ICS 2105:
Data Structures and Algorithms

Pointers

Acknowledgement

- Notes adapted from cplusplus.com

Learning Outcomes

- By the end of this lesson, a learner should be able to:
  - Describe pointer operators.
  - Declare and initialize pointers.
  - Write, test, and debug C++ programs using pointers.

Pointers

- Variables have been explained as locations in the computer's memory which can be accessed by their identifier (their name).
  - In this way, the program does not need to care about the physical address of the data in memory; it simply uses the identifier whenever it needs to refer to the variable.

Pointers

- For a C++ program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address.
- These single-byte memory cells are ordered in a way that allows data representations larger than one byte to occupy memory cells that have consecutive addresses.

Pointers

- This way, each cell can be easily located in the memory by means of its unique address.
  - E.g., the memory cell with the address 1776 always follows immediately after the cell with address 1775 and precedes the one with 1777, and is exactly one thousand cells after 776 and exactly one thousand cells before 2776.

Pointers

- When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address).
- Generally, C++ programs do not actively decide the exact memory addresses where its variables are stored.

Pointers

- Fortunately, that task is left to the environment where the program is run - generally, an operating system that decides the particular memory locations on runtime.
- However, it may be useful for a program to be able to obtain the address of a variable during runtime in order to access data cells that are at a certain position relative to it.

What is a Pointer?

- The variable that stores the address of another variable is what in C++ is called a *pointer*.
- Pointers are a very powerful feature of the language that has many uses in lower level programming.

Pointer Operators

- Address-of operator (&).
- Dereference operator (*)

Address-of operator (&)

- The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as *address-of operator*.
- For example: foo = &myvar;
  - This would assign the address of variable myvar to foo; by preceding the name of the variable myvar with the *address-of operator* (&), we are no longer assigning the content of the variable itself to foo, but its address.

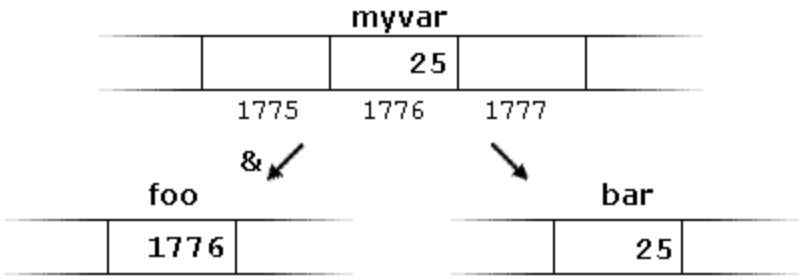
Address-of operator (&)

- The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that myvar is placed during runtime in the memory address 1776.
- In this case, consider the following code fragment:

```
myvar = 25;
foo = &myvar;
bar = myvar;
```

Address-of operator (&)

- The values contained in each variable after the execution of this are shown in the following diagram:



Dereference operator (*)

- A variable which stores the address of another variable is called a *pointer*.
- Pointers are said to "point to" the variable whose address they store.

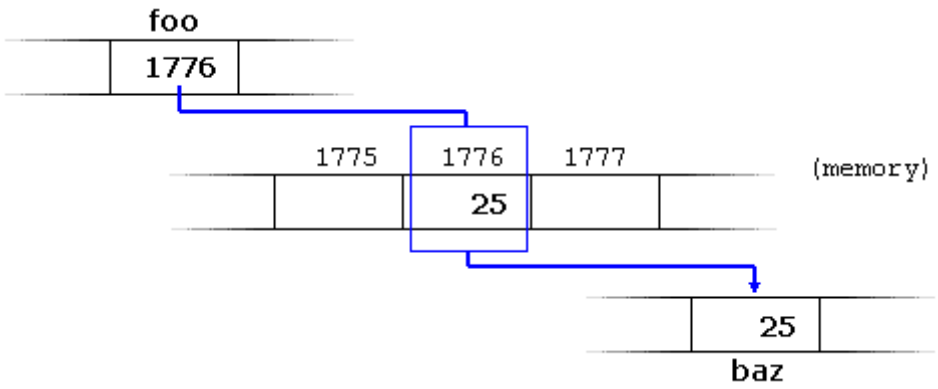
Dereference operator (*)

- An interesting property of pointers is that they can be used to access the variable they point to directly.
- This is done by preceding the pointer name with the *dereference operator* (*).
  - The operator itself can be read as "value pointed to by".

Dereference operator (*)

- E.g. baz = *foo;
  - could be read as: "baz equal to value pointed to by foo", and the statement would actually assign the value 25 to baz, since foo is 1776, and the value pointed to by 1776 would be 25.

Dereference operator (*)



Dereference operator (*)

- The reference and dereference operators are thus complementary:
  - & is the *address-of operator*, and can be read simply as "address of"
  - * is the *dereference operator*, and can be read as "value pointed to by"

Declaring pointers

- Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it points to a char than when it points to an int or a float.
- Once dereferenced, the type needs to be known.
- And for that, the declaration of a pointer needs to include the data type the pointer is going to point to.

Declaring pointers

- The declaration of pointers follows this syntax:

```
type * name;
```

  - where type is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to.

Declaring pointers

```
int * number;
char * character;
double * decimals;
```

- These are three declarations of pointers.
  - Each one is intended to point to a different data type, but, in fact, all of them are pointers and all of them are likely going to occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the program runs).

Declaring pointers

- Note that the asterisk (*) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the *dereference operator* seen a bit earlier, but which is also written with an asterisk (*).
  - They are simply two different things represented with the same sign.

Sample Pointer Program

```
// my first pointer
#include <iostream>
using namespace std;

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

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue <<
'\n';
  cout << "secondvalue is " << secondvalue <<
'\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

Using Pointers

- Notice that even though neither `firstvalue` nor `secondvalue` are directly set any value in the program, both end up with a value set indirectly through the use of `mypointer`.
- This is how it happens:
  - First, `mypointer` is assigned the address of firstvalue using the address-of operator (`&`). Then, the value pointed to by `mypointer` is assigned a value of `10`. Because, at this moment, `mypointer` is pointing to the memory location of `firstvalue`, this in fact modifies the value of `firstvalue`.

Using Pointers

- A pointer may point to different variables during its lifetime in a program.

```
// more pointers
#include <iostream>
using namespace std;

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

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value
pointed to by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

Pointers and Arrays

- The concept of arrays is related to that of pointers.
- Arrays work very much like pointers to their first elements, and, an array can always be implicitly converted to the pointer of the proper type.
- For example, consider these two declarations:

```
int myarray [20];
int * mypointer;
```

- The following assignment operation would be valid: `mypointer = myarray;`

Pointers and Arrays

- After that, `mypointer` and `myarray` would be equivalent and would have very similar properties.
- The main difference being that `mypointer` can be assigned a different address, whereas `myarray` can never be assigned anything, and will always represent the same block of 20 elements of type `int`.

Example Program that mixes Arrays and Pointers

```
// more pointers
#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;
}
```

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

Pointers and Arrays

- Pointers and arrays support the same set of operations, with the same meaning for both.
  - The main difference being that pointers can be assigned new addresses, while arrays cannot.
- The brackets ([]) in array were explained as specifying the index of an element of the array.
  - These brackets are a dereferencing operator known as *offset operator*.
  - They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced.

Pointers and Arrays

- For example:

```
a[5] = 0;        // a [offset of 5] = 0
*(a+5) = 0;      // pointed to by (a+5) = 0
```

- These two expressions are equivalent and valid, not only if a is a pointer, but also if a is an array.
- Remember that if an array, its name can be used just like a pointer to its first element.

Pointer Initialization

- Pointers can be initialized to point to specific locations at the very moment they are defined:

```
int myvar;
int * myptr = &myvar;
```

- The resulting state of variables after this code is the same as after:

```
int myvar;
int * myptr;
myptr = &myvar;
```

Pointer Initialization

- When pointers are initialized, what is initialized is the address they point to (i.e., myptr), never the value being pointed (i.e., *myptr).
- Therefore, the code above shall not be confused with:

```
int myvar;
int * myptr;
*myptr = &myvar;
```

Pointer Initialization

- Pointers can be initialized either to the address of a variable (such as in the case above), or to the value of another pointer (or array):

```
int myvar;
int *foo = &myvar;
int *bar = foo;
```

Pointer Arithmetics

- To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer types.
  - Only addition and subtraction operations are allowed; the others make no sense in the world of pointers.

Pointer Arithmetics

- But both addition and subtraction have a slightly different behavior with pointers, according to the size of the data type to which they point.
- Fundamental data types have different sizes.
  - For example: char always has a size of 1 byte, short is generally larger than that, and int and long are even larger; the exact size of these being dependent on the system. For example, let's imagine that in a given system, char takes 1 byte, short takes 2 bytes, and long takes 4.

Pointer Arithmetic

- Suppose now that we define three pointers in this compiler:

```
char *mychar;
short *myshort;
long *mylong;
```

and that we know that they point to the memory locations 1000, 2000, and 3000, respectively.

Pointer Arithmetics

- Therefore, if we write:

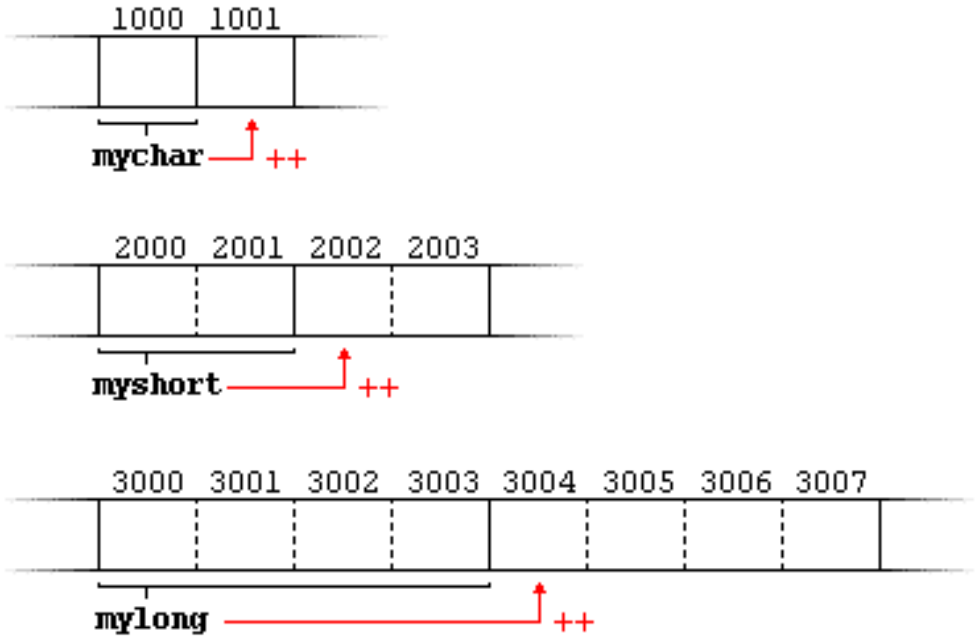
```
++mychar;
++myshort;
++mylong;
```

- mychar, as one would expect, would contain the value 1001.
- But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been incremented only once.

Pointer Arithmetics

- The reason is that, when adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer.

Pointer Arithmetics



End of Lesson