Lecture : Pointers

1 Background

1.1 Variables and Memory

When you declare a variable, the computer associates the variable name with a particular location in memory and stores a value there.

When you refer to the variable by name in your code, the computer must take two steps:

1. Look up the address that the variable name corresponds to
2. Go to that location in memory and retrieve or set the value it contains

C++ allows us to perform either one of these steps independently on a variable with the `&` and `*` operators:

1. `&x` evaluates to the address of `x` in memory.
2. `*( &x )` takes the address of `x` and *dereferences* it – it retrieves the value at that location in memory. `*( &x )` thus evaluates to the same thing as `x`.

1.2 Motivating Pointers

Memory addresses, or *pointers*, allow us to manipulate data much more flexibly; manipulating the memory addresses of data can be more efficient than manipulating the data itself. Just a taste of what we'll be able to do with pointers:

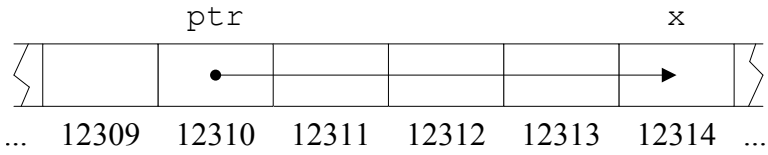
- More flexible pass-by-reference
- Manipulate complex data structures efficiently, even if their data is scattered in different memory locations
- Use polymorphism – calling functions on data without knowing exactly what kind of data it is (more on this in Lectures 7-8)

2 Pointers and their Behavior

2.1 The Nature of Pointers

Pointers are just variables storing integers – but those integers happen to be memory addresses, usually addresses of other variables. A pointer that stores the address of some variable `x` is said to *point to* `x`. We can access the value of `x` by dereferencing the pointer.

As with arrays, it is often helpful to visualize pointers by using a row of adjacent cells to represent memory locations, as below. Each cell represents 1 block of memory. The dot-arrow notation indicates that `ptr` "points to" `x` – that is, the value stored in `ptr` is 12314, `x`'s memory address.



2.2 Pointer Syntax/Usage

2.2.1 Declaring Pointers

To declare a pointer variable named `ptr` that points to an integer variable named `x`:

```
int *ptr = &x;
```

`int *ptr` declares the pointer to an integer value, which we are initializing to the address of `x`.

We can have pointers to values of any type. The general scheme for declaring pointers is:

```
data_type *pointer_name; // Add "= initial_value" if applicable
```

`pointer_name` is then a variable of type `data_type *` – a "pointer to a `data_type` value."

2.2.2 Using Pointer Values

Once a pointer is declared, we can dereference it with the `*` operator to access its value:

```
cout << *ptr; // Prints the value pointed to by ptr,
              // which in the above example would be x's value
```

We can use deferenced pointers as l-values:

```
*ptr = 5; // Sets the value of x
```

Without the * operator, the identifier x refers to the pointer itself, not the value it points to:

```
cout << ptr; // Outputs the memory address of x in base 16
```

Just like any other data type, we can pass pointers as arguments to functions. The same way we'd say `void func(int x) {...}`, we can say `void func(int *x){...}`. Here is an example of using pointers to square a number in a similar fashion to pass-by-reference:

```
1 void squareByPtr( int *numPtr ) {
2   *numPtr = *numPtr * *numPtr;
3 }
4
5 int main() {
6   int x = 5;
7   squareByPtr(&x);
8   cout << x; // Prints 25
9 }
```

Note the varied uses of the * operator on line 2.

2.2.3 `const` Pointers

There are two places the `const` keyword can be placed within a pointer variable declaration. This is because there are two different variables whose values you might want to forbid changing: the pointer itself and the value it points to.

```
const int *ptr;
```

declares a changeable pointer to a constant integer. The integer value cannot be changed through this pointer, but the pointer may be changed to point to a different constant integer.

```
int * const ptr;
```

declares a constant pointer to changeable integer data. The integer value can be changed through this pointer, but the pointer may not be changed to point to a different constant integer.

```
const int * const ptr;
```

forbids changing either the address `ptr` contains or the value it points to.

2.3 Null, Uninitialized, and Deallocated Pointers

Some pointers do not point to valid data; dereferencing such a pointer is a runtime error. Any pointer set to 0 is called a *null pointer*, and since there is no memory location 0, it is an invalid pointer. One should generally check whether a pointer is null before dereferencing it. Pointers are often set to 0 to signal that they are not currently valid.

Dereferencing pointers to data that has been erased from memory also usually causes runtime errors. Example:

```
1  int *myFunc() {
2    int phantom = 4;
3    return &phantom;
4  }
```

`phantom` is deallocated when `myFunc` exits, so the pointer the function returns is invalid.

As with any other variable, the value of a pointer is undefined until it is initialized, so it may be invalid.

3 References

When we write `void f(int &x) {...}` and call `f(y)`, the reference variable `x` becomes another name – an *alias* – for the value of `y` in memory. We can declare a reference variable locally, as well:

```
int y;
int &x = y; // Makes x a reference to, or alias of, y
```

After these declarations, changing `x` will change `y` and vice versa, because they are two names for the same thing.

References are just pointers that are dereferenced every time they are used. Just like pointers, you can pass them around, return them, set other references to them, etc. The only differences between using pointers and using references are:

- References are sort of pre-dereferenced – you do not dereference them explicitly.
- You cannot change the location to which a reference points, whereas you can change the location to which a pointer points. Because of this, references must always be initialized when they are declared.
- When writing the value that you want to make a reference to, you do not put an `&` before it to take its address, whereas you do need to do this for pointers.

3.1 The Many Faces of * and &

The usage of the * and & operators with pointers/references can be confusing. The * operator is used in two different ways:

1. When declaring a pointer, * is placed before the variable name to indicate that the variable being declared is a pointer – say, a pointer to an `int` or `char`, not an `int` or `char` value.
2. When using a pointer that has been set to point to some value, * is placed before the pointer name to dereference it – to access or set the value it points to.

A similar distinction exists for &, which can be used either

1. to indicate a reference data type (as in `int &x;`), or
2. to take the address of a variable (as in `int *ptr = &x;`).

4 Pointers and Arrays

The name of an array is actually a pointer to the first element in the array. Writing `myArray[3]` tells the compiler to return the element that is 3 away from the starting element of `myArray`.

This explains why arrays are always passed by reference: passing an array is really passing a pointer.

This also explains why array indices start at 0: the first element of an array is the element that is 0 away from the start of the array.

4.1 Pointer Arithmetic

Pointer arithmetic is a way of using subtraction and addition of pointers to move around between locations in memory, typically between array elements. Adding an integer `n` to a pointer produces a new pointer pointing to `n` positions further down in memory.

4.1.1 Pointer Step Size

Take the following code snippet:

```
1 long arr[] = {6, 0, 9, 6};
2 long *ptr = arr;
3 ptr++;
```

```
4 long *ptr2 = arr + 3;
```

When we add 1 to `ptr` in line 3, we don't just want to move to the next byte in memory, since each array element takes up multiple bytes; we want to move to the next element in the array. The C++ compiler automatically takes care of this, using the appropriate step size for adding to and subtracting from pointers. Thus, line 3 moves `ptr` to point to the second element of the array.

Similarly, we can add/subtract two pointers: `ptr2 - ptr` gives the number of array elements between `ptr2` and `ptr` (2). All addition and subtraction operations on pointers use the appropriate step size.

4.1.2 Array Access Notations

Because of the interchangeability of pointers and array names, *array-subscript notation* (the form `myArray[3]`) can be used with pointers as well as arrays. When used with pointers, it is referred to as *pointer-subscript notation*.

An alternative is *pointer-offset notation*, in which you explicitly add your offset to the pointer and dereference the resulting address. For instance, an alternate and functionally identical way to express `myArray[3]` is `*(myArray + 3)`.

4.2 `char *` Strings

You should now be able to see why the type of a string value is `char *`: a string is actually an array of characters. When you set a `char *` to a string, you are really setting a pointer to point to the first character in the array that holds the string.

You cannot modify string literals; to do so is either a syntax error or a runtime error, depending on how you try to do it. (String literals are loaded into read-only program memory at program startup.) You can, however, modify the contents of an array of characters. Consider the following example:

```
char courseName1[] = {'6', '.', '0', '9', '6', '\0' };
char *courseName2 = "6.096";
```

Attempting to modify one of the elements `courseName1` is permitted, but attempting to modify one of the characters in `courseName2` will generate a runtime error, causing the program to crash.