Pointers

Class 29

Recap

- a variable is allocated exactly enough memory to hold one value of the declared type

```
int value;
```

int 1234 value

```
double price;
```

double 123.4567 price

```
char initial;
```

char 'A' initial

Variables in Memory

- a computer's memory is a list of numbered locations, each of which refers to a byte of 8 bits
- the number of a byte is its address
- a simple variable (e.g., int or double) refers to a portion of memory containing a number of consecutive bytes
- the number of bytes is determined by the type of the variable (e.g., on ice, 4 bytes for unsigned, 8 bytes for double)
- the address of the variable is the address of the first byte where it is located

Address Operator

- when you use a variable in a program, the compiler assumes you want the contents of that variable's location in memory

Address Operator

- when you use a variable in a program, the compiler assumes you want the contents of that variable's location in memory
- but sometimes you actually want the address of the variable in memory

Address Operator

- when you use a variable in a program, the compiler assumes you want the contents of that variable's location in memory
- but sometimes you actually want the address of the variable in memory
- sometimes you also want to know how many bytes of memory a variable occupies

Address Operator

- when you use a variable in a program, the compiler assumes you want the contents of that variable's location in memory
- but sometimes you actually want the address of the variable in memory
- sometimes you also want to know how many bytes of memory a variable occupies
- there is a way to do each of these

Address Operator

- when you use a variable in a program, the compiler assumes you want the contents of that variable's location in memory
- but sometimes you actually want the address of the variable in memory
- sometimes you also want to know how many bytes of memory a variable occupies
- there is a way to do each of these

- to get a variable's address, we use the address-of operator: &

Address Operator

- when you use a variable in a program, the compiler assumes you want the contents of that variable's location in memory
- but sometimes you actually want the address of the variable in memory
- sometimes you also want to know how many bytes of memory a variable occupies
- there is a way to do each of these

- to get a variable's address, we use the address-of operator: &
- to get the number of bytes a variable holds, we use the sizeof operator (it looks like a function, but it's really an operator)

Address Operator

- when you use a variable in a program, the compiler assumes you want the contents of that variable's location in memory
- but sometimes you actually want the address of the variable in memory
- sometimes you also want to know how many bytes of memory a variable occupies
- there is a way to do each of these

- to get a variable's address, we use the address-of operator: &
- to get the number of bytes a variable holds, we use the sizeof operator (it looks like a function, but it's really an operator)

see program_9_1.cpp

New Things

- the new concepts introduced in program 9-1:
  - the address-of operator &

```
cout << "the address of x is " << &x
```

New Things

- the new concepts introduced in program 9-1:
  - the address-of operator &
    ```
    cout << "the address of x is " << &x
    ```
  - the sizeof operator
    ```
    cout << "the size of x is " << sizeof x
    ```

Pointer Variables

- a pointer variable aka pointer is a variable that holds a memory address

Pointer Variables

- a pointer variable aka pointer is a variable that holds a memory address
- just as the purpose of an `int` is to hold an integer

Pointer Variables

- a pointer variable aka pointer is a variable that holds a memory address
- just as the purpose of an `int` is to hold an integer
- and a `double` is to hold a double

Pointer Variables

- a pointer variable aka pointer is a variable that holds a memory address
- just as the purpose of an `int` is to hold an integer
- and a `double` is to hold a double
- the purpose of a pointer is to hold an address

Pointer Variables

- a pointer variable aka pointer is a variable that holds a memory address
- just as the purpose of an `int` is to hold an integer
- and a `double` is to hold a double
- the purpose of a pointer is to hold an address
- this allows you to indirectly reference a memory location though the use of a variable that "points to" another location

Reference Variables

- you have used variables that refer to other locations already

Reference Variables

- you have used variables that refer to other locations already
- a reference parameter is an alias for the "real" variable that is located in the calling scope

Reference Variables

- you have used variables that refer to other locations already
- a reference parameter is an alias for the "real" variable that is located in the calling scope
- an array parameter is an alias for the "real" array that is located in the calling scope

Reference Variables

- you have used variables that refer to other locations already
- a reference parameter is an alias for the "real" variable that is located in the calling scope
- an array parameter is an alias for the "real" array that is located in the calling scope

- pointers are very similar to references, but operate at a lower level

Reference Variables

- you have used variables that refer to other locations already
- a reference parameter is an alias for the "real" variable that is located in the calling scope
- an array parameter is an alias for the "real" array that is located in the calling scope

- pointers are very similar to references, but operate at a lower level
- almost all the mechanics of references are done for you by the compiler

Reference Variables

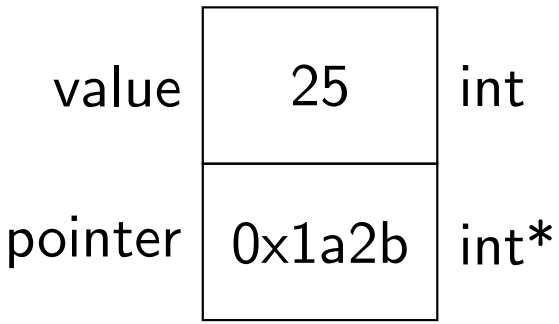
- you have used variables that refer to other locations already
- a reference parameter is an alias for the "real" variable that is located in the calling scope
- an array parameter is an alias for the "real" array that is located in the calling scope

- pointers are very similar to references, but operate at a lower level
- almost all the mechanics of references are done for you by the compiler
- pointers require you to do the mechanics yourself

Declaring a Pointer

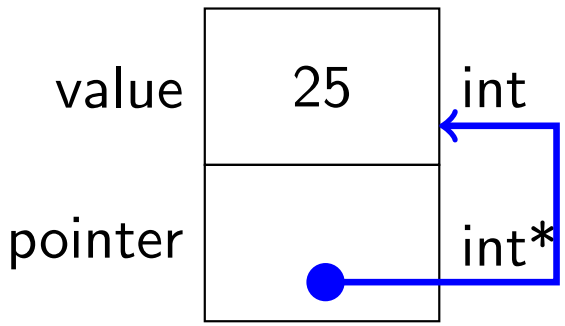
```
int value = 25;
int* pointer = &value;

cout << "value: " << value << endl;
cout << "value's address: " << pointer
  << endl;
```



Depicting a Pointer

- usually instead of writing the actual address value
- we show the address symbolically with an arrow
- this shows the pointer variable pointing to a different memory location



Using a Pointer Variable

- once a pointer variable has a valid value, it can be used
- the value in the pointer variable itself is an address, usually not directly useful
- to get at the value the pointer is pointing to, we must dereference it using the dereference operator *

```
1 int value = 5;
2 int* pointer = &value;
3
4 value++;
5 *pointer += 5;
6 cout << "value is " << value << endl;
7 cout << "pointer points to " << *pointer << endl;
```

draw a picture of memory

A Note on Initialization

- your author makes a big deal about initializing pointer variables with nullptr the instant they are declared
- he frequently has code like this:

```
int* svalue = nullptr;
*value = 5;
```

- this is poor form
- the rules of pointer declaration and initialization are no different than for any other variable
  1. declare a variable as close to the point of use as possible
  2. initialize a variable at declaration if necessary and useful
  3. do not initialize a variable needlessly, such as if it's immediately going to be given a value with an input statement

Pointers and Arrays

- when we introduced arrays, we said that the array variable name itself, without brackets, really stored the starting address of the array

Pointers and Arrays

- when we introduced arrays, we said that the array variable name itself, without brackets, really stored the starting address of the array
- but that's exactly what a pointer is!

Pointers and Arrays

- when we introduced arrays, we said that the array variable name itself, without brackets, really stored the starting address of the array
- but that's exactly what a pointer is!
- an array name is a pointer

Pointers and Arrays

- when we introduced arrays, we said that the array variable name itself, without brackets, really stored the starting address of the array
- but that's exactly what a pointer is!
- an array name is a pointer
- here, I'll prove it:

```
int numbers[] {10, 20, 30};
cout << *numbers << endl; // this prints 10!
```

Pointers and Arrays

- it gets stranger:

Pointers and Arrays

- it gets stranger:

```
int numbers[] {10, 20, 30};
cout << *numbers << endl; // this prints 10
cout << *(numbers + 1) << endl; // this prints 20!
cout << *(numbers + 2) << endl; // this prints 30!
```

Pointers and Arrays

- it gets stranger:

```
int numbers[] {10, 20, 30};
cout << *numbers << endl; // this prints 10
cout << *(numbers + 1) << endl; // this prints 20!
cout << *(numbers + 2) << endl; // this prints 30!
```

- remember, numbers refers to the address of a byte of memory

Pointers and Arrays

- it gets stranger:

```
int numbers[] {10, 20, 30};
cout << *numbers << endl; // this prints 10
cout << *(numbers + 1) << endl; // this prints 20!
cout << *(numbers + 2) << endl; // this prints 30!
```

- remember, numbers refers to the address of a byte of memory
- but numbers + 1 does not refer to the byte after numbers

Pointers and Arrays

- it gets stranger:

```
int numbers[] {10, 20, 30};
cout << *numbers << endl; // this prints 10
cout << *(numbers + 1) << endl; // this prints 20!
cout << *(numbers + 2) << endl; // this prints 30!
```

- remember, numbers refers to the address of a byte of memory
- but numbers + 1 does not refer to the byte after numbers
- the compiler knows that an int takes up 4 bytes

Pointers and Arrays

- it gets stranger:

```
int numbers[] {10, 20, 30};
cout << *numbers << endl; // this prints 10
cout << *(numbers + 1) << endl; // this prints 20!
cout << *(numbers + 2) << endl; // this prints 30!
```

- remember, numbers refers to the address of a byte of memory
- but numbers + 1 does not refer to the byte after numbers
- the compiler knows that an int takes up 4 bytes
- thus "numbers + 1" is really "numbers plus enough bytes to get to the next int"

Pointers and Arrays

- it gets stranger:

```
int numbers[] {10, 20, 30};
cout << *numbers << endl; // this prints 10
cout << *(numbers + 1) << endl; // this prints 20!
cout << *(numbers + 2) << endl; // this prints 30!
```

- remember, numbers refers to the address of a byte of memory
- but numbers + 1 does not refer to the byte after numbers
- the compiler knows that an int takes up 4 bytes
- thus "numbers + 1" is really "numbers plus enough bytes to get to the next int"
- in other words, "numbers plus sizeof int"

Syntactic Sugar

values[index]

and

*(values + index)

are exactly the same thing

Arrays and Pointers

- array names and pointers are interchangeable
- each cout below prints two identical values

```
double coins1[] {0.01, 0.05, 0.1, 0.25, 0.5, 1.0};
double* coins2 = coins1;

cout << coins1[0] << " and " << *coins2 << endl;
cout << coins1[1] << " and " << *(coins2 + 1) << endl;
cout << *(coins1 + 2) << " and " << coins2[2] << endl;
```

Arrays and Pointers

- there is one difference between pointers and array names
- a pointer can be reassigned to point to different things, but an array name cannot be reassigned

```
int values1[] {1, 2, 3, 4, 5};
int values2[] {6, 7, 8, 9, 10};
int* pointer = &values1[2]; // points to one thing
pointer = &values2[4];  // now points to a different thing
pointer = values1; // now points to yet another thing

values1 = pointer; // illegal! cannot change what values1 points to
```

- so an array name is a constant pointer

Pointer Arithmetic

- since a pointer stores a numeric value, you can use arithmetic operators on it

```
double coins[] {0.01, 0.05, 0.1, 0.25, 0.5, 1.0};

double* d_pointer = &coins[2]; // d_pointer points to the dime
d_pointer++; // now points to the quarter
d_pointer -= 2; // now points to the nickel
```

- illegal to multiply or divide pointers, can only add and subtract

Comparing Pointers

- pointers may be compared using any of the relops
- what does each of the following lines print?

```
cout << (coins < &coins[1]) << endl;
cout << (coins < &coins[4]) << endl;
cout << (coins == &coins[0]) << endl;
cout << (&coins[2] == coins + 2) << endl;
cout << (&coins[2] != &coins[3]) << endl;
```

Comparing Pointers

- pointers may be compared using any of the relops
- what does each of the following lines print?

```
cout << (coins < &coins[1]) << endl;
cout << (coins < &coins[4]) << endl;
cout << (coins == &coins[0]) << endl;
cout << (&coins[2] == coins + 2) << endl;
cout << (&coins[2] != &coins[3]) << endl;
```

- this works because array elements are always contiguous in memory

Comparing Pointers

- pointers may be compared using any of the relops
- what does each of the following lines print?

```
cout << (coins < &coins[1]) << endl;
cout << (coins < &coins[4]) << endl;
cout << (coins == &coins[0]) << endl;
cout << (&coins[2] == coins + 2) << endl;
cout << (&coins[2] != &coins[3]) << endl;
```

- this works because array elements are always contiguous in memory
- smaller-index elements have smaller addresses than larger-index ones

Pointer Parameters

- a pointer can easily be a parameter to a function

see program_9_11.cpp

Pointer Parameters

- a pointer can easily be a parameter to a function
- indeed, since an array name is a pointer, every array parameter you have used is really a pointer parameter

see program_9_11.cpp

Pointer Parameters

- a pointer can easily be a parameter to a function
- indeed, since an array name is a pointer, every array parameter you have used is really a pointer parameter
- arrays are not really passed by reference, they are passed by pointer (vectors, however, are passed by reference)

see program_9_11.cpp

Pointer Parameters

- a pointer can easily be a parameter to a function
- indeed, since an array name is a pointer, every array parameter you have used is really a pointer parameter
- arrays are not really passed by reference, they are passed by pointer (vectors, however, are passed by reference)
- because array syntax obscures the pointer syntax, let's look at a pure pass-by-pointer program not using arrays

see program_9_11.cpp

Pointer Arithmetic and Arrays

we use a for loop to iterate through arrays

```
double[] coins {0.01, 0.05, 0.1, 0.25, 0.5, 1.0};
for (size_t index = 0; index < 6; index++)
{
  cout << coins[index] << ' ';
}
```

Pointer Arithmetic and Arrays

we use a for loop to iterate through arrays

```
double[] coins {0.01, 0.05, 0.1, 0.25, 0.5, 1.0};
for (size_t index = 0; index < 6; index++)
{
  cout << coins[index] << ' ';
}
```

we can also use pointer arithmetic and pointer comparison

```
double* last_coin = &coins[5];
for (double* coin = coins; coin <= last_coin; coin++)
{
  cout << *coin << ' ';
}
```