Chapter 9: Pointers
9.1

Getting the Address of a Variable
Getting the Address of a Variable

• Each variable in program is stored at a unique address
• Use address operator `&` to get address of a variable:

```cpp
int num = -99;
cout << &num; // prints address
    // in hexadecimal
```
9.2

Pointer Variables
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 \textit{numbers} to the \textit{showValues} function:

\begin{verbatim}
showValues(numbers, SIZE);
\end{verbatim}
Something Like Pointers: Arrays

The `values` parameter, in the `showValues` function, points to the `numbers` array.

C++ automatically stores the address of `numbers` in the `values` parameter.
Something Like Pointers: Reference Variables

• We have also worked with something like pointers when we learned to use reference variables. Suppose we have this function:

```cpp
void getOrder(int &donuts) {
    cout << "How many doughnuts do you want? ";
    cin >> donuts;
}
```

• And we call it with this code:
```cpp
int jellyDonuts;
getOrder(jellyDonuts);
```
Something Like Pointers: Reference Variables

The `donuts` parameter, in the `getOrder` function, points to the `jellyDonuts` variable.

C++ automatically stores the address of `jellyDonuts` in the `donuts` parameter.

```cpp
void getOrder(int &donuts) {
    cout << "How many doughnuts do you want? ";
    cin >> donuts;
}
```
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:

```c
int * intptr;
```

• Read as:

“`intptr` can hold the address of an int”

• Spacing in definition does not matter:

```c
int * intptr; // same as above
int* intptr;  // same as above
```
Pointer Variables

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

• Memory layout:

  ![Memory Layout Diagram]

  address of num: 0x4a00
Program 9-2

1 // This program stores the address of a variable in a pointer.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int x = 25;   // int variable
8     int *ptr;   // Pointer variable, can point to an int
9
10    ptr = &x;   // Store the address of x in ptr
11    cout << "The value in x is " << x << endl;
12    cout << "The address of x is " << ptr << endl;
13    return 0;
14 }

Program Output
The value in x is 25
The address of x is 0x7e00
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.
Program 9-3

// This program demonstrates the use of the indirection operator.
#include <iostream>
using namespace std;

int main()
{
    int x = 25;       // int variable
    int *ptr;        // Pointer variable, can point to an int

    ptr = &x;         // Store the address of x in ptr

    // Use both x and ptr to display the value in x.
    cout << "Here is the value in x, printed twice:\n";
    cout << x << endl;   // Displays the contents of x
    cout << *ptr << endl; // Displays the contents of x

    // Assign 100 to the location pointed to by ptr. This
    // will actually assign 100 to x.
    *ptr = 100;

    // Use both x and ptr to display the value in x.
    cout << "Once again, here is the value in x:\n";
    cout << x << endl;   // Displays the contents of x
    cout << *ptr << endl; // Displays the contents of x
    return 0;
}
Program 9-3 (continued)

Program Output
Here is the value in x, printed twice:
25
25
Once again, here is the value in x:
100
100
9.3

The Relationship Between Arrays and Pointers
The Relationship Between Arrays and Pointers

- Array name is starting address of array

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

starting address of `vals`: 0x4a00

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

```cpp
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 *valp.ptr = vals;
cout << valp.ptr[1]; // displays 7
```
Program 9-5

// This program shows an array name being dereferenced with the * operator.
#include <iostream>
using namespace std;

int main()
{
    short numbers[] = {10, 20, 30, 40, 50};

    cout << "The first element of the array is ";
    cout << *numbers << endl;
    return 0;
}

Program Output
The first element of the array is 10
Pointers in Expressions

Given:

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

What is `valpPtr + 1`? It means (address in `valpPtr`) + (1 * size of an int)

```c
cout << *(valpPtr+1); //displays 7
cout << *(valpPtr+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:** $vals[i]$ is equivalent to $*(vals + i)$

- No bounds checking performed on array access, whether using array name or a pointer
From Program 9-7

```cpp
9  const int NUM_COINS = 5;
10  double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
11  double *doublePtr;  // Pointer to a double
12  int count;          // Array index
13
14  // Assign the address of the coins array to doublePtr.
15  doublePtr = coins;
16
17  // Display the contents of the coins array. Use subscripts
18  // with the pointer!
19  cout << "Here are the values in the coins array:\n";
20  for (count = 0; count < NUM_COINS; count++)
21      cout << doublePtr[count] << " ";
22
23  // Display the contents of the array again, but this time
24  // use pointer notation with the array name!
25  cout << "\nAnd here they are again:\n";
26  for (count = 0; count < NUM_COINS; count++)
27      cout << *(coins + count) << " ";
28  cout << endl;
```

**Program Output**

Here are the values in the coins array:
0.05 0.1 0.25 0.5 1
And here they are again:
0.05 0.1 0.25 0.5 1
9.4 Pointer Arithmetic
### Pointer Arithmetic

- **Operations on pointer variables:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int vals[]={4,7,11}; int *valptr = vals;</td>
</tr>
<tr>
<td>++, --</td>
<td>valptr++; // points at 7</td>
</tr>
<tr>
<td></td>
<td>valptr--; // now points at 4</td>
</tr>
<tr>
<td>+, - (pointer and int)</td>
<td>cout &lt;&lt; *(valptr + 2); // 11</td>
</tr>
<tr>
<td>+=, -= (pointer and int)</td>
<td>valptr = vals; // points at 4</td>
</tr>
<tr>
<td></td>
<td>valptr += 2; // points at 11</td>
</tr>
<tr>
<td>- (pointer from pointer)</td>
<td>cout &lt;&lt; valptr-val; // difference</td>
</tr>
<tr>
<td></td>
<td>//(number of ints) between valptr</td>
</tr>
<tr>
<td></td>
<td>// and val</td>
</tr>
</tbody>
</table>
From Program 9-9

```c
const int SIZE = 8;
int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
int *numPtr;    // Pointer
int count;      // Counter variable for loops

// Make numPtr point to the set array.
numPtr = set;

// Use the pointer to display the array contents.
cout << "The numbers in set are:\n";
for (count = 0; count < SIZE; count++)
{
    cout << *numPtr << " ";
    numPtr++;
}

// Display the array contents in reverse order.
cout << "\nThe numbers in set backward are:\n";
for (count = 0; count < SIZE; count++)
{
    numPtr--;
    cout << *numPtr << " ";
}
```

**Program Output**
The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backward are:
40 35 30 25 20 15 10 5
9.5 Initializing Pointers
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
Comparing Pointers

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

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

  \[
  \begin{align*}
  \text{if } (\text{ptr1} & \text{ == ptr2}) & \quad & \text{// compares addresses} \\
  \text{if } (*\text{ptr1} & \text{ == *ptr2}) & \quad & \text{// compares contents}
  \end{align*}
  \]
9.7

Pointers as Function Parameters
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
     ```cpp
     void getNum(int *ptr); // ptr is pointer to an int
     ```
  2) asterisk * in body to dereference the pointer
     ```cpp
     cin >> *ptr;
     ```
  3) address as argument to the function
     ```cpp
     getNum(&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);
```
Program 9-11

1  // This program uses two functions that accept addresses of
2  // variables as arguments.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototypes
7  void getNumber(int *);
8  void doubleValue(int *);
9
10 int main()
11 {
12   int number;
13
14   // Call getNumber and pass the address of number.
15   getNumber(&number);
16
17   // Call doubleValue and pass the address of number.
18   doubleValue(&number);
19
20   // Display the value in number.
21   cout << "That value doubled is " << number << endl;
22   return 0;
23 }
24
(Program Continues)
void getNumber(int *input)
{
  cout << "Enter an integer number: ";
  cin >> *input;
}

void doubleValue(int *val)
{
  *val *= 2;
}

Program Output with Example Input Shown in Bold
Enter an integer number: 10 [Enter]
That value doubled is 20
Pointers to Constants

• If we want to store the address of a constant in a pointer, then we need to store it in a pointer-to-const.
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.

```cpp
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

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

* const indicates that ptr is a constant pointer.

This is what ptr points to.
Constant Pointers to Constants

• A constant pointer to a constant is:
  – a pointer that points to a constant
  – a pointer that cannot point to anything except what it is pointing to

• Example:
  ```
  int value = 22;
  const int * const ptr = &value;
  ```
Constant Pointers to Constants

* const indicates that ptr is a constant pointer.

const int * const ptr

This is what ptr points to.
9.8

Dynamic Memory Allocation
Dynamic Memory Allocation

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

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

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

1 // This program totals and averages the sales figures for any
2 // number of days. The figures are stored in a dynamically
3 // allocated array.
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
8 int main()
9 {
10    double *sales, // To dynamically allocate an array
11        total = 0.0, // Accumulator
12        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```

// Calculate the total sales
for (count = 0; count < numDays; count++)
{
    total += sales[count];
}

// Calculate the average sales per day
average = total / numDays;

// Display the results
cout << fixed << showpoint << setprecision(2);
cout << "\n\nTotal Sales: $" << total << endl;
cout << "Average Sales: $" << average << endl;

// Free dynamically allocated memory
delete [] sales;
sales = 0;       // Make sales point to null.

return 0;
Program Output with Example Input Shown in Bold

How many days of sales figures do you wish to process? 5 [Enter]
Enter the sales figures below.
Day 1: 898.63 [Enter]
Day 2: 652.32 [Enter]
Day 3: 741.85 [Enter]
Day 4: 852.96 [Enter]
Day 5: 921.37 [Enter]

Total Sales: $4067.13
Average Sales: $813.43

Notice that in line 49 the value 0 is assigned to the sales pointer. It is a good practice to store 0 in a pointer variable after using delete on it. First, it prevents code from inadvertently using the pointer to access the area of memory that was freed. Second, it prevents errors from occurring if delete is accidentally called on the pointer again. The delete operator is designed to have no effect when used on a null pointer.
9.9

Returning Pointers from Functions
Returning Pointers from Functions

- Pointer can be the return type of a function:
  ```c
  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;
}