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
```
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);
  ```

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

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

  The `values` parameter, in the `showValues` function, points to the `numbers` array.
Lecture 09 - Pointers

POINTER VARIABLES
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 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.

- 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
  ```
**POINTER VARIABLES**

- Assigning an address to a pointer variable:
  ```
  int *intptr;
  intptr = &num;
  ```
- Memory layout:
  ```
  int     intptr
  25      0x4a00
  ```
  address of num: 0x4a00

---

**POINTER VARIABLES**

**Program 9-2**

```cpp
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
POINTER VARIABLES

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

```cpp
// 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 << 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 << x << endl;  // Displays the contents of x
    cout << *ptr << endl; // Displays the contents of x
    return 0;
}
```
POINTER VARIABLES

**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

THE RELATIONSHIP BETWEEN ARRAYS AND POINTERS

- Array name is starting address of array
  ```
  int vals[] = {4, 7, 11};
  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 RELATIONSHIP BETWEEN ARRAYS AND POINTERS

Program 9-5

```cpp
1 // This program shows an array name being dereferenced with the *
2 // operator.
3 include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     short numbers[] = {10, 20, 30, 40, 50};
9
10    cout << "The first element of the array is ";
11    cout << *numbers << endl;
12    return 0;
13 }
```

Program Output

The first element of the array is 10

THE RELATIONSHIP BETWEEN ARRAYS AND POINTERS

POINTERS IN EXPRESSIONS

Given:

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

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

```cpp
cout << *(valptr+1); //displays 7
cout << *(valptr+2); //displays 11
```

Must use () as shown in the expressions
THE RELATIONSHIP BETWEEN ARRAYS AND POINTERS

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>

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

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

PROGRAM

```c
const int NUM_COINS = 5;
double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
double *doublePtr; // Pointer to a double
int count; // Array index

// Assign the address of the coins array to doublePtr.
doublePtr = coins;

// Display the contents of the coins array. Use subscripts with the pointer.
cout << "Here are the values in the coins array:\n";  
for (count = 0; count < NUM_COINS; count++)  
    cout << doublePtr[count] << " ";

// Display the contents of the array again, but this time use pointer notation with the array name.
cout << "And here they are again:\n";  
for (count = 0; count < NUM_COINS; count++)  
    cout << *(coins + count) << " ";

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
### POINTER ARITHMETIC

#### OPERATIONS ON POINTER VARIABLES

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

#### PROGRAM

```cpp
const int SIZE = 8;
int size[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 << "The 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
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:
  ```c
  if (ptr1 == ptr2)    // compares // addresses
                      // contents
  if (*ptr1 == *ptr2) // compares
  ```
POINTER 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**

**PROGRAM**

**Program 9-11**

```cpp
// This program uses two functions that accept addresses of
// variables as arguments.
#include <iostream>
using namespace std;

// Function prototypes
void getNumber(int *);
void doubleValue(int *);

int main()
{
    int number;

    // Call getNumber and pass the address of number.
    getNumber(&number);

    // Call doubleValue and pass the address of number.
    doubleValue(&number);

    // Display the value in number.
    cout << "That value doubled is " << number << endl;
    return 0;
}
```

**Program 9-11 (continued)**

```cpp
// Definition of getNumber. The parameter, input, is a pointer.
// This function asks the user for a number. The value entered
// is stored in the variable pointed to by input.
void getNumber(int *input)
{
    cout << "Enter an integer number: ";
    cin >> *input;
}

// Definition of doubleValue. The parameter, val, is a pointer.
// This function multiplies the variable pointed to by val by 2.
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 as 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;
    }
}
```

# 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;
  dptr = new double;
  ```
- `new` returns address of memory location
- 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:
  ```c
  delete fptr;
  ```
- Use `[]` to free dynamic array:
  ```c
  delete [] arrayptr;
  ```
- Only use `delete` with dynamic memory!

DYNAMIC MEMORY

PROGRAM

```c
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
```
DYNAMIC MEMORY
PROGRAM

Program 9-14  (continued)

```cpp
class Program 9-14
{  
  int numDays;    // To hold the number of days of sales
  int count;      // Counter variable

  // Get the number of days of sales.
cout << "How many days of sales figures do you wish ";
cin >> numDays;

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

  // Get the sales figures for each day.
cout << "Enter the sales figures below.\n";
for (count = 0; count < numDays; count++)
{  
  cout << "Day " << (count + 1) << ": ";
cin >> sales[count];
}

  // 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 << "\nTotal Sales: $" << total << endl;
cout << "Average Sales: $" << average << endl;

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

 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
RETURNING POINTERS FROM FUNCTIONS

```c
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;
}
```