Pointers and References

Lecture 13-2

Discouragement and failure are two of the surest stepping stones to success.
Dale Carnegie

Addresses in C++

- Each variable you create is assigned a location in the computer's memory.
- Memory of a computer is like a succession of one-byte memory cells. These cells are ordered in a way so that data representations whose size is larger than one byte can occupy consecutive memory cells.
- Generally, C++ programs do not actively decide the exact memory addresses where their variables are stored (usually done by the OS).

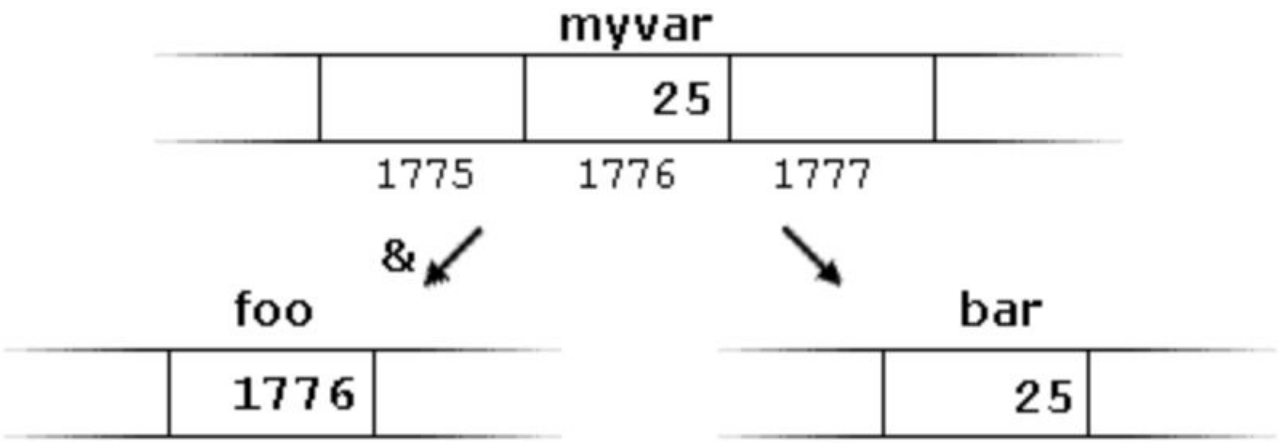
Address-of Operator

- The address of a variable can be obtained by preceding the variable name with an ampersand (`&`) sign, known as the address-of operator.
- For example, `foo = &myvar` will assign the address of `myvar` to `foo`.

Address-of Operator Example

- The diagram shows what happens after running the following code fragment.

```
1 myvar = 25;
2 foo = &myvar;
3 bar = myvar;
```



Assume that `myvar` is stored in the memory address 1776.

What Are Pointers?

- As just seen, a variable (`foo` in this case) that stores the address of another variable is called a ***pointer.***
- Pointers are powerful features that differentiate C++ from other programming languages like Java, JavaScript, Python, etc.

Usage of Pointers

- Modify variables inside another function.
- Optimize for the memory usage
  - e.g) free unused space right away
- Dynamically allocate a large memory space in the heap
- Implement advanced data structure like a linked list or tree
- Handle overriding and dynamic binding for inherited classes

Declaring Pointers

- Since pointers can be used to access the variable they point to, pointers have different properties when they point to different data types.
- Therefore, the declaration of a pointer needs to include information about the data type that the pointer will point to.

```
1 int * number;
2 char * character;
3 double * decimals;
```

Dereference Operators

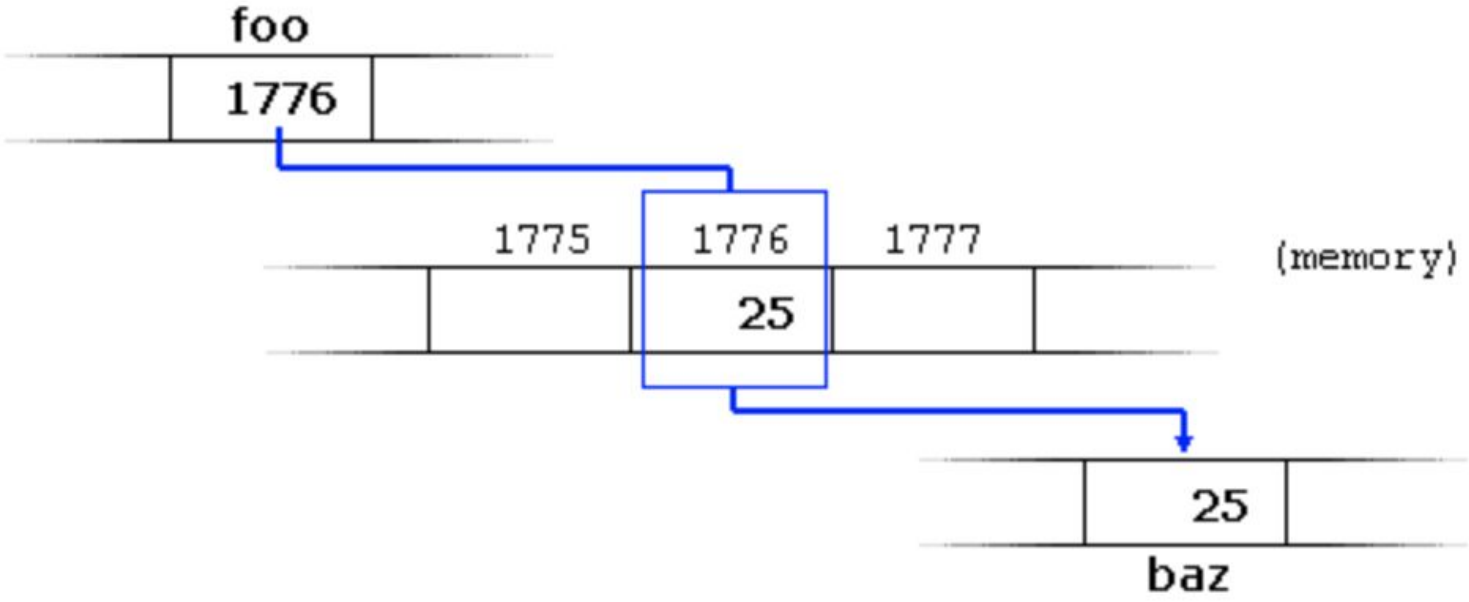
- Through pointers, we can directly access the variables that they point to using what is called the dereference operator (`*`) in front of the pointer name.

Dereference Operator Example

- Following the previous example,

```
baz = *foo;
```

will assign the value of 25 to the variable `baz`.



Dereference Operator vs. Pointer Declaration

- Note that the asterisk (`*`) is used both as dereference operator and as pointer declaration.

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

int main ()
{
  int intvalue;
  int * mypointer; // pointer declaration

  mypointer = &intvalue;
  *mypointer = 10; // dereference operator
  cout << "value is " << intvalue << endl; // 10
  return 0;
}
```

Arrays and Pointers

- An array variable can be considered a constant pointer to the array's first element.
- Assigning the pointer variable the value of a same-type array variable is perfectly valid.
- However, an array variable cannot be assigned a new address.

```
int myarray[20];
int* myptr;
myptr = myarray; // valid
myarray = myptr; // invalid
```

Brackets and Dereference Operator

- Brackets `[ ]` which specifies the index of an array is actually a dereference operator known as *offset* operator.
- It works the same as dereference operator, but it adds the number between the brackets to the address of the array variable.

```
a[5] = 7;
```

is the same as

```
*(a+5) = 7;
```

Arrays and Pointers Example

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

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;
}
```

Output

```
10, 20, 30, 40, 50,
```

Array of Pointers

- Of course, you may define an array of pointers.

```
int main()
{
    int a[10];
    int i;
    for(i=0;i<10;i++)
        a[i]=i+1;
    cout << "Without pointers :\n";
    for(i=0;i<10;i++)
        cout << a[i] << " ";
    cout << "\n";
    ...
```

Array of Pointers

- Of course, you may define an array of pointers.

```
    ...
    int *p[10]; // 10 pointers to point to each element in a
    for(i=0;i<10;i++)
        p[i] = &a[i]; // get the address of each element
    cout << "Using pointers :\n";
    for(i=0;i<10;i++)
        cout << *p[i] << " ";

    return 0;
}
```

Output

```
Without pointers:
1 2 3 4 5 6 7 8 9 10
Using pointers:
1 2 3 4 5 6 7 8 9 10
```

Sizeof Operator

- Returns the size of operand in bytes
  - Done at compile time
  - Unsigned int
- Can be used with:
  - Variable names
  - Type names
  - Constant value
  - Parenthesis only required for `sizeof(type_name)`

Sizeof Operator

- For arrays, `sizeof` returns: `(size of an element) * (number of elements)`.

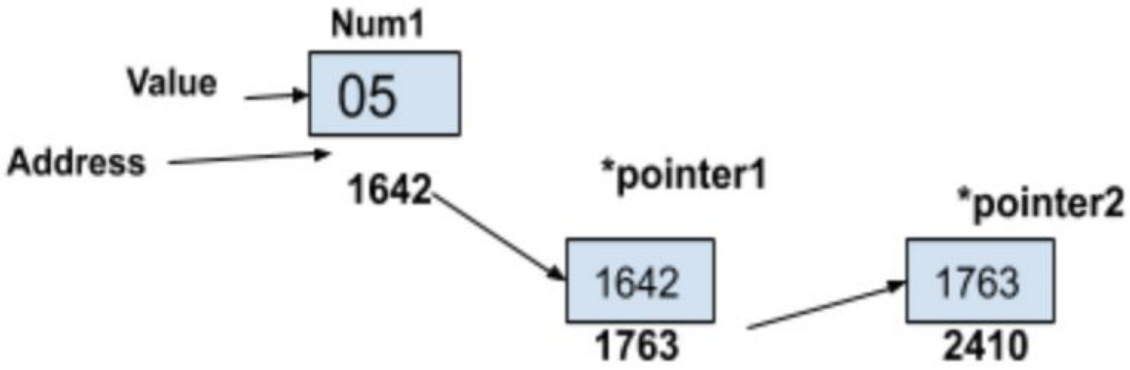
```cpp
#include <iostream>
using namespace std;

int main()
{
    int my_arr[10];
    cout << sizeof(my_arr) << endl; // 40

    return 0;
}
```

Sizeof Operator Example

```
void increase (void* data, int psize)
{
    if ( psize == sizeof(char) )
    { char* pchar; pchar=(char*)data; ++(*pchar); }
    else if (psize == sizeof(int) )
    { int* pint; pint=(int*)data; ++(*pint); }
}
int main ()
{
    char a = 'x';
    int b = 1602;
    increase (&a,sizeof(a));
    increase (&b,sizeof(b));
    cout << a << ", " << b << '\n'; // y, 1603
    return 0;
}
```

Pointer to Pointer

- A pointer that stores the address of another pointer is called a *pointer to pointer.*



Pointer to Pointer

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

int main()
{
    int a = 10;
    int *p = &a; // stores the address of integer a
    int **q = &p; // stores the address of pointer p

    cout << "a address: " << p << "\n";
    cout << "p address: " << q << "\n";
    cout << "p: " << p << " " << *q << "\n";
    cout << "a: " << a << " " << *p << " " << **q << "\n";
    return 0;
}
```

Output

```
a address: 0x7ffee07ee8a8
p address: 0x7ffee07ee8a0
p: 0x7ffee07ee8a8 0x7ffee07ee8a8
a: 10 10 10
```

Pointers and `const`

- We can define pointers so that only reading the value of the address that the pointer is pointing to is possible.
- This is done by qualifying the type pointed by the pointer as `const`.

```
int x;
int y = 10;
const int * p = &y; // can also be defined as int const * p = &y;
x = *p;             // ok: reading p
*p = x;             // error: modifying p, which is const-qualified
```

Pointers and `const`

- We can define pointers so that the value of the pointer cannot be changed.
- This is done by qualifying the pointer as `const`.

```
int x;
int y = 10;
int * const p = &y;
*p = 30;          // ok: changing the value pointed by p
p = &x;           // error: modifying p, which is const-qualified
```

Pointers and `const`

- Of course, we can make different combinations using these two.

```
int x;
int *       p1 = &x;  // non-const pointer to non-const int
const int *       p2 = &x;  // non-const pointer to const int
int * const p3 = &x;  // const pointer to non-const int
const int * const p4 = &x;  // const pointer to const int
```

Pointers and `const` Example

```
void increment_all (int* start, int* stop)
{
  int * current = start;
  while (current != stop) {
    ++(*current);  // increment value pointed
    ++current;     // increment pointer
  }
}
void print_all (const int* start, const int* stop)
{
  const int * current = start;
  while (current != stop) {
    cout << *current << '\n';
    ++current;     // increment pointer
  }
}
```

Pointers and `const` Example

```
int main ()
{
  int numbers[] = {10,20,30};
  increment_all (numbers,numbers+3);
  print_all (numbers,numbers+3);
  return 0;
}
```

Output

```
11
21
31
```

Function Pointers

- You can also make pointers point to functions in C++.
- The typical use of function pointers is passing functions as arguments to another function.
- Declaration is done as the following:

```
returntype (* functionptrname) (arg1, arg2, ...) = ...
```

Function Pointer Example

```
int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}
```

Function Pointer Example

```
int main ()
{
  int m,n;
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout << n;
  return 0;
}
```

Output

```
8
```

Void Pointers

- Generic form of pointer that can be used to store reference of any type of variable.
- Typecasting needs to be done before dereferencing void pointers.

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

int main() {
    int a = 0;
    void* test;
    test = &a;
    cout << * (int*) test << endl; // 0
    return 0;
}
```

Invalid Pointers

- Pointers can point to any address, including addresses that do not refer to any valid element.
- Typical examples are uninitialized pointers and elements of array that are out of bounds.

```
int * p;                  // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20; // element out of bounds
```

- However, the above code fragment does not raise any errors.

Invalid Pointers

- Problem arises when trying to dereference these invalid pointers.
- Trying to do so results in undefined behavior, ranging from runtime errors to accessing some random garbage value.

```
int* ptr;
int arr[3];

cout << *(arr+5) << endl; // -364201784 (for example)
cout << *ptr << endl; // -17958193 (for example)
```

Dangling Pointer

- An invalid pointer which points to a memory location which no longer exists - that is, the memory location has been deleted and returned to the system.

```
int main()
{
    int *a = new int[2];
    a[0] = 10; a[1] = 20;
    int *p = &a[0];
    delete a;

    cout << "Value is: " << *p << "\n"; // undefined behavior
    return 0;
}
```

Dangling Pointer from Stack Memory

- Returning reference / pointer to a variable defined in the stack memory will be invalid once the function is done running.

```
int * test (){
   int arr[10];
   return arr;
}
int main()
{
    int * testptr = test();
    cout << *testptr << endl; // undefined behavior
    return 0;
}
```

Null Pointer

- You can explicitly tell the pointer to point to nowhere using the below three methods.

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

int main()
{
    int * p = NULL;
    int * t = 0;
    int * r = nullptr;
    cout << "VALUE : " << p << endl; // VALUE : 0x0
    cout << "VALUE : " << t << endl; // VALUE : 0x0
    cout << "VALUE : " << r << endl; // VALUE : 0x0

    return 0;
}
```

Pointer Arithmetic

- Conducting arithmetic operations on a pointer is a little bit different from conducting them on regular integers.
- Only additions and subtractions are possible, and their behaviors are dependent on the size of the data type to which they point.

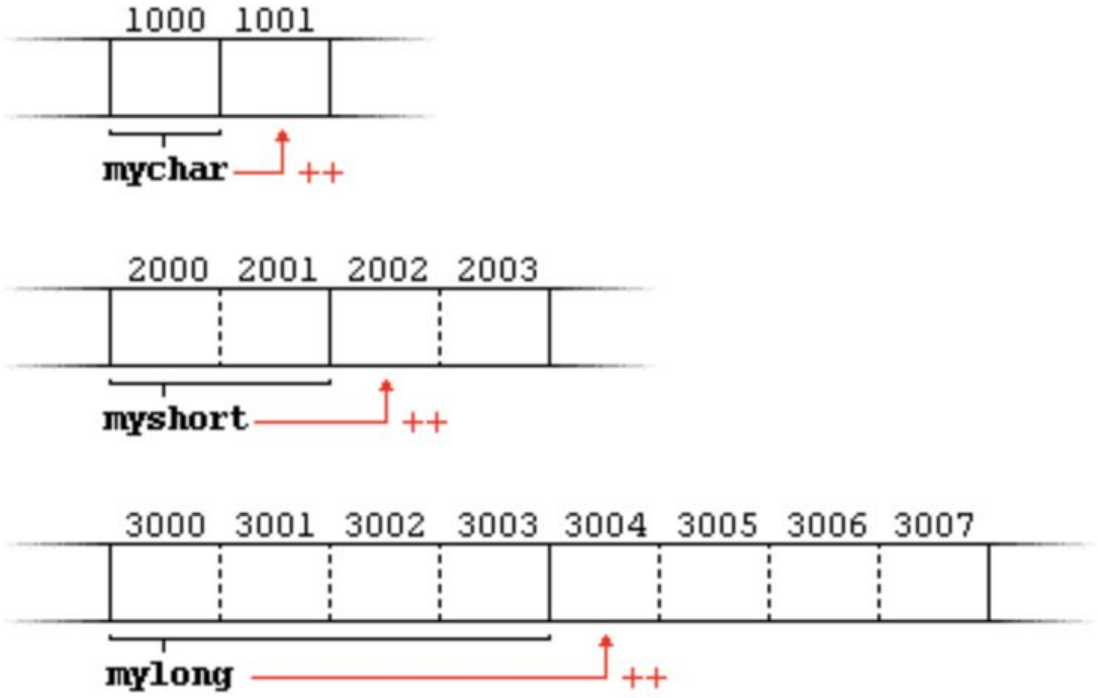
Pointer Arithmetic Example

- Different types have different data sizes, but the exact value is dependent on the system.
- Let's assume that `char` takes 1 byte, `short` takes 2 bytes, and `long` takes 4 bytes.
- We have the following pointers defined who point to the designated memory cells:

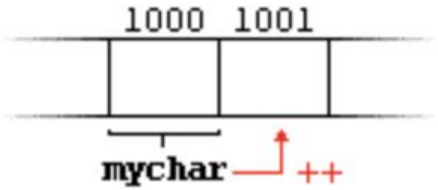
```
char * mychar; // address 1000
short * myshort; // address 2000
long * mylong; // address 3000
```

Pointer Arithmetic Example

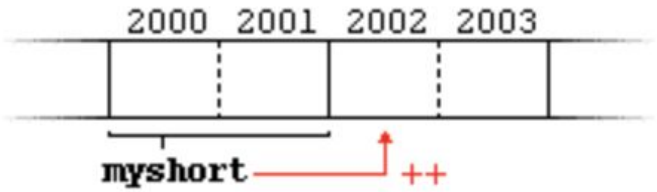
- Calling the increment operator (++) on these pointers will result in the following modification to the values of the pointers:



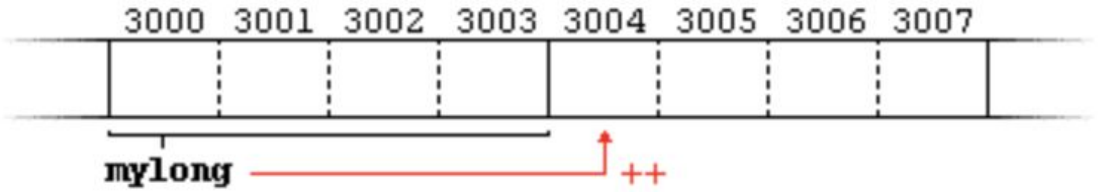
Pointer Arithmetic Example



The size of `char` is 1 byte, so 1 is added to the pointed address.



However, adding one to `myshort` results in adding 2 to the address to point to the next available memory address.



Likewise, 4 is added to the address pointed by `mylong` to point tso the next available memory address.

String Literals and Pointers

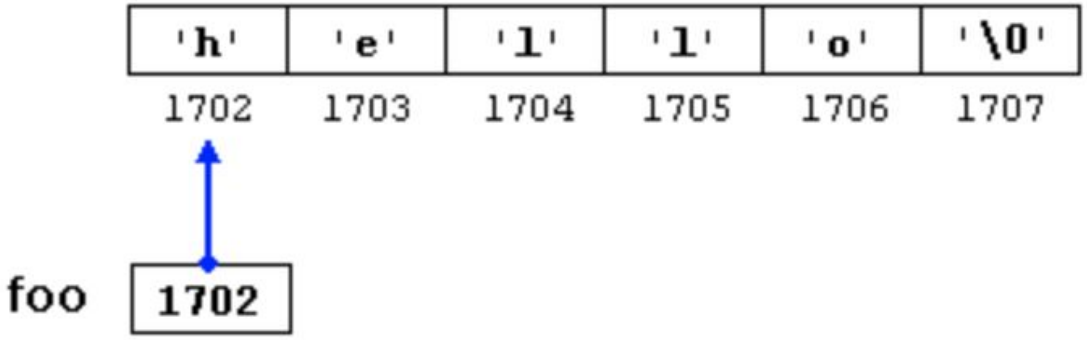
- String literals are arrays containing null-terminated character sequences.
- Since a string literal cannot be changed, it is defined as a proper array of type `const char`.

String Literal and Pointers Example

- For example,

```
const char * foo = "hello";
```

declares an array with the literal representation for “hello”, and then a pointer to its first element is assigned to `foo`.



- Remember, `foo` is a pointer with value 1702, not `‘h’` nor `“hello”`.

String Literal and Pointers Example

- We can declare an array of pointers pointing to string literals.

```
const char * args[4] = { "Hearts", "Diamonds", "Clubs", "Spades" };
```

- This kind of definition is commonly used as command-line arguments to the main function.