Getting the Address of a Variable

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

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

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:

```cpp
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.

```cpp
int values[] = {1, 2, 3, 4, 5};

void showValues(int* values, int size) {
    for (int i = 0; i < size; i++) {
        cout << values[i] << endl;
    }
}
```

### 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
  intptr = &num;
  ```

- **Memory layout:**

  ```cpp
  int ptr
  address of num: 0x4a00
  ```

- **Assigning an address to a pointer variable:**

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

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

9.3

The Indirection Operator in Program 9-3

```cpp
// 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;
  // Print the contents of x
  cout << "The address of x is " << uintptr << " ptr of addr;" << endl;
  return 0;
}
```

Program Output

The address of x is 0x7fa800000018
The contents of x is 25
The Relationship Between Arrays and Pointers

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

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

The Array Name Being Dereferenced in Program 9-5

**Program 9-5**

```
// This program shows an array being dereferenced with the "*" operator.
#include <iostream>

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

**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 []**
    ```
    vals[2] = 17;
    ```
  - **Pointer to array and []**
    ```
    *valptr[2] = 17;
    ```
  - **Array name and subscript arithmetic**
    ```
    *(vals + 2) = 17;
    ```
  - **Pointer to array and subscript arithmetic**
    ```
    *(valptr + 2) = 17;
    ```

- **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>
| **++, --** | `valptr++;` // points at 7  
 | `valptr--;` // now points at 4 |
| **+ (pointer and int)** | `cout << *(valptr + 2);` // 11 |
| **+= (pointer and int)** | `valptr += 2;` // points at 4 |
| **- (pointer from pointer)** | `cost += valptr-val;` // difference  
 | `// (number of ints) between valptr  
 | `// and val` |

### 9.5 Initializing Pointers

#### Initializing Pointers

- Can initialize at definition time:
  ```
  int num, *numptr = &num;
  ```
- 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:

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

Example

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

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

Pointers as Function Parameters in Program 9-11
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.

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.

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

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

Dynamic Memory Allocation in Program 9-14

```c++
// 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 <cstdlib>
using namespace std;

int main()
{
  double *sales = nullptr; // To dynamically allocate an array
  int sales[7]; // To hold average sales
  int total = 0;
  // Input variable
  double avg = 0.0;
  // The number of days of sales
  cout << "How many days of sales do you wish? ";
  cin >> numDays;
  sales[0] = nullptr;
}
```

Notice that in line 49 `nullptr` is assigned to the sales pointer. The delete operator is designed to have no effect when used on a null pointer.

Dynamic Memory Allocation in Program 9-14 (Continued)

```c++
// Program Output with Example Input Shown Below
Enter the sales figures below: Enter the sales figures below.
Total sales: 41771.23 Average sales: 5967.31
```

9.9 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 *mem)
{
    int *arr = malloc(sizeof(int) * *mem); // Array to hold the numbers
    if (*mem == 0)
        return NULL;
    // Dynamically allocate the array.
    arr = malloc(sizeof(int) * *mem);
    srand(time(0));
    // Seed the random number generator by passing
    // the current value of time() as seed.
    srand(time(0));
    // Populate the array with random numbers.
    for (int count = 0; count < *mem; count++)
        arr[count] = rand();
        // Assign a pointer to the array.
    return arr;
}