# CSCE 121
# Introduction to Program Design & Concepts
# Pointers

Dr. Tim McGuire

*Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.*

# 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:

```
int num = 42;
cout << &num; // prints address of num
              // 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

# Pointer Variables

- Definition:

```
int  *intptr;
```

- Read as:

  "**intptr** can hold the address of an int"

- Spacing in definition does not matter:

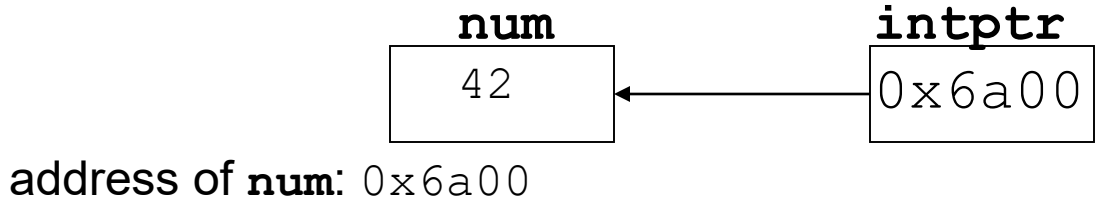
```
int * intptr;  // same as above
int*  intptr;  // same as above
```

# Pointer Variables

- Assigning an address to a pointer variable:

```
int *intptr;
intptr = &num;
```

- Memory layout:



# Pointer Variables

- Initialize pointer variables with the special value **`nullptr`**.
- In C++11, the **`nullptr`** key word was introduced to represent a pointer that does not point to a valid location (actually, location 0).
- Before C++11, a macro named NULL was used for this purpose.
- Here is an example of how you define a pointer variable and initialize it with the value **`nullptr`**:

```
int *ptr = nullptr;
```

# A Pointer Variable in a Program

pointer-example.cpp

```cpp
// This program stores the address of a variable in a pointer.
#include <iostream>
using namespace std;

int main()
{
   int x = 42;              // int variable
   int* ptr = nullptr;      // 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;
}
```

```
Wed Sep 29 mcguire Pointers-References $ g++ pointer-example.cpp
Wed Sep 29 mcguire Pointers-References $ ./a.out
The value in x is 42
The address of x is 0x7fffc57d1000
Wed Sep 29 mcguire Pointers-References $
```

# The Indirection Operator

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

```
int* intptr;
int x = 42;
intptr = &x;
cout << *intptr << endl;
```

# The Indirection Operator in a Program

indirection-operator.cpp

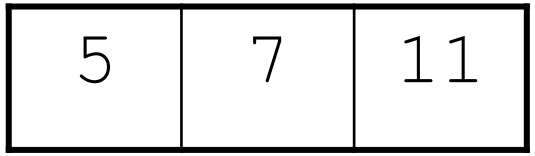
```
1  // This program demonstrates the use of the indi
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int x = 25;           // int variable
8     int* ptr = nullptr;      // Pointer variable,
9
10    ptr = &x;       // Store the address of x in ptr
11
12    // Use both x and ptr to display the value in x.
13    cout << "Here is the value in x, printed twice:\n";
14    cout << x << endl;     // Displays the contents of x
15    cout << *ptr << endl;  // Displays the contents of x
16
17    // Assign 42 to the location pointed to by ptr. This
18    // will actually assign 42 to x.
19    *ptr = 42;
20
21    // Use both x and ptr to display the value in x.
22    cout << "Once again, here is the value in x:\n";
23    cout << x << endl;     // Displays the contents of x
24    cout << *ptr << endl;  // Displays the contents of x
25    return 0;
26 }
```

```
Wed Sep 29 mcguire Pointers-References $ g++ indirection-operator.cpp
Wed Sep 29 mcguire Pointers-References $ ./a.out
Here is the value in x, printed twice:
25
25
Once again, here is the value in x:
42
42
Wed Sep 29 mcguire Pointers-References $
```

# The Relationship Between Arrays and Pointers

- Array name is starting address of array

```
int vals[] = {5, 7, 11};
```




starting address of **vals**: 0x6a00

```
cout << vals;          // displays 0x6a00
cout << vals[0];       // displays 5
```

# The Relationship Between Arrays and Pointers

- An array name can be used as a pointer constant:

```
int vals[] = {4, 7, 11};
cout << *vals;    // displays 4
```

- A pointer can be used as an array name:

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

# The Array Name Being Dereferenced in a Program

```
// This program shows an array name being dereferenced with the * operator.
#include <iostream>
using std::cout, std::endl;

int main()
{r
   int 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
```

# Pointers in Expressions

Given:

```
int vals[]={4,7,42}, *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 42
```

Must use **`( )`** as shown in the expressions

# Array Access

- Array elements can be accessed in many ways:

| Array access method | Example |
|---|---|
| array name and `[]` | `vals[2] = 42;` |
| pointer to array and `[]` | `valptr[2] = 42;` |
| array name and subscript arithmetic | `*(vals + 2) = 42;` |
| pointer to array and subscript arithmetic | `*(valptr + 2) = 42;` |

# Array Access

- In other words,

  If **`a`** is an array, and **`i`** is an index into the array, then

  **`a[i]`** and **`*(a+i)`** mean exactly the same thing

  (As a matter of fact, when the C++ compiler sees an expression like **`a[i]`**, it converts it to **`*(a+i)`** before translating it into machine code.)

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

# Pointer Arithmetic

# Pointer Arithmetic

- Operations on pointer variables:

| Operation | Example<br>`int vals[]={4,7,11};`<br>`int *valptr = vals;` |
|---|---|
| `++, --` | `valptr++; // points at 7`<br>`valptr--; // now points at 4` |
| `+, -` (pointer and `int`) | `cout << *(valptr + 2); // 11` |
| `+=, -=` (pointer and `int`) | `valptr = vals; // points at 4`<br>`valptr += 2;  // points at 11` |
| `-` (pointer from pointer) | `cout << valptr-val; // difference`<br>`//(number of ints) between valptr`<br>`// and val` |

# Using a Pointer to Display Array Contents

```
7     const int SIZE = 8;
8     int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 42};
9     int *numPtr = nullptr;  // Pointer
10    int count;              // Counter variable for loops
11
12    // Make numPtr point to the set array.
13    numPtr = set;
14
15    // Use the pointer to display the array contents.
16    cout << "The numbers in set are:\n";
17    for (count = 0; count < SIZE; count++)
18    {
19       cout << *numPtr << " ";
20       numPtr++;
21    }
22
23    // Display the array contents in reverse order.
24    cout << "\nThe numbers in set backward are:\n";
25    for (count = 0; count < SIZE; count++)
26    {
27       numPtr--;
28       cout << *numPtr << " ";
29    }
30    cout << endl;
31    return 0;
32 }
```

```
Thu Sep 30 mcguire Pointers-References $ g++ pointer-array.cpp
Thu Sep 30 mcguire Pointers-References $ ./a.out
The numbers in set are:
5 10 15 20 25 30 35 42
The numbers in set backward are:
42 35 30 25 20 15 10 5
```

# Initializing Pointers

# Initializing Pointers

- Can initialize at definition time:

```
int num, *numptr = &num;
int val[3], *valptr = val;
```

- Cannot mix data types:

```
double cost;
int *ptr = &cost; // won't work
```

- Can test for an invalid address for **`ptr`** with:

```
if (!ptr) ...
```

# Comparing Pointers

# 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
```

# CSCE 121
# Introduction to Program Design & Concepts
# Passing by Reference

Dr. Tim McGuire

*Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.*

# Using Reference Variables as Parameters

- A mechanism that allows a function to work with the original argument from the function call, not a copy of the argument
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to ‘return’ more than one value

# Passing by Reference

- A reference variable is an alias for another variable
- Defined with an ampersand (&)

```
void getDimensions(int&, int&);
```

- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters *by reference*

The & in the prototype indicates that the parameter is a reference variable

Here we are passing the variable by reference

```
1  // This program uses a reference variable as a function parameter.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype. The parameter is a reference variable.
6  void doubleTheNum(int &);
7
8  int main()
9  {
10    int value = 21;
11
12    cout << "In main, value is " << value << endl;
13    cout << "Now calling doubleNum..." << endl;
14    doubleTheNum(value);
15    cout << "Now back in main. value is " << value << endl;
16    return 0;
17 }
```

*(Program Continues)*

```
18
19  //*********************************************************
20  // Definition of doubleNum.                               *
21  // The parameter refVar is a reference variable. The value *
22  // in refVar is doubled.                                  *
23  //*********************************************************
24
25  void doubleTheNum (int &refVar)
26  {
27     refVar *= 2;
28  }
```

The & also appears in the function header indicating that theparameter is a reference variable

# Reference Variable Notes

- Each reference parameter must contain `&`
- Space between type and `&` is unimportant
- Must use `&` in both prototype and header
- Argument passed to reference parameter must be a variable – cannot be an expression or constant
- Use when appropriate – don’t use when argument should not be changed by function, or if function needs to return only 1 value

# Arrays as Function Arguments

# Arrays as Function Arguments

- To pass an array to a function, just use the array name:

```
showScores(tests);
```

- To define a function that takes an array parameter, use empty `[]` for array argument:

```
// function prototype
void showScores(int []);

// function header
void showScores(int tests[])
```

# Arrays as Function Arguments

- When passing an array to a function, it is common to pass array size so that function knows how many elements to process:

```
showScores(tests, ARRAY_SIZE);
```

- Array size must also be reflected in prototype, header:

```
// function prototype
void showScores(int [], int);

// function header
void showScores(int tests[], int size)
```

# Passing an Array to a Function

```
1  // This program demonstrates an array being passed to a function.
2  #include <iostream>
3  using namespace std;
4
5  void showValues(int [], int); // Function prototype
6
7  int main()
8  {
9       const int ARRAY_SIZE = 8;
10      int numbers[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
11
12      showValues(numbers, ARRAY_SIZE);
13      return 0;
14 }
15
16 //*************************************************
17 // Definition of function showValue.              *
18 // This function accepts an array of integers and  *
19 // the array's size as its arguments. The contents *
20 // of the array are displayed.                     *
21 //*************************************************
22
23 void showValues(int nums[], int size)
24 {
25      for (int index = 0; index < size; index++)
26          cout << nums[index] << " ";
27      cout << endl;
28 }
```

**Program Output**

```
5 10 15 20 25 30 35 40
```

# Modifying Arrays in Functions

- Array names in functions are like  reference variables – changes made to array in a function are reflected in actual array in calling function.

- Need to exercise caution that array is not inadvertently changed by a function.
- Use the **const** keyword in the parameter list to avoid that situation.

# Pointers as Function Parameters

# 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

```
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

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

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

# Pointer vs. Reference

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

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

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

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