Chapter 9:

Pointers
Pointer Variables

• **Pointer variable**: Often just called a pointer, it's a variable that holds an address

• Because a pointer variable holds the address of another piece of data, it "points" to the data
Something Like Pointers: Arrays

• We have already worked with something similar to pointers, when we learned to pass arrays as arguments to functions.

• For example, suppose we use this statement to pass the array numbers to the showValues function:

  showValues(numbers, SIZE);
The `values` parameter, in the `showValues` function, points to the `numbers` array.

C++ automatically stores the address of `numbers` in the `values` parameter.

```cpp
void showValues(int values[], int size)
{
    for (int count = 0; count < size; count++)
        cout << values[count] << endl;
}
```
Pointer Variables

- Pointer variables are yet another way using a memory address to work with a piece of data.

- Pointers are more "low-level" than arrays and reference variables.

- This means you are responsible for finding the address you want to store in the pointer and correctly using it.
Pointer Variables

• Definition:
  \[ \text{int * intptr;} \]

• Read as:
  "\text{intptr can hold the address of an int}"

• Spacing in definition does not matter:
  \[ \text{int * intptr; // same as above} \]
  \[ \text{int* intptr; // same as above} \]
Pointer Variables

• Assigning an address to a pointer variable:
  ```c
  int *intptr;
  intptr = &num;
  ```

• Memory layout:

  - `num` 25
  - `intptr` 0x4a00

  address of `num`: 0x4a00
The Indirection Operator

• The indirection operator (*) dereferences a pointer.
• It allows you to access the item that the pointer points to.

```cpp
int x = 25;
int *intptr = &x;
cout << *intptr << endl;
```

This prints 25.
The Relationship Between Arrays and Pointers

• Array name is starting address of array

```cpp
int vals[] = {4, 7, 11};
```

```
4 7 11
```

starting address of vals: 0x4a00

```cpp
cout << vals; // displays
// 0x4a00

cout << vals[0]; // displays 4
```
The Relationship Between Arrays and Pointers

• Array name can be used as a pointer constant:

```cpp
int vals[] = {4, 7, 11};
cout << *vals;    // displays 4
```

• Pointer can be used as an array name:

```cpp
int *valpPtr = vals;
cout << valpPtr[1]; // displays 7
```
Pointers in Expressions

Given:

```c
int vals[]={4,7,11}, *valpdr;
valpdr = vals;
```

What is `valpdr + 1`?

It means *(address in valpdr) + (1 * size of an int)*

```c
cout << *(valpdr+1);  //displays 7
cout << *(valpdr+2);  //displays 11
```

Must use ( ) as shown in the expressions
Array Access

- Array elements can be accessed in many ways:

<table>
<thead>
<tr>
<th>Array access method</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>array name and []</td>
<td>vals[2] = 17;</td>
</tr>
<tr>
<td>pointer to array and []</td>
<td>valptr[2] = 17;</td>
</tr>
<tr>
<td>array name and subscript arithmetic</td>
<td>*(vals + 2) = 17;</td>
</tr>
<tr>
<td>pointer to array and subscript arithmetic</td>
<td>*(valptr + 2) = 17;</td>
</tr>
</tbody>
</table>
Array Access

• Conversion:

\[ \text{vals}[i] \text{ is equivalent to } *(\text{vals} + i) \]

• No bounds checking performed on array access, whether using array name or a pointer
# Pointer Arithmetic

- Operations on pointer variables:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>++, --</td>
<td>valptr++;</td>
</tr>
<tr>
<td></td>
<td>valptr--;</td>
</tr>
<tr>
<td>+, - (pointer and int)</td>
<td>cout &lt;&lt; *(valptr + 2);</td>
</tr>
<tr>
<td>+=, -= (pointer and int)</td>
<td>valptr = vals;</td>
</tr>
<tr>
<td></td>
<td>valptr += 2;</td>
</tr>
<tr>
<td>- (pointer from pointer)</td>
<td>cout &lt;&lt; valptr-val;</td>
</tr>
</tbody>
</table>

```
int vals[]={4,7,11};
int *valptr = vals;
```

```
valptr++;       // points at 7
valptr--;       // now points at 4
```

```
cout << *(valptr + 2); // 11
```

```
valptr = vals;   // points at 4
valptr += 2;     // points at 11
```

```
cout << valptr-val; // difference
// (number of ints) between valptr
// and val
```
Initializing Pointers

- Can initialize at definition time:
  ```c
  int num, *numptr = &num;
  int val[3], *valptr = val;
  ```
- Cannot mix data types:
  ```c
  double cost;
  int *ptr = &cost; // won't work
  ```
- Can test for an invalid address for `ptr` with:
  ```c
  if (!ptr) ...
  ```
Comparing Pointers

• Relational operators (<, >=, etc.) can be used to compare addresses in pointers

• Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

  if (ptr1 == ptr2)  // compares addresses
  if (*ptr1 == *ptr2) // compares contents
Pointers as Function Parameters

- A pointer can be a parameter
- Works like reference variable to allow change to argument from within function
- Requires:
  1) asterisk * on parameter in prototype and heading

```c
void getNum(int *ptr); // ptr is pointer to an int
```

  2) asterisk * in body to dereference the pointer

```c
cin >> *ptr;
```

  3) address as argument to the function

```c
genum(&num); // pass address of num to getNum
```
Example

```c
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

int num1 = 2, num2 = -3;
swap(&num1, &num2);
```
Pointers to Constants

• Example: Suppose we have the following definitions:

```c
const int SIZE = 6;
const double payRates[SIZE] =
    { 18.55, 17.45, 12.85,
        14.97, 10.35, 18.89 };
```

• In this code, `payRates` is an array of constant doubles.
Pointers to Constants

• Suppose we wish to pass the `payRates` array to a function? Here's an example of how we can do it.

```c
void displayPayRates(const double *rates, int size)
{
    for (int count = 0; count < size; count++)
    {
        cout << "Pay rate for employee " << (count + 1) << " is $" << *(rates + count) << endl;
    }
}
```

The parameter, `rates`, is a pointer to `const double`. 
Declaration of a Pointer to Constant

The asterisk indicates that rates is a pointer.

const double *rates

This is what rates points to.
Constant Pointers

• A constant pointer is a pointer that is initialized with an address, and cannot point to anything else.

• Example

```c
int value = 22;
int * const ptr = &value;
```
Constant Pointers

* const indicates that ptr is a constant pointer.

int * const ptr

This is what ptr points to.
Constant Pointers to Constants

* const indicates that ptr is a constant pointer.

const int * const ptr

This is what ptr points to.
Dynamic Memory Allocation: `new`

- Can allocate storage for a variable while program is running
- Computer returns address of newly allocated variable
- **Uses** `new` **operator to allocate memory:**
  ```
  double *dptr;
  dptr = new double;
  ```
- **`new` returns address of memory location**
Dynamic Memory Allocation

• Can also use `new` to allocate array:
  ```
  const int SIZE = 25;
  arrayPtr = new double[SIZE];
  ```

• Can then use `[]` or pointer arithmetic to access array:
  ```
  for(i = 0; i < SIZE; i++)
    *arrayptr[i] = i * i;
  ```
  or
  ```
  for(i = 0; i < SIZE; i++)
    *(arrayptr + i) = i * i;
  ```

• Program will terminate if not enough memory available to allocate
Releasing Dynamic Memory

- **Use** `delete` **to free dynamic memory:**
  
  ```
  delete fptr;
  ```
- **Use** `[]` **to free dynamic array:**
  
  ```
  delete [] arrayptr;
  ```
- **Only use** `delete` **with dynamic memory!**
Program 9-14

// This program totals and averages the sales figures for any
// number of days. The figures are stored in a dynamically
// allocated array.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double *sales, // To dynamically allocate an array
    total = 0.0, // Accumulator
    average;   // To hold average sales
Program 9-14  (continued)

```cpp
13    int numDays,    // To hold the number of days of sales
14        count;     // Counter variable
15
16    // Get the number of days of sales.
17    cout << "How many days of sales figures do you wish ";
18    cout << "to process? ";
19    cin >> numDays;
20
21    // Dynamically allocate an array large enough to hold
22    // that many days of sales amounts.
23    sales = new double[numDays];
24
25    // Get the sales figures for each day.
26    cout << "Enter the sales figures below.\n";
27    for (count = 0; count < numDays; count++)
28    {
29        cout << "Day " << (count + 1) << ": ";
30        cin >> sales[count];
31    }
32```

Returning Pointers from Functions

• Pointer can be the return type of a function:
  
  int* newNum();

• The function must not return a pointer to a local variable in the function.

• A function should only return a pointer:
  – to data that was passed to the function as an argument, or
  – to dynamically allocated memory
int *getRandomNumbers(int num)
{
    int *array;  // Array to hold the numbers
    // Return null if num is zero or negative.
    if (num <= 0)
        return NULL;
    // Dynamically allocate the array.
    array = new int[num];
    // Seed the random number generator by passing
    // the return value of time(0) to srand.
    srand( time(0) );
    // Populate the array with random numbers.
    for (int count = 0; count < num; count++)
        array[count] = rand();
    // Return a pointer to the array.
    return array;
}