CSE2050
Programming in a Second Language (C++)
Today’s lecture

- Accessing memory addresses
- Pointers and arrays
- Passing pointers to functions
Getting the address of a variable

- Each variable in the 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
```
9.2 Pointer Variables

Figure 9-4 illustrates the relationship between \texttt{ptr} and \texttt{x}.

As shown in Figure 9-4, \texttt{x}, which is located at memory address 0x7e00, contains the number 25. \texttt{ptr} contains the address 0x7e00. In essence, it "points" to the variable \texttt{x}.

The real benefit of pointers is that they allow you to indirectly access and modify the variable being pointed to. In Program 9-2, for instance, \texttt{ptr} could be used to change the contents of the variable \texttt{x}. This is done with the indirection operator, which is an asterisk (*).

When the indirection operator is placed in front of a pointer variable name, it dereferences the pointer. When you are working with a dereferenced pointer, you are actually working with the value the pointer is pointing to. This is demonstrated 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 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 << "Here is the value in x, printed twice:\n";
    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 << "Once again, here is the value in x:\n";
    cout << x << endl;     // Displays the contents of x
    cout << *ptr << endl;  // Displays the contents of x

    return 0;
}
```

Program Output

```
The value in x is 25
The address of x is 0x7e00
```
Chapter 9 Pointers

Creating and Using Pointer Variables

The definition of a pointer variable looks pretty much like any other definition. Here is an example:

```
int *ptr;
```

The asterisk in front of the variable name indicates that `ptr` is a pointer variable. The `int` data type indicates that `ptr` can be used to hold the address of an integer variable. The definition statement above would read "`ptr` is a pointer to an `int`." Some programmers prefer to define pointers with the asterisk next to the type name, rather than the variable name. For example, the previous definition shown above could be written as:

```
int* ptr;
```

This style of definition might visually reinforce the fact that `ptr`'s data type is not `int`, but `pointer-to-int`. Both definition styles are correct.

Program 9-2 demonstrates a very simple usage of a pointer: storing and printing the address of another variable.

In Program 9-2, two variables are defined: `x` and `ptr`. The variable `x` is an `int` and the variable `ptr` is a pointer to an `int`. The variable `x` is initialized with the value 25. The variable `ptr` is assigned the address of `x` with the following statement in line 10:

```
ptr = &x;
```

NOTE: In this definition, the word `int` does not mean that `ptr` is an integer variable. It means that `ptr` can hold the address of an integer variable. Remember, pointers only hold one kind of value: an address.

Program 9-2

```
// This program stores the address of a variable in a pointer.
#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
    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 0x7e00
```
Figure 9-4 illustrates the relationship between \textit{ptr} and \textit{x}.

As shown in Figure 9-4, \textit{x}, which is located at memory address 0x7e00, contains the number 25. \textit{ptr} contains the address 0x7e00. In essence, it "points" to the variable \textit{x}.

The real benefit of pointers is that they allow you to indirectly access and modify the variable being pointed to. In Program 9-2, for instance, \textit{ptr} could be used to change the contents of the variable \textit{x}. This is done with the indirection operator, which is an asterisk (*).

When the indirection operator is placed in front of a pointer variable name, it dereferences the pointer. When you are working with a dereferenced pointer, you are actually working with the value the pointer is pointing to. This is demonstrated in Program 9-3.

```
// 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 << "Here is the value in x, printed twice:
";
    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 << "Once again, here is the value in x:
";
    cout << x << endl;     // Displays the contents of x
    cout << *ptr << endl;  // Displays the contents of x

    return 0;
}
```
Accessing the value at a memory address

```cpp
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 << "Here is the value in x, printed twice:\n";
cout << x << endl;  // Displays the contents of x
cout << *ptr << endl;  // Displays the contents of x
```
Accessing the value at a memory address

Figure 9-4 illustrates the relationship between \texttt{ptr} and \texttt{x}. As shown in Figure 9-4, \texttt{x}, which is located at memory address 0x7e00, contains the number 25. \texttt{ptr} contains the address 0x7e00. In essence, it "points" to the variable \texttt{x}.

The real benefit of pointers is that they allow you to indirectly access and modify the variable being pointed to. In Program 9-2, for instance, \texttt{ptr} could be used to change the contents of the variable \texttt{x}. This is done with the indirection operator, which is an asterisk (*).

When the indirection operator is placed in front of a pointer variable name, it dereferences the pointer. When you are working with a dereferenced pointer, you are actually working with the value the pointer is pointing to. This is demonstrated in Program 9-3.

```cpp
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.
// Assign 100 to the location pointed to by ptr. This will actually assign 100 to x.
*ptr = 100;
```
Three uses of $\ast$

- As the multiplication operator, in statements such as
  \[ \text{distance} = \text{speed} \times \text{time}; \]
- In the definition of a pointer variable, such as
  \[ \text{int} \ *\text{ptr}; \]
- As the indirection operator, in statements such as
  \[ *\text{ptr} = 100; \]
Pointers and arrays

9.3 The Relationship Between Arrays and Pointers

CONCEPT:
Array names can be used as constant pointers, and pointers can be used as array names.

You learned in Chapter 7 that an array name, without brackets and a subscript, actually represents the starting address of the array. This means that an array name is really a pointer. Program 9-5 illustrates this by showing an array name being used with the indirection operator.

Because `numbers` works like a pointer to the starting address of the array, the first element is retrieved when `numbers` is dereferenced. So how could the entire contents of an array be retrieved using the indirection operator? Remember, array elements are stored together in memory, as illustrated in Figure 9-5.

It makes sense that if `numbers` is the address of `numbers[0]`, values could be added to `numbers` to get the addresses of the other elements in the array. It's important to know, however, that pointers do not work like regular variables when used in mathematical statements. In C++, when you add a value to a pointer, you are actually adding that value times the size of the data type being referenced by the pointer.

NOTE:
So far you've seen three different uses of the asterisk in C++:

• As the multiplication operator, in statements such as `distance = speed * time;`
• In the definition of a pointer variable, such as `int *ptr;`
• As the indirection operator, in statements such as `*ptr = 100;`

Program 9-5

```cpp
#include <iostream>
using namespace std;

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

Program Output

```
The first element of the array is 10
```

Array name is address of the array’s first element.
Pointers and arrays

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;

for (count = 0; count < NUM_COINS; count++)
    cout << *(coins + count) << " ";
cout << endl;
Pointers and arrays

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;

cout << "Here are the values in the coins array:\n";
for (count = 0; count < NUM_COINS; count++)
    cout << doublePtr[count] << " ";

Program 9-8
// This program uses the address of each element in the array.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    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;

    cout << "Here are the values in the coins array:\n";
    for (count = 0; count < NUM_COINS; count++)
        cout << doublePtr[count] << " ";

    return 0;
}
Incrementing/decrementing points: jumps sizeof(type)

5.3 The Relationship Between Arrays and Pointers

One to numbers, you are actually adding \(1 \times \text{sizeof(short)}\) to numbers. If you add two to numbers, the result is numbers + 2 \times \text{sizeof(short)}, and so forth. On a PC, this means the following are true, because short integers typically use two bytes:

* \(\text{*(numbers + 1)}\) is actually \(\text{*((numbers + 1) \times 2)}\)
* \(\text{*numbers}\)

This automatic conversion means that an element in an array can be retrieved by using its subscript or by adding its subscript to a pointer to the array. If the expression *numbers, which is the same as *\(\text{(numbers + 0)}\), retrieves the first element in the array, then *\(\text{(numbers + 1)}\) retrieves the second element. Likewise, *\(\text{(numbers + 2)}\) retrieves the third element, and so forth. Figure 9-6 shows the equivalence of subscript notation and pointer notation.

Program 9-6 shows the entire contents of the array being accessed, using pointer notation.

```c
1 // This program processes an array using pointer notation.
2 #include <iostream>
3 using namespace std;
4 (program continues)
```

NOTE: The parentheses are critical when adding values to pointers. The * operator has precedence over the + operator, so the expression *number + 1 is not equivalent to *(number + 1). *number + 1 adds one to the contents of the first element of the array, while *(number + 1) adds one to the address in number, then dereferences it.
What is wrong?

```cpp
float myFloat;
int *pint = &myFloat;
```
Passing pointers to functions

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

void getNumber(int *input)
{
    cout << "Enter an integer number: ";
    cin >> *input;
}
void doubleValue(int *val)
{
    *val *= 2;
}
```
Chapter 9 Pointers

Here is the definition of a function that uses a pointer parameter:

```c
void doubleValue(int *val)
{
    *val *= 2;
}
```

The purpose of this function is to double the variable pointed to by `val` with the following statement:

```
*val *= 2;
```

When `val` is dereferenced, the `*=` operator works on the variable pointed to by `val`. This statement multiplies the original variable, whose address is stored in `val`, by two. Of course, when the function is called, the address of the variable that is to be doubled must be used as the argument, not the variable itself. Here is an example of a call to the `doubleValue` function:

```
doubleValue(&number);
```

This statement uses the address operator (`&`) to pass the address of `number` into the `val` parameter. After the function executes, the contents of `number` will have been multiplied by two. The use of this function is illustrated in Program 9-11.

### Program 9-11

```c
// 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;
}
```

**Passing pointers to functions**
Passing pointers to functions: arrays

When the `getNumber` function is called in line 15, the address of the `number` variable in `function main` is passed as the argument. After the function executes, the value entered by the user is stored in `number`. Next, the `doubleValue` function is called in line 18, with the address of `number` passed as the argument. This causes `number` to be multiplied by two.

Pointer variables can also be used to accept array addresses as arguments. Either subscript or pointer notation may then be used to work with the contents of the array. This is demonstrated in Program 9-12.

```
// This program demonstrates that a pointer may be used as a parameter to accept the address of an array.

#include <iostream>
#include <iomanip>
using namespace std;

// Function prototypes
void getSales(double *, int);
double totalSales(double *, int);

int main()
{
  const int QTRS = 4;
  double sales[QTRS];

  // Get the sales data for all quarters.
  getSales(sales, QTRS);

  // Set the numeric output formatting.
  cout << fixed << showpoint << setprecision(2);

  // Display the total sales for the year.
  cout << "The total sales for the year are $";
  cout << totalSales(sales, QTRS) << endl;
  return 0;
}

//*****************************************************************
// Definition of getSales. This function uses a pointer to accept *
// the address of an array of doubles. The function asks the user *
// to enter sales figures and stores them in the array.           *
//*****************************************************************

void getSales(double *arr, int size)
{
  for (int count = 0; count < size; count++)
  {
    cout << "Enter the sales figure for quarter ";
    cout << (count + 1) << ": ";
    cin >> arr[count];
  }
}
```
Passing pointers to functions: arrays

```cpp
const int QTRS = 4;
double sales[QTRS];

// Get the sales data for all quarters.
getSales(sales, QTRS);

void getSales(double *arr, int size)
{
    for (int count = 0; count < size; count++)
    {
        cout << "Enter the sales figure for quarter ";
        cout << (count + 1) << " : ";
        cin >> arr[count];
    }
}
```

Array name is address of the array’s first element.