Data Structures and Algorithms
(ES221)

Pointers

Dr. Zubair Ahmad

Data Types

Data Type	Size	Description
boolean	1 byte	Stores true or false values
char	1 byte	Stores a single character/letter/number, or ASCII values
int	2 or 4 bytes	Stores whole numbers, without decimals
float	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits
double	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits

Abstract Data Types

A data type that is defined by its behavior (operations) rather than its implementation

Abstract data types are purely theoretical entities used

- to simplify the description of abstract algorithms,
- to classify and evaluate data structures, and
- to formally describe the type systems of programming languages.

However, an ADT may be

- implemented by specific data types or data structures, in many ways and in many programming languages; or
- described in a formal specification language.

Abstract Data Types

ADTs hide the implementation details of data and expose only the necessary operations.

Defined Operations:

•An ADT specifies a set of operations that can be performed on the data. For example:

- Adding, removing, and accessing elements in a list.
- Pushing and popping elements in a stack.

abstract stack

Example: A complete functional-style definition of a stack ADT could use the three operations:

- **push**: takes a stack state and an arbitrary value, returns a stack state;
- **top**: takes a stack state, returns a value;
- **pop**: takes a stack state, returns a stack state;

Difference between Data Types & Abstract Data Types?

Data Type

:Think of it as raw materials, like wood or metal (e.g., int, float).

Specifies the type of data

Abstract Data Type:

Think of it as a piece of furniture made from those materials, like a chair (e.g., stack or queue). The user only cares about what the chair does (sit), not how it was made.

Specifies the behavior of a data structure.

what operations can be performed on the data (e.g., insert, delete, search) without describing **how** the data is stored or the operations are implemented.

Real-Life Analogy for Pointers

- Imagine you have a **house** (a variable) and an **address** written on a piece of paper (a pointer). The address tells you where the house is located.

Real-Life Analogy for Pointers

The House (Variable):

- The house represents the **actual data** or value stored in memory.
- Example: The house contains a value, like "10 chairs."

The Address (Pointer):

- The address is not the house itself, but it tells you where the house is.
- In programming, this is like a pointer storing the memory address of a variable.

The Paper with the Address (Pointer Variable):

- The paper itself is the pointer variable. It stores the location (address) of the house, so you know where to find it.

Real-Life Analogy for Pointers

- If someone gives you the address (pointer), you can go to the house and see what’s inside (dereferencing the pointer).

- If you lose the paper (pointer), you can no longer find the house, even though the house still exists in its location.

Real-Life Analogy for Pointers

Accessing the Value:

To see the value of the variable (e.g., "10 chairs"), you need to visit the house using the address written on the paper.

Changing the Value:

If you write "15 chairs" inside the house, it means you have updated the variable value using the pointer.

Pointer Arithmetic:

If the houses are arranged in a row (like an array), moving to the next house is like incrementing the pointer to point to the next variable.

Pointers

Variables that store the memory address of another variable

Fundamental concepts in programming, especially in C++, and are crucial for working with memory

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

Pointers

Variables that store the memory address of another variable

```
data_type *pointer_name;
```

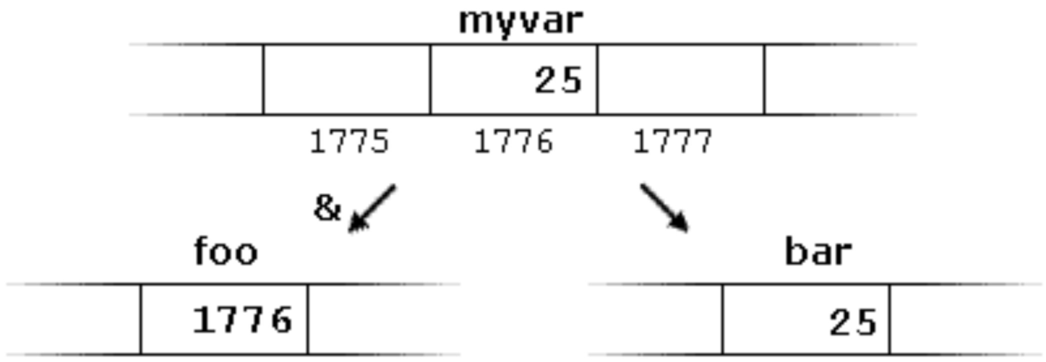
data_type: Specifies the type of the variable the pointer will point to..

*: Indicates that the variable is a pointer

pointer_name: The name of the pointer

Pointers

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



First, we have assigned the value 25 to **myvar** (a variable whose address in memory we assumed to be 1776).

The second statement assigns **foo** the address of **myvar**, which we have assumed to be 1776.

Finally, the third statement, assigns the value contained in **myvar** to **bar**. This is a standard assignment operation

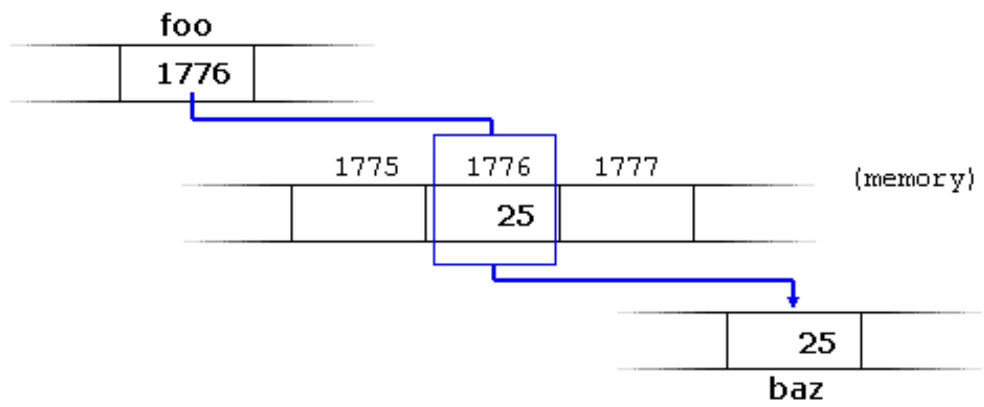
The main difference between the second and third statements is the appearance of the *address-of operator* (&).

Pointers

Dereference operator (*)

```
andy = 25;
fred = andy;
ted = &andy;
```

```
baz = *foo;
```



This 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 (following the example above) would be 25.

Pointers

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"

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

```
myvar == 25
&myvar == 1776
foo == 1776
*foo ==
```

Memory Diagrams

Memory diagrams help visualize how pointers interact with memory, showing how variables and addresses are related

```
int a = 5;
int *p = &a;
```

Address	Value	Name
0x7ffeabcd10	5	a
0x7ffeabcd14	0x7ffeabcd10	p

The variable a is stored at 0x7ffeabcd10 with a value of 5.
The pointer p stores the address of a

Pointers (Example(1))

```
int main() {
  string food = "Pizza";  // Variable declaration
  string* ptr = &food;    // Pointer declaration

  // Reference: Output the memory address of food with the pointer
  cout << ptr << "\n";

  // Dereference: Output the value of food with the pointer
  cout << *ptr << "\n";
  return 0;
}
```

Note that the * sign can be confusing here, as it does two different things in our code:

- When used in declaration (string* ptr), it creates a **pointer variable**.
- When not used in declaration, it act as a **dereference operator**.

Pointers (Example (1))

```
int main ()      {
        int value1 = 5, value2 = 15;
        int * mypointer;
        mypointer = &value1;
        *mypointer = 10;
        mypointer = &value2;
        *mypointer = 20;
        cout << "value1==" << value1 << "/
value2==" << value2;
        return 0;
  }
```

Output: value1==10 / value2==20

Pointers (Example (2))

```
int main () {
     int value1 = 10, value2 = 15;
    int *p1, *p2;
    p1 = &value1;   // p1 = address of value1
    p2 = &value2;   // p2 = address of value2
    *p1 = 10;       // value pointed by p1 = 10
    *p2 = *p1;     // value pointed by p2 = value pointed by p1
    p1 = p2;        // p1 = p2 (value of pointer copied)
    *p1 = 20;      // value pointed by p1 = 20
    cout << "value1==" << value1 << "/ value2==" << value2;
    return 0;
}
```

Output: value1==10 / value2==20

Pointers (Example)

```
int main () {
    int var1 = 10, var2 = 20, var3 = 30;
     int *ptr1, *ptr2, *ptr3;
     ptr1 = &var1;  //  p1 = address of var1
     ptr2 = &var2; //   p2 = address of var2
     ptr3 = &var3; //   p3 = address of var3
     *ptr1 = 5;           // value pointed by ptr1 = 10
     *ptr3 = *ptr1;       // value pointed by ptr2 = value pointed by ptr1
      ptr3 = ptr2;
      ptr2 = ptr1;
     *ptr2 = 7;
     *ptr3 = 9;
     cout << "var1 = " << var1 << " | var2 = " << var2 <<" |  var3 = " <<
     var3 << endl;
     return 0;
 }
```

<u>Output:</u> var1 = 7 | var2 = 9 | var3 = 5

Pointer Arithmetic

A way to perform operations directly on memory addresses

When you increment a pointer (e.g., ptr++), you're not simply adding 1 to the address. Instead, you're advancing the pointer by the size of the data type it points to.

For example, if ptr is an int* (and an int is 4 bytes), ptr++ will increase the address stored in ptr by 4 bytes. This makes the pointer point to the next integer in memory.

Pointer Arithmetic

A way to perform operations directly on memory addresses

Similarly, decrementing a pointer (e.g., ptr--) moves the pointer backward in memory by the size of the data type.

Pointer Arithmetic

A way to perform operations directly on memory addresses

You can add an integer to a pointer (e.g., **ptr + n**). This moves the pointer forward by n times the size of the data type.

For example, if ptr is a float* (and a float is 4 bytes), ptr + 3 will increase the address by 12 bytes (3 * 4 bytes).

Subtracting an integer from a pointer (e.g., ptr - n) moves the pointer backward by n times the size of the data type.

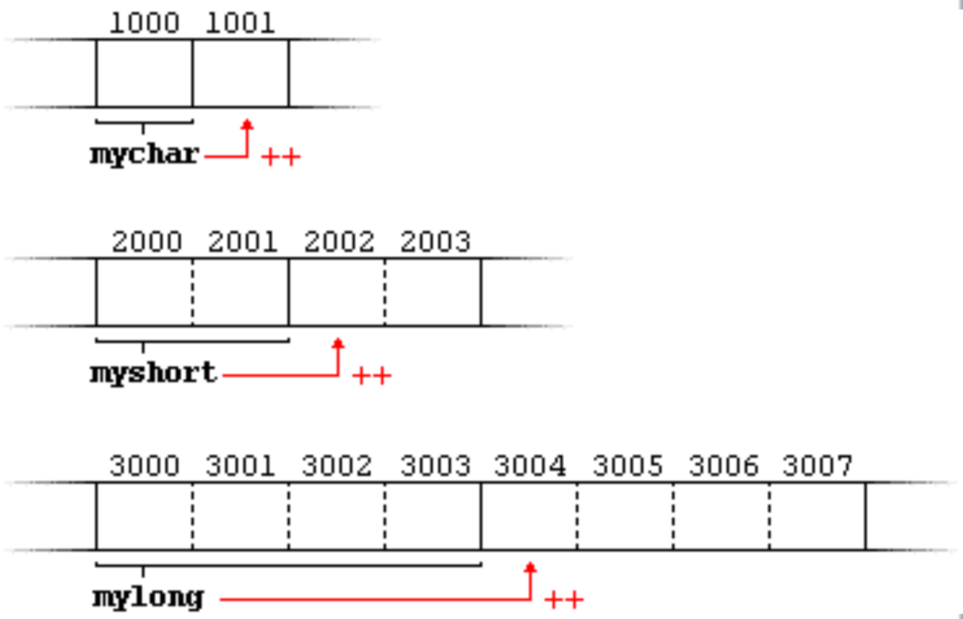
Pointer Arithmetic

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

char takes 1 byte, short takes 2 bytes, and long takes 4.

they point to the memory locations 1000, 2000, and 3000, respectively

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



Questions?

zahmaad.github.io