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:

\[
\text{int num} = -99; \\
\text{cout } \ll \text{ num; // prints address in hexadecimal}
\]

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 \text{numbers} to the \text{showValues} function:

\[
\text{showValues(numbers, SIZE);}
\]
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

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

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

---

Pointer Variables

- **Definition:**
  
  ```
  int * intptr;
  ```

- **Read as:**
  
  "intptr can hold the address of an int"

- **Spacing in definition does not matter:**
  
  ```
  int * IntPtr; // same as above
  int* intptr; // same as above
  ```

---

Assigning an address to a pointer variable:

```
int * intptr;
intptr = &num;
```
Pointer Variables

- Initialize pointer variables with the special value `nullptr`.
- In C++ 11, the `nullptr` key word was introduced to represent the address 0.
- Here is an example of how you define a pointer variable and initialize it with the value `nullptr`:

```c++
int *ptr = nullptr;
```

The Indirection Operator

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

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

This prints 25.

The Relationship Between Arrays and Pointers

9.3

The Indirection Operator in Program 9-3

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

int main()
{
    int x = 25;
    int intptr = &x;
    cout << *intptr << endl;
    return 0;
}
```

Program Output

The value in x is 25
The address of x is 0x60000000

Program 9-3 (continued)

```c++
// Use both x and ptr to display the value in x.
// cout << *intptr; prints the contents of x
// cout << *ptr; prints the contents of x
int x = 100;
int intptr = &x;
int ptr = &x;

int main()
{
    cout << *intptr;
    cout << *ptr;
    cout << endl;
    return 0;
}
```

Program Output

Here is the value in x, printed twice:
25
25
Now again, here is the value in x:
100
100

A Pointer Variable in Program 9-2

```c++
// This program stores the address of a variable in a pointer.
#include <iostream>
using namespace std;

int main()
{
    int x = 25; // int variable
    int *ptr = nullptr; // Pointer variable, can point to an int
    ptr = &x; // Store the address of x in ptr
    cout << "The address of x is " << ptr << endl;
    return 0;
}
```

Program Output

The value in x is 25
The address of x is 0x60000000
The Relationship Between Arrays and Pointers

- Array name is starting address of array
  
  ```c
  int vals[] = {4, 7, 11};
  
  starting address of vals: 0x4a00
  cout << vals; // displays 0x4a00
  cout << vals[0]; // displays 4
  ```

- Array name can be used as a pointer constant:
  
  ```c
  int vals[] = {4, 7, 11};
  cout << *vals; // displays 4
  ```

- Pointer can be used as an array name:
  
  ```c
  int *valptr = vals;
  cout << valptr[1]; // displays 7
  ```

The Array Name Being Dereferenced in Program 9-5

Program 9-5

```c
#include <iostream>
using namespace std;

int main()
{
  short numbers[] = {10, 20, 30, 40, 50};
  cout << "The first element of the array is ";
  cout << numbers[0];
  return 0;
}
```

Program Output

The first element of the array is 10

Pointers in Expressions

Given:

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

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

```c
cout << *(valptr+1); //displays 7
cout << *(valptr+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 [i]</td>
<td>vals[2] = 17;</td>
</tr>
<tr>
<td>pointer to array and [i]</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>

Conversion: vals[i] is equivalent to *(vals + i)

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

Operations on pointer variables:

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

9.5 Initializing Pointers

- Can initialize at definition time:
  - `int num, *numptr = &num;`
  - `int val[3], *valptr = val;`

- Cannot mix data types:
  - `double cost;
    int *ptr = &cost; // won't work`

- Can test for an invalid address for `ptr` with:
  - `if (!ptr) ...`
9.6 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
```

9.7 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
```
void getNum(int *ptr); // ptr is pointer to an int
```
2) Asterisk * in body to dereference the pointer
```
cin >> *ptr;
```
3) Address as argument to the function
```
getNum(&num);     // pass address of num to getNum
```

Example

```
void swap(int *x, int *y)
{   int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
int num1 = 2, num2 = -3;
swap(&num1, &num2);
```

Pointers as Function Parameters in Program 9-11

```
void swap(int *x, int *y)
{   int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
int num1 = 2, num2 = -3;
swap(&num1, &num2);
```

(Program Continues)
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.

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.

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.

```c
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 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:
```c
int value = 22;
const int * const ptr = &value;
```

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:
  ```c
double *dptr = nullptr;
dptr = new double;
```
- new returns address of memory location
Releasing Dynamic Memory

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

Dynamic Memory Allocation in Program 9-14

```cpp
// 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 = nullptr; // To dynamically allocate an array
  total = sума; // accumulator
  average; // To hold average sales
  int numbDays; // To hold the number of days of sales
  cout; // Counter variable
  
  // Get the number of days of sales.
  cout << "How many days of sales figures do you wish?");
  cin >> numbDays;

  // Dynamically allocate an array large enough to hold
  // that many days of sales amounts.
  sales = new double[ numbDays ];

  // Set the sales figures for each day.
  // sum of sales for the sales figures below n:
  // cout << setw(5) << sales[0] << " \";
  // cout << setw(5) << sales[1] << " \";

  // Display the database:
  // even << setw(5) << sales[0] " \"; sales[0] << " \";

  // Calculate the total sales
  // sum = total / numbDays;
  // total = sales / numbDays;

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

  // Display the results
  // cout << setw(5) << sales[0] " \";
  // cout << setw(5) << sales[1] " \";

  // Free dynamically allocated memory
  delete [] sales;
  sales = nullptr;

  return 0;
}
```

9.9 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

Program Output with Example Input Shown in Bold

How many days of sales figures do you wish to process? 9 [Enter]
Sales: 892.12 [Enter]
May: 741.43 [Enter]
May: 612.50 [Enter]
May: 520.37 [Enter]
Total sales: 3664.2
Average sales: 407.14
Using Smart Pointers to Avoid Memory Leaks

In C++ 11, you can use smart pointers to dynamically allocate memory and not worry about deleting the memory when you are finished using it.

Three types of smart pointer:

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

Must include the memory header file:
```cpp
#include <memory>
```

In this book, we introduce `unique_ptr`:
```cpp
unique_ptr<int> ptr( new int );
```

The `<int>` indicates that the pointer can point to an `int`.
The name of the pointer is `ptr`.
The expression `new int` allocates a chunk of memory to hold an `int`.
The address of the chunk of memory will be assigned to `ptr`.

Using Smart Pointers in Program 9-17

```cpp
Program 9-17
```