Chapter 9: Pointers

9.1 Getting the Address of a Variable

- Each variable in a program is stored at a unique address.
- Use the address operator `&` to get the address of a variable:
  ```
  int num = -99;
  cout << &num; // prints address in hexadecimal
  ```

9.2 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);
  ```
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);
```

Pointer Variables

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

- **Read as:**
  “`intptr` can hold the address of an int”

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

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

- **Memory layout:**

```cpp
address of num: 0x4a00
intptr 0x4a00
```

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

```cpp
int *ptr = nullptr;
```

A Pointer Variable in Program 9-2

Program 9-2

```
// This program stores the address of a variable in a pointer.

int x = 25; // int variable
int * intptr = &x; // Pointer variable, one point to an int
ptr = &x; // store the address of x in ptr
cout << "The value in x is " << x << endl;
cout << "The address of x is " << ptr << endl;
return 0;
```

Program Output

The value in x is 25
The address of x is 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
- 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 *valptr = vals;
cout << valptr[1]; // displays 7
```
The Array Name Being Dereferenced in Program 9-5

Program 9-5
1: // This program shows an array name being dereferenced with the * operator.
2: #include <iostream>
3: using namespace std;
4: int main()
5: {
7:   int i;
8:   cout << "The first element of the array is: ";
9:   return 0;
10:}

Program Output
The first element of the array is 10

Pointers in Expressions

Given:
- int vals[]={4,7,11};
- *valptr;
- valptr = vals;

What is valptr + 1?
It means (address in valptr) + (1 * size of an int)
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:
  - array name and []
  - pointer to array and []
  - array name and subscript arithmetic
  - pointer to array and subscript arithmetic

<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</td>
<td>*(vals + 2) = 17;</td>
</tr>
<tr>
<td>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

From Program 9-7

```c++
9.4
Pointer Arithmetic
```

Program 9-7
9: case int 0x99C99 = 9;
10: double double99C99 = 0.39, 0.1, 0.29, 0.39, 0.79;
11: double double99C99; // allocate a double
12: int main()
13: {  // Array index
14:   double theAddressOfTheArray = &double99C99;
15:   double99C99 = theAddressOfTheArray;
16:   // Display the contents of the array, five subroutines
17:   // with the program.
18:   cout << "This is the value at the array address(\n);";
19:   for (count = 0; count < array99C99.count++;)
20:     cout << "The version (\n count = 99C99.count++) in the array
";
21:   // Display the contents of the array again, but this time
22:   // with a pointer to the array.
23:   // first time this is executed. This is why the error
24:   if (n = array99C99; n + = 1;)
25:     cout << "Show how they are again\n;";
26:     cout << "This is the value at the array
";
27:     for (count = 0; count < array99C99.count++;)
28:     cout << "The version (\n count = 99C99.count++) in the array
";
29:   cout << "\n;";```

Program Output
Here are the values in the array object:
0.39 0.1 0.29 0.39 0.79
After they get assigned:
0.39 0.1 0.29 0.39 0.79
Pointer Arithmetic

- Operations on pointer variables:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>++, --</td>
<td>int vals[3]={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></td>
<td>-valptr; // now points at 4</td>
</tr>
<tr>
<td></td>
<td>cout &lt;&lt; *(valptr + 2); // 11</td>
</tr>
<tr>
<td></td>
<td>valptr = vals; // points at 4 + 2;</td>
</tr>
<tr>
<td></td>
<td>valptr = 2; // points at 11</td>
</tr>
<tr>
<td></td>
<td>cout &lt;&lt; valptr-val; // difference (\text{(number of ints)}) between valptr (\text{and val})</td>
</tr>
</tbody>
</table>

From Program 9-9

```cpp
// From Program 9-9
```

Initializing Pointers

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

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

- Can test for an invalid address for `ptr` with:
  ```cpp
  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:
  ```cpp
  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

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

```c
int num1 = 2, num2 = -3;
swap(&num1, &num2);
```

Pointers as Function Parameters in Program 9-11

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:
  
  ```
  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.

  ```
  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 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

• 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 = nullptr;
  dptr = new double;
  ```

• Uses new operator 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!
Notice that in line 48 nullptr is assigned to the sales pointer. The delete operator is designed to have no effect when used on a null pointer.

Returning Pointers from Functions

- Pointer can be the return type of a function:
  ```cpp
  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

From Program 9-15

```cpp
int *getRandomNumbers(int num) {
    int *arr = new int[num]; // Array to hold the numbers
    // Return a null pointer if num is zero or negative.
    if (num == 0) {
        return nullptr;
    }
    // Dynamically allocate the array.
    arr = new int[num];
    // Seed the random number generator by passing
    // the secure value of time(); to srand.
    srand(time(0));
    // Populate the array with random numbers.
    for (int count = 0; count < num; count++) {
        arr[count] = rand();
    }
    // Return a pointer to the array.
    return arr;
}
```