C++ Programming Bootcamp 2

COSC2802

Topic 7 Pointers I

Workshop Overview

1. Pointers, References, and all things confusing with C++
2. Pass by reference with pointers
3. Connection to Built-in Arrays

Pointer Basics

Pointer Basics

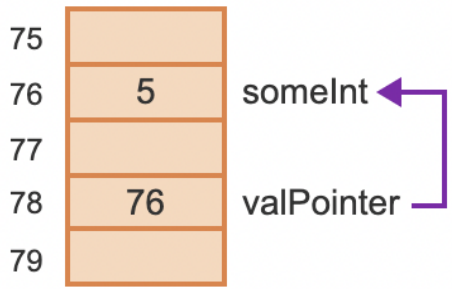
Normally, a variable directly contains a specific value:

pointer variables instead contain ***memory addresses.***

Example

```
int someInt = 5;
int* valPointer = &someInt; // more on & in a bit
```

These memory addresses tell you where the actual value is stored.



Source: zybooks

Why Pointers? (see Roberts 2.3)

1. Pointers allow you to refer to a large data structure in a compact way:
   - use the address as a shorthand for the complete value

2. Pointers allow you to reserve new memory during program execution
   - Dynamic Allocation

3. Pointers can be used to record relationships among data items:
   - Example linked data structures e.g.
     zybooks vector/array insert problem

More on this in Programming Studio 2

(i) Declaring Pointer Variables

Precede variable name with asterisk (*): the *pointer declaration operator*

Example declarations:

- *int_ptr* to be of the type *pointer-to-int* `int *int_ptr;`
- *char_ptr* to be of type *pointer-to-char* `char *char_ptr;`

Either are fine:

```
int* int_ptr;
int *int_ptr;
```

NOTE:

- *pointer-to-int* and *pointer-to-char* are **distinct types** even though both are **represented internally as an address** (so compiler knows how to interpret value at the address)
- The type of the value a pointer points is the **base type** of that pointer e.g.

  the type *pointer-to-int* has *int* as its base type.

lvalues (an aside)

lvalues can appear **on left hand side** of assignment e.g.

x = 1.0;

rvalue (rhs):
temporary values or literals that don't have memory address that can be referred to

Properties of lvalues:

- Every lvalue is stored somewhere in memory and has an **address.**
- Once declared, the ***address*** of an lvalue **never changes**
  - even though the **contents** of the lvalue may change
- Different lvalues require different amounts of memory, depends on type
- **Address** of an lvalue is a **pointer value** which can be
  - stored in memory
  - manipulated as data.

See: Roberts pg 52

(ii) Using Pointers (*)

Use:

- (*) the unary ***dereference*** operator (aka Pointer ***Indirection*** or ***value-at*** operator)
- obtains the **variable** to which its operand points
- operand of (*) operator must be of a ***pointer type***
- called dereferencing the pointer

Dereference Operator *

- Operator takes a value of any pointer type and returns the ***lvalue*** it points to
- operation produces an ***lvalue,*** so you can assign a value to a dereferenced pointer.

(ii) Using Pointers (*)

Use:

- (*) the unary ***dereference*** operator (aka Pointer ***Indirection*** or ***value-at*** operator)
- obtains the **variable** to which its operand points
- operand of (*) operator must be of a ***pointer type***
- called dereferencing the pointer

Dereference Operator *

- Operator takes a value of any pointer type and returns the ***lvalue*** it points to
- operation produces an ***lvalue,*** so you can assign a value to a dereferenced pointer.

```
int a = 7;
int* ptr = &a;
std::cout << a << std::endl;
std::cout << ptr << std::endl;
std::cout << *ptr << std::endl;

*ptr = 10;
std::cout << a << std::endl;
std::cout << *ptr << std::endl;
```

What will be printed?

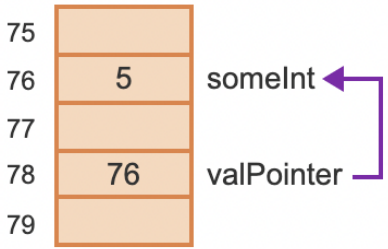
See: zybooks “Pointers Example”

(iii) Using Pointers (&) – Address of operator

Use:

- & operator (address-of) **returns the memory address** in which that lvalue is stored

Example:




```
int someInt = 5;
int *valPointer = &someInt;

std::cout << valPointer << " : " << *valPointer << std::endl;
std::cout << &someInt << " : " << someInt << std::endl;
```

```
$ ./ex
0x7fff94532430 : 5
0x7fff94532430 : 5
```

Reference Variables (&) – Reference Declarator

A reference variable is one that **refers to the address** of another variable

- an alias i.e. no new storage - it “refers” to the existing variable.

& can be used as a **reference declarator**

```
int target = 100;
int &rTarget = target;

std::cout << "Target: " << target << std::endl;
std::cout << "rTarget: " << target << std::endl;

rTarget = 200;

std::cout << "Target: " << target << std::endl;
std::cout << "rTarget: " << target << std::endl;
```

Here, *&rTarget* makes a new reference variable and points it to *target*. So the values are now linked.

- Must initialize each reference with an existing variable.

Main use of references:

- acting as **function formal parameters to support pass-by-reference**

Difference between & operator and * operator:

The & operator:

- **takes an lvalue** as its operand and **returns the memory address** in which that lvalue is stored.

The * operator:

- **takes a value of any pointer type** and **returns the lvalue** it points to
  - Called **dereferencing** the pointer
  - Produces an **lvalue**: means you can **assign a value to a dereferenced pointer**

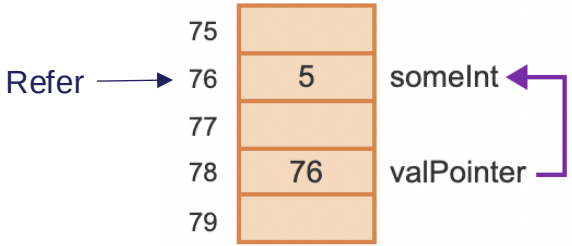
Difference between References and Pointers:

References are just aliases to already existing variables. So their value is in the same location as the original.

Whereas a pointer's value is the address of the value of another variable.

```
int someInt = 5;
int *valPointer = &someInt;
int &refer = someInt;

std::cout << valPointer << " : " << *valPointer << std::endl;
std::cout << &someInt << " : " << someInt << std::endl;
std::cout << &refer << " : " << refer << std::endl;
```

```
Output:
0x7fff4f56326c : 5
0x7fff4f56326c : 5
0x7fff4f56326c : 5
```

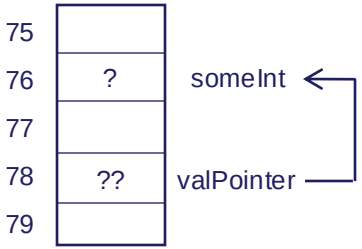


Summary

*	Pointer declaration	Declares a pointer variable	`int* ptr = &x;`	ptr is a pointer to an integer, storing the memory address of x.
*	Dereferencing a pointer	Accesses or modifies the value at the memory address	`*ptr = 20;`	Here it dereferences ptr and assigns the value 20 to the memory location it points to
&	Address-of operator	Retrieves the memory address of a variable	`int* ptr = &x;`	&x returns the memory address of x. This is assigned to ptr
&	Reference declarator (in declaration)	Declares a reference variable	`int& ref = x;`	ref becomes an alias for x. It refers to (a link) to the same memory location as x

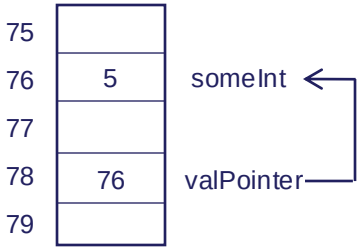
Example: (also see zybooks: Pointers Example)

```
int main()
{
    // (1) declare someInt of type int and valPointer as type pointer-to-int
    int someInt;
    int *valPointer;
    someInt = 5;

    //  (2) & operator: takes lvalue as operand; returns memory address where lvalue is stored
    std::cout << "someInt address is " << &someInt << "\n";
    valPointer = &someInt;
    std::cout << "valPointer is " << valPointer << "\n";
```



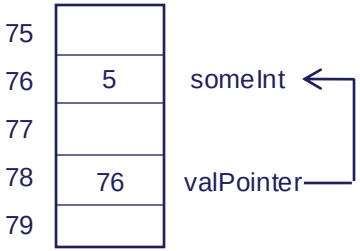
Example:

```
int main()
{
    // (1) declare someInt of type int and valPointer as type pointer-to-int
    int someInt;
    int *valPointer;
    someInt = 5;

    //  (2) & operator: takes lvalue as operand; returns memory address where lvalue is stored
    std::cout << "someInt address is " << &someInt << "\n";
    valPointer = &someInt;
    std::cout << "valPointer is " << valPointer << "\n";
```



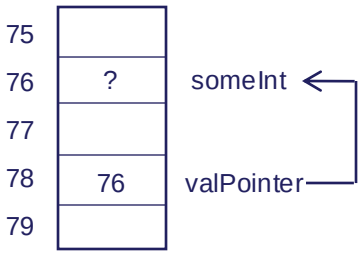
Example:

```
int main()
{
    // (1) declare someInt of type int and valPointer as type pointer-to-int
    int someInt;
    int *valPointer;
    someInt = 5;

    //  (2) & operator: takes lvalue as operand; returns memory address where lvalue is stored
    std::cout << "someInt address is " << &someInt << "\n";
    valPointer = &someInt;
    std::cout << "valPointer is " << valPointer << "\n";

    //  (3) * operator: takes a value of any pointer type and returns the lvalue it points to
    std::cout << "valPointer dereferenced value is " << *valPointer << "\n";
```



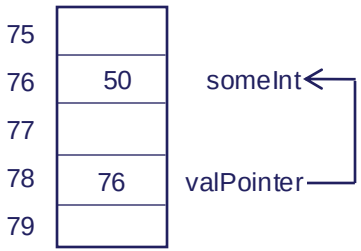
Example:

```
int main()
{
    // (1) declare someInt of type int and valPointer as type pointer-to-int
    int someInt;
    int *valPointer;
    someInt = 5;

    //  (2) & operator: takes lvalue as operand; returns memory address where lvalue is stored
    std::cout << "someInt address is " << &someInt << "\n";
    valPointer = &someInt;
    std::cout << "valPointer is " << valPointer << "\n";

    //  (3) * operator: takes a value of any pointer type and returns the lvalue it points to
    std::cout << "valPointer dereferenced value is " << *valPointer << "\n";

    //  (4) can assign a value to a dereferenced pointer
    *valPointer *= 10;
    std::cout << "valPointer dereferenced value is " << *valPointer << "\n";
```



Example:

```
int main()
{
    // (1) declare someInt of type int and valPointer as type pointer-to-int
    int someInt;
    int *valPointer;
    someInt = 5;

    //  (2) & operator: takes lvalue as operand; returns memory address where lvalue is stored
    std::cout << "someInt address is " << &someInt << "\n";
    valPointer = &someInt;
    std::cout << "valPointer is " << valPointer << "\n";

    //  (3) * operator: takes a value of any pointer type and returns the lvalue it points to
    std::cout << "valPointer dereferenced value is " << *valPointer << "\n";

    //  (4) can assign a value to a dereferenced pointer
    *valPointer *= 10;
    std::cout << "valPointer dereferenced value is " << *valPointer << "\n";
```



Common Syntax Errors (zybooks)

(1) using the dereference operator when initializing a pointer. Example:

```
int maxValue;
int* valPointer;
*valPointer = &maxValue; // syntax error - WHY?
```

because **valPointer* is referring to *the value pointed to*, *not the pointer itself*

(2) declaring multiple pointers on the same line and forget the * before each pointer name. Example:

```
int* valPointer1, valPointer2;
```

declares *valPointer1* as a pointer, but *valPointer2* is declared as an integer because no * exists before valPointer2.

Good practice: declare one pointer per line to avoid this

Common Runtime Errors (zybooks)

1) using the dereference operator when a pointer has not been initialized. Ex:
*cout << *valPointer;*

may cause a program to crash if *valPointer* holds an unknown address or an address the program is not allowed to access.

2) dereferencing a null pointer. Example: If valPointer is null, then
*cout << *valPointer;*

causes the program to crash.

Good practice:

- Always initialise pointers - *a valid address* or *nullptr*
- Check for *nullptr* ***before*** *dereferencing*

SEGMENTATION FAULTS: are almost always caused by bad pointer usage or arrays (accessing restricted/invalid mem location). If you get one, check your pointers! **Zybooks calls these SIGSEGV–11**

Null Pointer

When a pointer is declared, the pointer variable holds an **unknown address** until the pointer is initialized.

Can indicate that a pointer points to Null (ie nothing) by initializing a pointer to null. A pointer that is assigned with the keyword *nullptr* is said to be null. Example:

*int *maxValPointer = nullptr;*

makes maxValPointer null.

Pass by pointer

Summary: Passing arguments to functions

i. Pass by Value:

- function gets copy of the argument value. NO CHANGES to the original argument.
- Use? Modification isn't needed

ii. Pass by Reference:

- function gets reference to argument. CHANGES affect the original argument
- Use? Want to change original; avoid overhead of copying if large

iii. Pass by Pointer:

- Pass an **address.** In the function, dereference to access/change
- Use? Legacy code, working with pointers, dynamic memory, …

iv. …

Pass by Reference:

✅ Good for performance reasons: can eliminate overhead of copying lots of data.

⛔ Can weaken security: the called function can corrupt the caller's data

Pass by Pointer

Use pointers and the indirection operator (*) ie dereference

- pass the **address** of the variable to the function
- use the **dereferenced pointer** in the function

```
#include <iostream>

void cubeByReference(int *); // prototype (no need for parameter - ignored by compiler)

int main()
{
    int number{5}; // uniform initialization
    std::cout << "The original value of number is " << number;
    cubeByReference(&number); // pass number address to cubeByReference
    std::cout << "\nThe new value of number is " << number << std::endl;
}

// calculate cube of *nPtr; modifies variable number in main
void cubeByReference(int *nPtr)
{
    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
}
```

See: https://en.cppreference.com/w/cpp/language/operator_precedence

Pass by Reference vs Pass by Pointer

In general, **prefer pass by reference:**

- Cleaner and more readable syntax (e.g. work directly with variable name in function)
- Null Safety as references can't be null (removes need to explicitly check in function)
- Use of const gives efficiency games and guarantees original remains unchanged
- …

Specific cases require use of pass by pointer. For example:

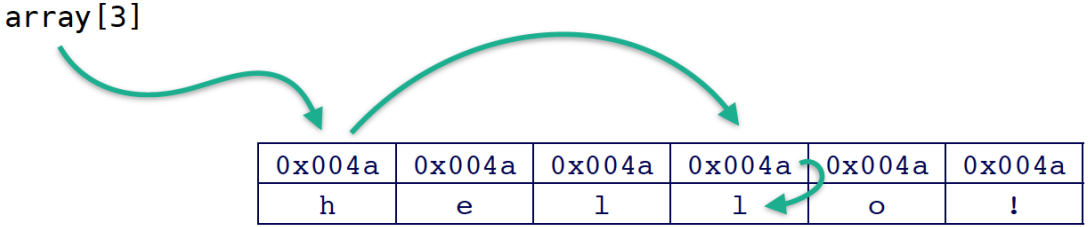
- Working with C code or libraries
- Nullable Arguments: have option to pass null pointer e.g. when you want to pass an argument that may or may not exist.
- …

Relationship | Pointers and Built-in Arrays

- An array is actually an ***abstraction for using pointers***!
- The actual array "variable" is a pointer to the first element of the array

  i.e. Arrays can decay (type of implicit conversion) into pointers to first element

- The square-bracket lookup notation is short-hand for:
  - Go to the memory location of the array
  - Go to the 'ith' memory location from this
  - Dereference that memory address



Built-in Arrays and Functions

Passing built-in array (1D) to functions:

- Can pass 1D array **directly** to functions without explicitly using pointers:
  - *arrayName* decays (converted) to *&arrayName[0]* i.e. pointer to first element
  - No need to use address-of (&) with built-in array – just array name

Declaring built-in array parameters:

Can use [] notation or pointer notation – compiler doesn't differentiate:

```
void modifyArray(int arr[], const int size)
void modifyArray(int* arr, const int size)
```

- [] notation preferred for clarity
- Note: *size* parameter needs to be passed to the function:
  - Why? decay to pointer means *sizeof* gives size of pointer not the array (see example and https://en.cppreference.com/w/cpp/language/sizeof)

Example: Passing Built-in Array to Function

see:

1. Example: Passing built-in array to function (zybooks)
2. Example: Built-In array: Allocated and Logical Length (zybooks)