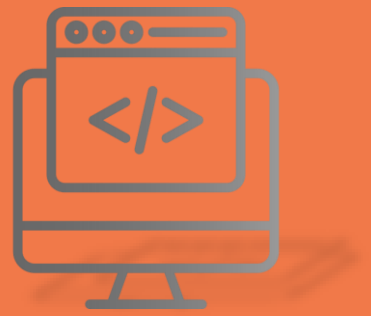
Tishk International University
Faculty of Applied Science
Information Technology Department

Global Variables & Pointers



Programming II

Lecture 3

Spring 2026

Course Code: IT118

Grade 1

Islam Abdulazeez
islam.abdulaziz@tiu.edu.iq

May 5, 2026

Outlines

- ✓ Local Variables
- ✓ Global Variables
- ✓ Pointers Basics
- ✓ Address & Dereference Operators
- ✓ Pointers and Arrays
- ✓ Pointer Arithmetic
- ✓ Pass by Reference

Learning Outcomes

- **At the end of today's session, you will be able to:**

  - ✓ Differentiate local and global variables.
  - ✓ Explain pointer concepts and memory addresses.
  - ✓ Apply & and * operators in C++.
  - ✓ Use pointers with arrays and functions.
  - ✓ Compare pass-by-value and pass-by-reference.
  - ✓ Develop simple programs using pointers and references.

Local and Global Variables

Local Variable	Global Variable
• Local variables are declares within a function. • It can be used only in function in which they declared. • Local variable are destroyed when control leave the function. • Local variables are used when the vales are to be used within a function.	• Global variables are declares outside any function • Global variables can be used in all function. • Global variable are destroyed when the program is terminated. • Global variables are used when values are to be shared among different functions.

Local Variable - Example

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

/*anotherFunction *
* This function displays the value of its local variable num. */
void anotherFunction(){
    int num = 20; // Local variable
    cout << "In anotherFunction, num is " << num << endl;
}

int main(){
    int num = 1; // Local variable

    cout << "In main, num is " << num << endl;
    anotherFunction();
    cout << "Back in main, num is still " << num << endl;
    return 0;
}
```

Output

```
In main, num is 1
In anotherFunction, num is 20
Back in main, num is still 1
```

Local Variable - Example

- local variables do not retain their values between function calls.

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


void showLocal() {
    int localNum = 5;
    cout << "localNum is " << localNum << endl;
    localNum = 99;
}

int main() {
    showLocal();
    showLocal();
    return 0;
}
```

Output

```
localNum is 5
localNum is 5
```

Global Variable

- A **global variable** is a variable that is declared outside all functions, and it can be used by any function in the program.

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

int num = 2; // Global variable

void anotherFunction() {
    cout << "In anotherFunction, num is " << num << endl;
    num = 50;
    cout << "But, it is now changed to " << num << endl;
}

int main() {
    cout << "In main, num is " << num << endl;
    anotherFunction();
    cout << "Back in main, num is " << num << endl;
    return 0;
}
```

Output

```
In main, num is 2
In anotherFunction, num is 2
But, it is now changed to 50
Back in main, num is 50
```

Global Variable

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

int num = 2; // Global variable

void anotherFunction() {
    cout << "In anotherFunction, num is " << num << endl;
    int num = 50;
    cout << "But, it is now changed to " << num << endl;
}

int main() {
    cout << "In main, num is " << num << endl;
    anotherFunction();
    cout << "Back in main, num is " << num << endl;
    return 0;
}
```

Output

```
In main, num is 2
In anotherFunction, num is 2
But, it is now changed to 50
Back in main, num is 2
```

Local and Global Variable Names

- Avoid using the same names for local and global variables
- In case by mistake, a function contains a local variable that has the same name as a global variable, the global variable will be inaccessible within the function.

Pointer

Pointer

- A **pointer** in programming is a variable that stores the memory address of another variable. Rather than holding a data value directly (like an integer or a character), a pointer holds the location (address) where the data is stored in memory.

- **Why Use Pointers?**

  - Efficiently pass large data to functions without copying.

  - Manipulate data directly in memory.

Address in C++

- To understand C++ pointer, we must understand how computers store data.
- When the variable is created in C++, it is assigned a space in computer **memory**.
- The **value** of this variable is stored in the assigned location.
- To know the location in the computer memory, C++ provides the **&** (ampersand, reference) operator. This operator returns the address where the variable is stored.

Declaring Pointer Variables

- Pointers declared like other types
- Add "*" before variable name,

  Produces "pointer to" that type
- The * must be placed before each variable.
- int *ptr;
- ptr holds pointers to int variables (i.e.

  ptr is a pointer to an integer)

```
int *x; // A pointer to integer
double *x; // A pointer to double
float *x; // A pointer to float
string *x; // A pointer to string
char *x; // A pointer to char
```

Declaring Pointer Variables

- Sets pointer variable **y** to "point to" int variable **x**
- Reference operator, & (ampersand): Determines "address of" variable **x**

```
int x = 5;
int *y = &x;
```

Same

```
int x = 5;
int *y;
y = &x;
```

Reference operator (&) and Dereference Operator (*)

- **&** (ampersand) operator returns the address of a variable.
- * operator returns the value stored in memory.
- For example: If a number variable is stored in the memory address **0x7ff7b3a74598** and it contains a value 7.
- The reference (&) operator gives the value **0x7ff7b3a74598**. While the dereference operator (*) gives the value 7.

Reference operator (&) and Dereference Operator (*)

```
int num = 9;
int* x = &num;

cout << "Memory address of number: " << x << endl;
cout << "Value stored at memory address " << *x << endl;
```

Output

```
Memory address of number: 0000005D240FF644
Value stored at memory address 9
```

Reference operator (&) and Dereference Operator (*)

```
int x = 3;
cout << "x= " << x << endl;
cout << "&x= " << &x << endl;

int* y = &x;
cout << "y= " << y << endl;
cout << "*y= " << *y << endl;
cout << "&y= " << &y << endl;
```

```
3
&x= 00000049739FFB14
y= 00000049739FFB14
*y= 3
&y= 000000A63AEFF968
```

Pointers and Arrays

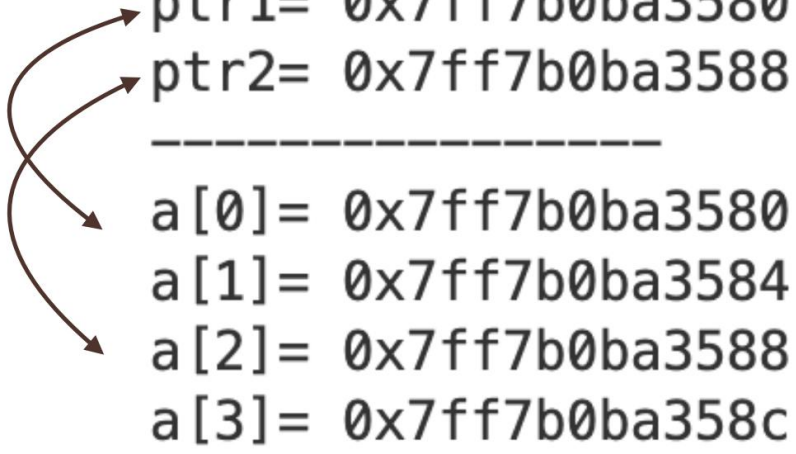
- Pointers are more efficient in handling array.
- Pointers provide an alternative way to access array's elements.

```
int numbers[4];

cout << numbers;        // points to the first element (&numbers[0])

cout << &numbers;       // points to the entire array as one block

cout << numbers[1];     // prints the value of the second element
```

Pointers and Arrays

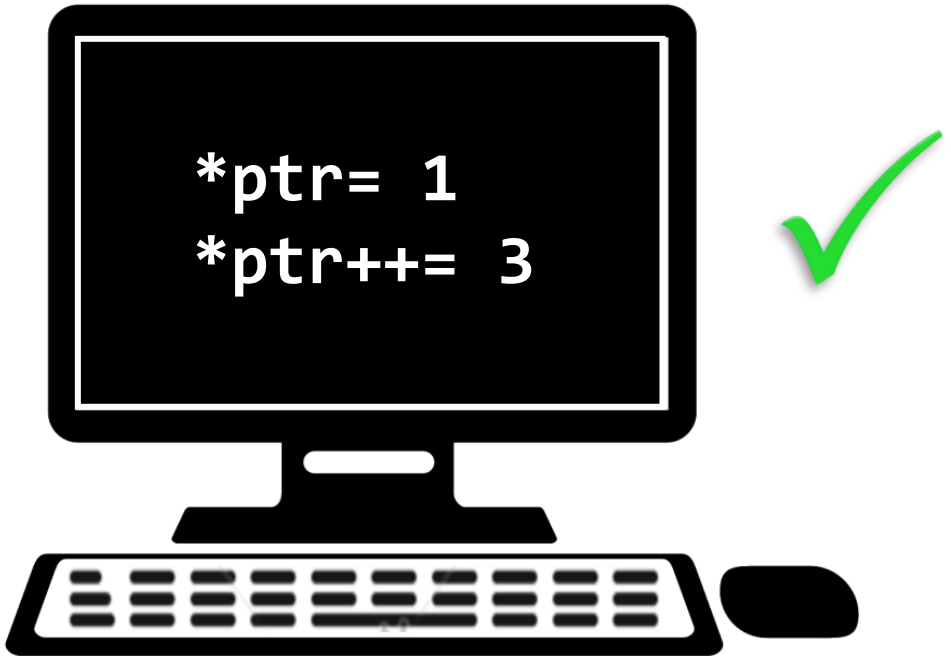
```
int *ptr1, *ptr2;
int a[4];
ptr1 = a;
ptr2 = &a[2];

cout<<"ptr1= "<<ptr1<<endl; // pointer to a[0]
cout<<"ptr2= "<<ptr2<<endl; // pointer to a[2]
cout<<"----------------"<<endl;
for(int i=0;i<4;i++){
    cout<<"a["<<i<<"]= "<<&a[i]<<endl;
}
```

Output



Pointer Arithmetic - Example

```
int* ptr;
int a[] = { 1, 3, 0, 9, 5 };

ptr = a;

cout << "*ptr= " << *ptr << endl;
ptr++;
cout << "*ptr++= " << *ptr++ << endl;
```



Pointer Arithmetic - Example

```
int* ptr;
int a[] = { 1, 3, 0, 9, 5 };

ptr = a;

for (int i = 0; i < 5; i++) {
    cout << "ptr= " << ptr << " - " << "*ptr= " << *ptr << endl;
    ptr++;
}
```

Output

```
ptr= 000000B520AFFBF8 - *ptr= 1
ptr= 000000B520AFFBFC - *ptr= 3
ptr= 000000B520AFFC00 - *ptr= 0
ptr= 000000B520AFFC04 - *ptr= 9
ptr= 000000B520AFFC08 - *ptr= 5
```

Pass by Reference

- Pointer variable can pass as a function argument and function can return pointer.
- There are two approaches to passing argument to a function:
  - Call by value (Previous lecture)
  - Call by reference
- **Pass by Reference:** A mechanism that allows a function to work with the **original variable** from the function call, not a copy of its value.
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to 'return' more than 1 value
- Defined with an ampersand (**&**)

Pass by Reference

- **Pass-by-Reference:** Pointers can be used to pass variables to functions by reference, which means that the function can modify the variable's value directly in memory. This can be useful for functions that need to modify the input variables.

```
void myFunc(int& d) {
    d = d + 3;
}

int main() {
    int x = 8;

    cout << "x before calling function = " << x << endl;
    myFunc(x);
    cout << "x after calling function = " << x << endl;

    return 0;
}
```

Output

```
x before calling function = 8
x after calling function = 11
```

Pass by Reference (Ex.)

Pass by Value

```
void func1(double a, int b)
{
    cout << a << " " << b << endl;
    a = 0.0;
    b = 10;
    cout << a << " " << b << endl;
}

int main()
{
    int x = 0;
    double y = 1.5;
    cout << x << " " << y << endl;
    func1(y, x);
    cout << x << " " << y << endl;
    return 0;
}
```

Output

```
0 1.5
1.5 0
0 10
0 1.5
```

Pass by Reference

```
void func1(double &a, int &b)
{
    cout << a << " " << b << endl;
    a = 0.0;
    b = 10;
    cout << a << " " << b << endl;
}

int main()
{
    int x = 0;
    double y = 1.5;

    cout << x << " " << y << endl;
    func1(y, x);
    cout << x << " " << y << endl;

    return 0;
}
```

Output

```
0 1.5
1.5 0
0 10
10 0
```

Pass by Reference (Ex.)

Call by value

```
void myFunc(int a, int b) {
    a = 10;
    b = 19;
}

int main() {

    int x = 12, y = 5;

    cout << "x before calling = " << x << endl;
    cout << "y before calling = " << y << endl;

    myFunc(x, y);

    cout << "x after calling = " << x << endl;
    cout << "y after calling = " << y << endl;

    return 0;
}
```

Output

```
x before calling = 12
y before calling = 5
x after calling = 12
y after calling = 5
```

Changing values using pointers

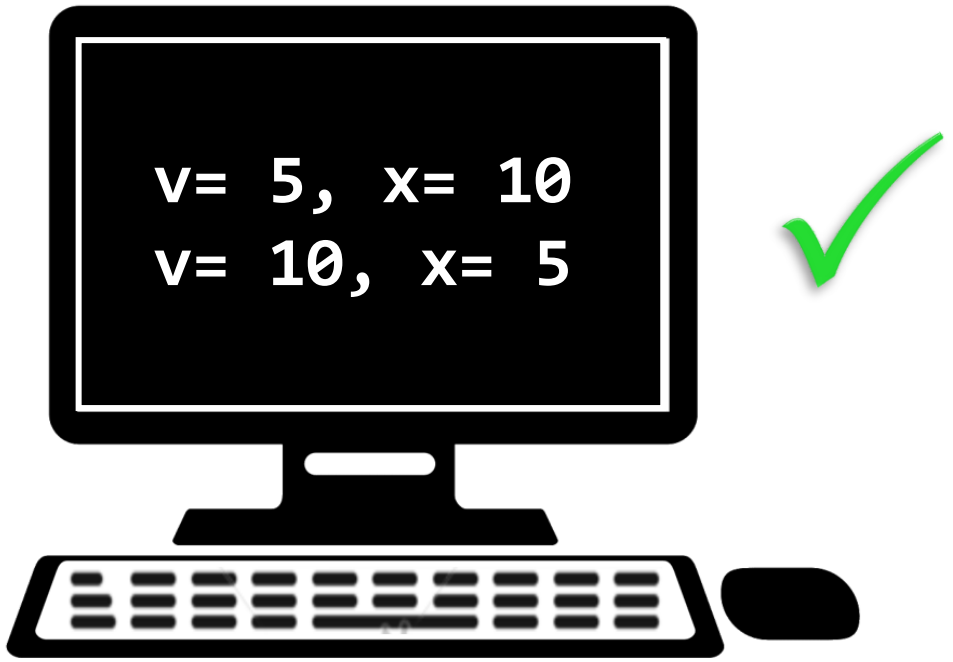
```
void myFunc(int* a, int* b) {
    *a = 10;
    *b = 19;
}

int main() {

    int x = 12, y = 5;

    cout << "x before calling = " << x << endl;
    cout << "y before calling = " << y << endl;

    myFunc(&x, &y);

    cout << "x after calling = " << x << endl;
    cout << "y after calling = " << y << endl;

    return 0;
}
```

Output

```
x before calling = 12
y before calling = 5
x after calling = 10
y after calling = 19
```

Swap Function with Reference Variables

```
void swap_func(int& a, int& b) {

    int temp;
    temp = a;
    a = b;
    b = temp;
}

int main() {

    int v = 5, x = 10;

    cout << "v= " << v << ", x= " << x << endl;

    swap_func(v, x);

    cout << "v= " << v << ", x= " << x << endl;

    return 0;
}
```



Reference Variable Notes

- Each reference parameter must contain **&**
- Argument passed to reference parameter must be a variable

  (**cannot be an expression or constant**).

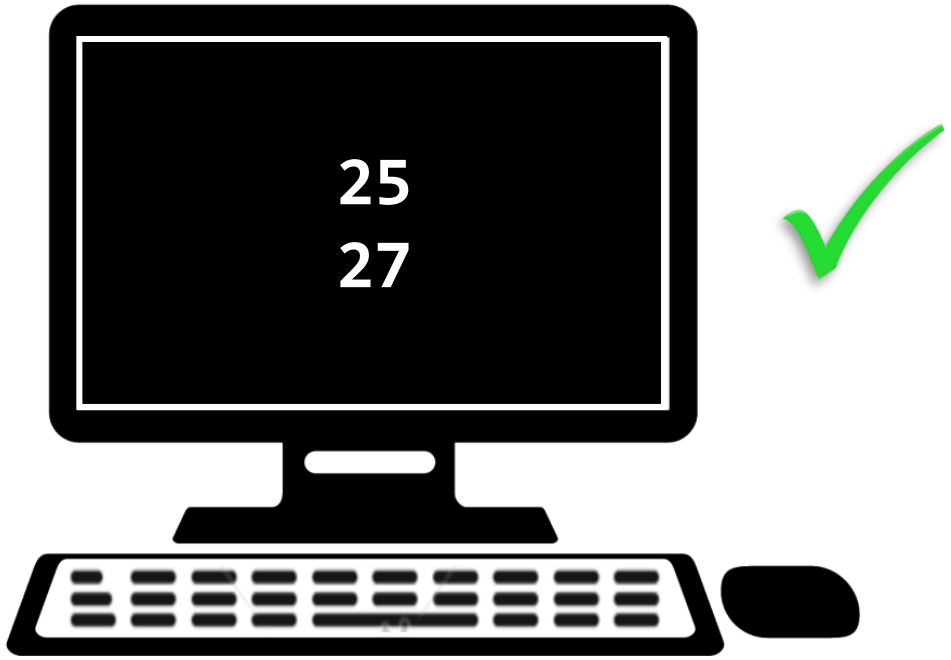
OK

Wrong

```
void modValue(int& x) {
    x = x + 2;
}

int main() {

    int a = 5;
    const int b = 10;

    modValue(a);   // It is OK

    modValue(b);   // WRONG... Gives you Error

    return 0;
}
```

Returning Multiple Values

```
void sqcube(int& x, int& y) {
    x = x * x;
    y = y * y * y;
}

int main() {
    int x = 5;
    int y = 3;

    sqcube(x, y);

    cout << x << endl;
    cout << y << endl;

    return 0;
}
```



Exercise #1

- Find the maximum number in the following array, using function.

  myArr[]={8, 2, 5, 1, 7, 4, 9, 3};

```
int max(int A[], int size) {
    int max = A[0];

    for (int i = 0; i < size; i++) {
        if (A[i] > max) {
            max = A[i];
        }
    }
    return max;
}

int main() {
    int myArr[] = { 8, 2, 5, 1, 7, 4, 9, 3 };

    int size = sizeof(myArr) / sizeof(myArr[0]);

    cout << "Max= " << max(myArr, size) << endl;

    return 0;
}
```

Exercise #2

- Find the minimum and maximum numbers in the following array, using function.

  myArr[]={8, 2, 5, 1, 7, 4, 9, 3};

```
void minmax(int A[], int size, int& min, int& max) {
    min = max = A[0];

    for (int i = 0; i < size; i++) {
        if (A[i] > max) {
            max = A[i];
        }

        if (A[i] < min) {
            min = A[i];
        }
    }
}

int main() {
    int myArr[] = { 8, 2, 5, 1, 7, 4, 9, 3 };
    int min, max;

    int size = sizeof(myArr) / sizeof(myArr[0]);

    minmax(myArr, size, min, max);

    cout << "Min= " << min << endl;
    cout << "Max= " << max << endl;

    return 0;
}
```

Activities and Next Lecture's Topic

Activities

- Review this lecture note
- Practice

Next Lecture's Topic

- Vector and Deque

References

- Gaddis, T. (2014). Starting out with C++: Early objects (7th ed.). Pearson Education.

Thank You!