Pointers and Dynamic Arrays
Chapter 9
Objectives

In this chapter, you will learn about:

- Pointers
- Dynamic Arrays
**Pointers**

**What is a Pointer?**

- **A pointer is the memory address of a variable**
- Recall, Computer memory is divided into cells (or **bytes**)
  - Each byte has a unique number called **address**
- An ordinary variable is stored in consecutive bytes
  - The number of bytes depends on the variable's type
- The variable's **address** is the address of its **first** byte

Recall, when we use a variable as a call-by-reference in a function call, we pass the **address** of the variable
Recall, the address identifies the location where the variable is stored in memory.

Pointers identify a variable via their address (not by the name)

- Pointers "point" to a variable by telling where the variable is located
  - Some Pointers will point to Integer Variables
  - Others will point to characters, double, long, integer, etc.

Note: A Pointer is an address; an Address is an integer, but a Pointer is not an integer!

So how do you declare a pointer variable?
Let’s Declare a Pointer variable \( p \) that “points” to a variable of type \( \text{double} \):

\[
\text{double } \ast \text{p;}
\]

- The asterisk identifies \( p \) as a pointer variable
- \( p \) holds pointers to variables of type \( \text{double} \)

\[
\text{int } \ast \text{p, } \ast \text{q; // declares variables p and q}
\]

\[
\text{ // to hold pointers of type int}
\]

- There must be an asterisk before each of the pointer variable

**Pointer Variables must be Declared to have a Pointer Type**
Let’s Declare Pointer variables \texttt{p1}, \texttt{p2}; and ordinary variables \texttt{r}, \texttt{s} each of type integer:

\begin{verbatim}
int *p1, *p2, r, s;
\end{verbatim}
Pointers
Address Assignment

- How does a Pointer point to a variable?
  - You assign a variable address to the pointer

Example

```c
int *p, v; // Declares p as a pointer to an integer variable
            // Declares v as an ordinary integer variable
p = &v;    // The address of v is assigned to p
```

The operator & in front of an ordinary variable produces the address of that variable.
After assigning a variable address to a pointer: \( p = \&v1; \)
you can refer to the variable (v) in two ways:

**Variable name v:**

**The variable pointed to by p (\( *p \))**

Example:

```c
int *p1, v1;
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

Displays:

```
42
42
```

The operator \( * \) in front of a pointer variable produces the value it points to
How do you assign a pointer to another pointer?

Example:

```c
int *p1, v1;
v1 = 0;
p1 = &v1;
p2 = p1

*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
cout << *p2 << endl;
```

p1 points to v1
p2 points to p1, which points to v1
Hence p2 points to v1
Therefore p1, p2 and v1 refer to variable v1
What is the difference between the following Assignment Statements?

- `p1 = p3;`
- `*p1 = *p3;`
Pointers
Basic Manipulations

Assignment Operator

Let’s analyze the code segment:

```c
int *p1, *p2, r, s;
r = 84;
s = 99;
p1 = &r;
p2 = &s;

p1 = p2;
*p1 = *p2
```
Pointers
The new Operator

- Pointers manipulate variables
- How do you create a pointer to manipulate a new "nameless" variable of type int?
  \[
  \text{p1 = new int;}
  \]
- The new variable is referred to as *p1
- *p1 can be used anywhere an integer variable can
  \[
  \text{cin >> *p1;}
  \]
  \[
  \text{*p1 = *p1 + 7;}
  \]
The **new** Operator

Let’s analyze the following code:

```cpp
int main ( )
{
    int *p1, *p2;
    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 = =" << *p1 << endl;
    cout << "*p2 = =" << *p2 << endl;

    *p1 = 53;
    cout << "*p1 = =" << *p1 << endl;
    cout << "*p2 = =" << *p2 << endl;

    *p1 = new int;
    *p1 = 88;
    cout << "*p1 = =" << *p1 << endl;
    cout << "*p2 = =" << *p2 << endl;
    return 0;
}
```
Variables created using the `new` operator are called dynamic variables.

- Dynamic variables are created and destroyed while the program is running.
An area of memory called the freestore is reserved for dynamic variables
- New dynamic variables use memory in the freestore
- If all of the freestore is used, calls to \texttt{new} will fail

Unneeded memory can be recycled
- When variables are no longer needed, they can be deleted and the memory they used is returned to the freestore
Delete Operator

- When dynamic variables are no longer needed, delete them to return memory to the freestore

  Example:

  ```c
  delete p;
  ```

  The value of p is now undefined and the memory used by the variable that p pointed to is back in the freestore
Dangling Pointers

- Using delete on a pointer variable destroys the dynamic variable pointed to.
- If another pointer variable was pointing to the dynamic variable, that variable is also undefined.
- Undefined pointer variables are called dangling pointers.
  - Dereferencing a dangling pointer (*p) is usually disastrous.
Variables declared in a function are created by C++ and destroyed when the function ends.
- These are called automatic variables because their creation and destruction is controlled automatically.

The programmer manually controls creation and destruction of pointer variables with operators new and delete.
Global Variables

- Variables declared outside any function definition are global variables
  - Global variables are available to all parts of a program
  - Global variables are not generally used
Typedef

- A name can be assigned to a type definition, then used to declare variables
- The keyword typedef is used to define new type names
- Syntax:
  ```
  typedef Known_Type_Definition New_Type_Name;
  ```
  - Known_Type_Definition can be any type
To avoid mistakes using pointers, define a pointer type name

- Example: `typedef int* IntPtr;`

- Hence,
  - `IntPtr p;` is equivalent to `int *p;`
Dynamic Arrays

- A Dynamic Array is an array whose size is determined when the program is running.
  - The size is not specified during declaration as for ordinary arrays
## Dynamic Arrays

### Advantages

<table>
<thead>
<tr>
<th>Normal Arrays</th>
<th>Dynamic Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>If programmer estimates very large array size:</td>
<td>N/A</td>
</tr>
<tr>
<td>- Memory is wasted</td>
<td>N/A</td>
</tr>
<tr>
<td>If programmer estimates very small size:</td>
<td>N/A</td>
</tr>
<tr>
<td>- The program may not work in some scenarios</td>
<td>Dynamic arrays can be created with just the right size while the program is running</td>
</tr>
</tbody>
</table>
Array variable is a pointer variable that points to the first indexed variable of the array:

- `int a[8];`
- `typedef int* IntPtr;`
- `IntPtr p; // p is pointer int variable`
- `// Note: a and p are of type int`
- `// Hence:`
- `p = a //Points p to the address of a`
- `// What is the address of a? a[0]`
- Therefore `p` points to `a[0]`

Recall

- int a[8];
- typedef int* IntPtr;
- IntPtr p; // p is pointer int variable
- // Note: a and p are of type int
- // Hence:
- p = a //Points p to the address of a
- // What is the address of a? a[0]
- Therefore p points to a[0]

- p is a pointer integer variable
  - Hence we can point p to other integer array addresses, b[0]

But

Can we point the array a to another (pointer) address?

suppose:
IntPtr p2;
a = p2 ← This is an illegal statement why?
**Dynamic Arrays**

Creating Dynamic Arrays

- **Step #1**: Define a pointer of the same *type* as array elements. Example:
  - `typedef double* DoubleArrayPtr;`

- **Step #2**: Declare pointer variable for the type:
  - `DoubleArrayPtr a;`

- **Step #3**: Create the dynamic array using the *new* operator
  - `a = new double[array_size];`

*array size*: int expression, int variable
Dynamic Arrays
Releasing Storage

- Recall, Dynamic variables, including **dynamic arrays**, are held in freestore memory
- Return dynamic arrays to freestore memory, when they are no longer referenced in the program:
  - **Syntax**: `delete [ ] a;`

What will likely happen if you omit the `[ ]`?
Pointer Arithmetic

Arithmetic of Addresses

- Concerned with Increment/Decrement of Memory Addresses stored in pointers
- consider:
  - typedef double* DoublePtr;
  - DoublePtr d;
  - d = new double[8];

- Recall array d points to d[0]
- The expression d + 1 evaluates to address of d[1]
- The expression d + 2 evaluates to address of d[2]
- So what is the meaning of d + 1?
**Pointer Arithmetic**

**Arithmetic Operations**

- Pointers of the same type (associated with a unique array) can be subtracted to give an intermediate array element address.
- Consider:
  - `typedef double* DoublePtr;`
  - `DoublePtr d;`
  - `d = new double[8];`
  - `for (int i = 0; i < array_size; i++)
      cout << *(d + i) << "  " ;`

Can you write using normal arrays?

Can you explain the following expressions?

- `cout << d++;`
- `cout << d--;`