;           // foo is a pointer (a memory address);
    foo = new int;      // the OS assigns an address (with 4 bytes) to foo
                        // program can write/read data (int) at that address
    *foo = 345;         // store integer 345 at address foo

    cout << foo << endl; // print out the address
    cout << *foo << endl; // print out the data at that address
    return 0;
}
```

Different
addresses when
we run the same
program for more
than once

0x7b1618
345

0x21618
345

0x691618
345

Pointer

- The **new** operator: operating system allocates some memory space to a pointer, so now the pointer is the memory address where the program can write/read

Examples: `int* foo = new int;` `double* bar = new double;`

- The **delete** operator: operating system deallocates the memory space at an address (pointer) (that memory space has been allocated to that pointer by the new operator)
 - This operator is applied when the program no longer needs the data stored at the memory space at the pointer
 - This operator is used to prevent the waste of computer memory

Pointer

- The **delete** operator (cont.)
 - Note **that the delete operator does not delete the data in memory**
 - Note that **the delete operator does not modify the value of the pointer**
 - The delete operator simply tells the operating system that the memory space at the pointer is available for other uses (storing other variables/data)
 - The chance is high that the original data at the address (pointer) may get overwritten soon
 - You can still write data to that address (pointer) after deleting (**delete**) it
 - But there is no guarantee that your data is protected by the operating system (i.e., that data may get overwritten at any time)
 - Likely you will cause programming errors/security flaws

Pointer

- Example:

```
int* foo = new int;
```

```
*foo = 23;
```

```
cout << foo << endl;
```

```
cout << *foo << endl;
```

```
delete foo;
```


```
//.....
```

```
cout << foo << endl;
```

```
cout << *foo << endl;
```



```
0x5d2ef0  
23
```




```
0x5d2ef0  
6101160
```

Pointer

- Assign an address to pointer

```
int* foo;  
int x = 12;  
foo = &x;           // assign the address of x to foo  
cout << foo << endl; // prints the address  
cout << *foo << endl; // prints the data at that address
```

The address may be different from computer to computer, and different when program gets executed many times



```
0x28fef8  
12
```

Pointer Arithmetic

- Example:

```
int* foo;
```

```
int x = 12;
```

```
foo = &x;           // assign the address of x to foo
```

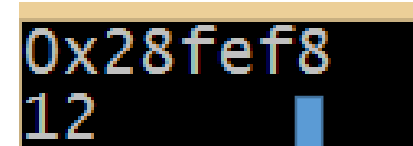
```
cout << foo << endl; // prints the address
```

```
cout << *foo << endl; // prints the data at that address
```

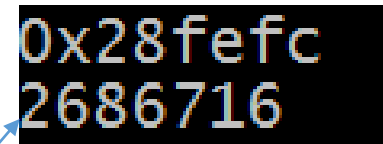
```
foo = foo+1;
```

```
cout << foo << endl; // prints out the address.
```

```
cout << *foo << endl; // prints the data at that address
```



0x28fef8
12



0x28fefc
2686716

Some random value

Pointer Arithmetic

- **Why is the address foo increased by 4 when we add 1 to it in the previous slides?**
 - Reason: foo is declared as the address to store an int type data (4 bytes), (foo + 1) is the next address to store an int type data, so the address should be increased by four bytes.
 - If foo is declared as the address to store a double type data (8 bytes), (foo + 1) is the address increase by 8 bytes.

Pointer Arithmetic

- Example 2:

```
double* foo;
```

```
double x = 0.6;
```

```
foo = &x;           // assign the address of x to foo
```

```
cout << foo << endl; // print out the address.
```

0x28fef0

```
cout << *foo << endl; // print out the data at that address
```

0.6

```
foo = foo+1;
```

```
cout << foo << endl; // print out the address.
```

0x28fef8

```
cout << *foo << endl; // print out the data at that address)
```

6.95224e-308

Some random value

Pointer and Array

- The starting address of an array arr is given as **&arr[0]**
 - It is also given as **arr**

```
int arr[10];
```

```
// the following two statements print the same address
```

```
cout << arr << endl;
```

```
cout << &arr[0] << endl;
```

Pointer and Array

- We can assign the starting address of an array to a pointer

```
int arr[5] = {0, 1, 2, 3, 4};
```

```
int* p;
```

```
p = arr;
```

```
cout << *p << endl;    // print the value of arr[0];
```

```
cout << *(p+1) << endl; // print the value of arr[1];
```

```
cout << *(p+2) << endl; // print the value of arr[2]
```

Dynamic Array

- **The new operator and delete operator can be applied to other data types and data structures**
 - These operators allow the dynamic memory allocation at run-time in C++
 - In C, the functions malloc and free are used (they can also be used in C++)
- **Now we apply new and delete operators to array**
 - Dynamic array

Dynamic Array

- In C++, the array size should be a constant in source code
 - It is inconvenient if the array size is only available during program execution
 - How to handle this problem?
We can use dynamic array (pointer is used)
Example:

```
int x;  
x = 10;  
int* foo = new int[x]; // declare foo as a pointer,  
                        // 10*4 bytes are allocated to foo  
for (int i = 0; i < x; i++)  
    foo[i] = i;         // access the dynamic array foo.  
delete [] foo;          // delete foo after use. Note []
```

Dynamic Array

- **What is the difference between dynamic array and vector?**
 - vector is built on dynamic array (but with wrapper and member functions)
- **When to use vector, and when to use dynamic array?**
 - Vector is more convenient to use than dynamic array: you can use vector to replace dynamic array completely
 - But you still need to know dynamic arrays in order to read C++ codes.

Reference

- We have already learned passing by reference to a function: the modified value of the variable (being passed by reference to a function) gets retained after the function call
- Reference is a data type, it is the synonym of another variable
 - The declaration is given as:
type& ref-name
 - Example:
int x;
int& foo = x;
(now foo is the synonym of x, they use the same memory address)

Reference

```
int x = 3;
```

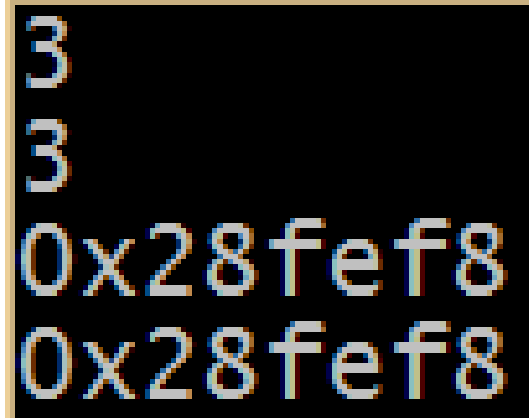
```
int& foo = x;
```

```
cout << x << endl;
```

```
cout << foo << endl;
```

```
cout << &x << endl;
```

```
cout << &foo << endl;
```



```
3
3
0x28fef8
0x28fef8
```

Reference

- A **reference must be initialized to a variable.**
 - Example:
`int& foo = 56; // Error`
 - Recall that **when we pass a variable by reference to a function, the argument must be a variable**
- A **reference must be initialized when declared.**
 - Example:
`int& foo; // Error`