9.1 Getting the Address of a Variable

- A variable has:
  - Name
  - Value
  - Location in memory
  - Type

- The location in memory is an 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 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

- Declaring a pointer:

  ```cpp
  type *variableName;
  ```

- Assigning an address to a pointer variable:

  ```cpp
  int *intptr;  // same as above
  intptr = num;  // same as above
  ```

- Memory layout:

  ```plaintext
<table>
<thead>
<tr>
<th>num</th>
<th>intptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0x4a00</td>
</tr>
</tbody>
</table>
  ```

- Example:

  ```cpp
  int*pointer1;
  double *pointer2;
  char *pointer3;
  int number;
  double number2;
  char letter1;
  
  pointer1=& number;
  pointer2=&number2;
  pointer3=&letter1;
  ```

9 Getting the Address of a Variable

Variable

- Use address operator \& to get address of a variable
- The location in memory is an address
  - Type
  - Location in memory
  - Value
  - Name
- A variable has:

CS 1400
**Pointer Variable**

- You can print it out:
  ```
  cout << pointer3;
  ```
- With the address of a variable you have complete access to the variable

**Example**

```
9
```

**The Indirection Operator**

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

```
int x = 25;
int *intptr = &x;

cout << *intptr << endl;
```

This prints 25.

**Notes**

- be careful * is used in 3 different ways
  - Multiplication
  - to define a pointer variable
  - Indirection Operator

**Example**

```
This prints 25.
```

**The Indirection Operator**

- A pointer.
  - If allows you to access the item that the pointer points to.
  - The indirection operator (`*`) dereferences the address of a variable to the variable.

**Example**

```
int x = 25;
int *intptr = &x;

cout << *intptr << endl;
```

You can print it out:
Arrays vs. Pointers

• An array name is a const pointer
• It contains the address of the beginning of an array and cannot be changed

```cpp
int vals[] = {4, 7, 11};
starting address of vals: 0x4a00
```

```cpp
cout << vals; // displays 0x4a00
cout << vals[0]; // displays 4
```

• You can do the same things with pointers and array names.
• Exception: you cannot change the value of the array name

```cpp
int myArray[SIZE] = {5, 23, 78, 62, 3};
int* myPointer;
myPointer = myArray; // correct
myArray = myPointer; // wrong
```

• Since the name of the array is already an address you do not use the & operator:

```cpp
myPointer = myArray;
```

• If you want to get the address of an individual element of an array you would use the & operator:

```cpp
myPointer = &myArray[2];
```

Array Access

• Arrays vs. Pointers
• Since array name can be used as a pointer constant:

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

```cpp
cout << *vals; // displays 4
```

• Pointer can be used as an array name:

```cpp
int *valptr = vals;
```

```cpp
cout << valptr[1]; // displays 7
```

• Conversion:

```cpp
*(valptr + 2) = 17;
```

```cpp
*(vals + 2) = 17;
```

```cpp
valptr[2] = 17;
```

```cpp
vals[2] = 17;
```

Example Array access method

• Conversion:

```cpp
*(valptr + 2) = 17;
```

```cpp
*(vals + 2) = 17;
```

```cpp
valptr[2] = 17;
```

```cpp
vals[2] = 17;
```

Array Access

• Array elements can be accessed in many ways:

```cpp
*(valptr + 2) = 17;
```

```cpp
*(vals + 2) = 17;
```

```cpp
valptr[2] = 17;
```

```cpp
vals[2] = 17;
```

Pointers in Expressions

```cpp
Given:
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
```

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

Must use parentheses as shown in the expressions.

Arrays vs. Pointers

• Conversion:

```cpp
vals[i]
```

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

```cpp
*(vals + i)
```

• Pointer can be used as an array name:

```cpp
int *valptr = vals;
```

```cpp
cout << valptr[1]; // displays 7
```

• Since array name can be used as a pointer constant:

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

```cpp
cout << vals[1]; // displays 7
```

Array Access

• Conversion:

```cpp
vals[i]
```

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

```cpp
*(vals + i)
```

• Pointer can be used as an array name:

```cpp
int *valptr = vals;
```

```cpp
cout << valptr[1]; // displays 7
```

• Since array name can be used as a pointer constant:

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

```cpp
cout << vals[1]; // displays 7
```

Array Access

• Conversion:

```cpp
vals[i]
```

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

```cpp
*(vals + i)
```

• Pointer can be used as an array name:

```cpp
int *valptr = vals;
```

```cpp
cout << valptr[1]; // displays 7
```

• Since array name can be used as a pointer constant:

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

```cpp
cout << vals[1]; // displays 7
```
9.4 Pointer Arithmetic

- Operations on pointer variables:
  - cout << valptr–val; // difference
  - (number of ints) between valptr
  - (pointer from pointer)
  - valptr = vals; // points at 4
  - valptr += 2;   // points at 11

- +=, -= (pointer
  - and
  - int)

- cout << *(valptr + 2); // 11 +, -

- valptr++; // points at 7
- valptr--; // now points at 4

Example
- int *valptr = vals;

9.5 Initializing Pointers

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

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

- Can test for an invalid address for ptr:
  - if (!ptr) double cost?

- Cannot mix data types:
  - int val[3], *valptr = val?
  - num->number = num?

- Can initialize at definition time:

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

- A pointer can be a parameter

- Works like reference variable to allow change to
  - argument from within function

- Requires:
  - asterisk * on parameter in prototype and heading
  - void getNum(int *ptr); // ptr is pointer to an int
  - asterisk * in body to dereference the pointer
  - cin >> *ptr;
  - address as argument to the function
  - getNum(&num);     // pass the address of num to getNum
  - Or getNum(ptr1);  // pass the pointer to getNum
  - Or getNum(myArray); // pass the address of array
Pointers as Function Parameters

• Since you are passing a memory address—the function has complete access to the variable.
• It can read and write the value at the address.

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

Example:

```c
const int SIZE = 6;
const double payRates[SIZE] = { 18.55, 17.45, 12.85, 14.97, 10.35, 18.89 };
```

It can read and write the value at the variable.

Example:

```c
void swap(int const *x, int const *y)
{
    int temp = *y;
    *y = *x;
    *x = temp;
    printf("(%d, %d)\n", *x, *y);
}
```

The function has complete access to the variable.

Example:

```c
void swap(int const *x, int const *y)
{
    int temp = *y;
    *y = *x;
    *x = temp;
    printf("(%d, %d)\n", *x, *y);
}
```

Since you are passing a memory address to the void swap( int const *x, int const *y )

Pointers as Function Parameters
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`.

A constant pointer is a pointer that is initialized with an address, and cannot point to anything else.

```cpp
int value = 22;
int * const ptr = &value;
```

A constant pointer is a pointer that points to a constant.

A constant pointer to a constant is:

```cpp
int value = 22;
const int * const ptr = &value;
```

- a pointer that cannot point to anything except what it is pointing to
- a pointer that is a pointer to a constant
9.8 Dynamic Memory Allocation

- Can allocate storage for a variable while the program is running.
- The computer returns the address of the newly allocated variable.
- Uses the `new` operator to allocate memory:
  ```cpp
double *dptr;
dptr = new double;
```
  - `new` returns the address of the memory location.

- Dynamic memory can also be allocated to an array:
  ```cpp
  const int SIZE = 25;
  double *arrayPtr = new double[SIZE];
  ```
  - Can then use either `[]` or pointer arithmetic to access the array:
    ```cpp
    for(i = 0; i < SIZE; i++)
      arrayPtr[i] = i * i;
    ```
  - For pointer arithmetic:
    ```cpp
    for(i = 0; i < SIZE; i++)
      *(arrayPtr + i) = i * i;
    ```

- The program will terminate if not enough memory is available to allocate the dynamic memory.

### Releasing Dynamic Memory

- Use `delete[]` to release dynamically allocated arrays:
  ```cpp
  delete[] arrayPtr;
  ```
- Use `delete` to release dynamically allocated variables:
  ```cpp
  delete fptr;
  ```

- Only use `delete[]` to release dynamically allocated arrays.

---

Program 9-14 (Continued)
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 it. First, it prevents code from inadvertently using the pointer to access the 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

- A function that returns a pointer to a value.

```c
int* test(int array[], int size, int value) {
    for (int index = 0; index < size; index++) {
        if (array[index] == value)
            return &array[index];
    }
    return NULL;
}
```

- What do we return if we don't find the value?

The NULL Pointer

- This is a special value that is of type pointer.
- It refers to no actual address.
- Do not try to dereference a pointer that may contain the NULL pointer.
- It may contain the NULL pointer.

In the calling function you need to test for NULL before you dereference the pointer.

```c
int* ptr1 = test(myArray, size, myValue);
if (ptr1 == NULL)
    cout << "Value not found\n";
else
    cout << "The value " << *ptr1;
```

- Beware of returning a pointer to memory that no longer exists.

- Pointer can be the return type of a function.
- In function call:
- In function definition:
- In prototype: