Arguments passed by value

- **Pass by value**: when an argument is passed to a function, its value is *copied* into the parameter.
- It is implemented using variable initialization (behind the scenes):
  ```
  int param = argument;
  ```
- Changes to the parameter in the function body do not affect the value of the argument in the call
- The parameter and the argument are stored in separate variables; separate locations in memory.

Example: Pass by Value

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

void changeMe(int);

int main() {
    int number = 12;
    cout << "number is " << number << endl;
    changeMe(number);
    cout << "Back in main, number is " << number << endl;
    return 0;
}

void changeMe(int myValue) {
    myValue = 200;
    cout << "myValue is " << myValue << endl;
}
```

Output:

```
number is 12
myValue is 200
Back in main, number is 12
```

Parameter passing by Reference

- **Pass by reference**: when an argument is passed to a function, the function has direct access to the original argument (no copying).
- Pass by reference in C++ is implemented using a reference parameter, which has an ampersand (&) in front of it:
  ```
  void changeMe (int &myValue);
  ```
- A reference parameter acts as an *alias* to its argument, it is NOT a separate storage location.
- Changes to the parameter in the function **DO** affect the value of the argument
Example: Pass by Reference

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

void changeMe(int &);

int main() {
    int number = 12;
    cout << "number is " << number << endl;
    changeMe(number);
    cout << "Back in main, number is " << number << endl;
    return 0;
}

void changeMe(int &myValue) {
    myValue = 200;
    cout << "myValue is " << myValue << endl;
}
```

Output: number is 12
myValue is 200
Back in main, number is 200

Arrays

- An **array** is:
  - A series of elements of the same type
  - placed in contiguous memory locations
  - that can be individually referenced by adding an index to a unique identifier.
- To declare an array:
  - datatype is the type of the elements
  - identifier is the name of the array
  - size is the number of elements (constant)

```cpp
int numbers[5];
```

Array initialization

- To specify contents of the array in the definition:
  - creates an array of size 3 containing the specified values.
    ```cpp
    float scores[3] = {86.5, 92.1, 77.5};
    ```
  - creates an array containing the specified values followed by 7 zeros (partial initialization).
    ```cpp
    float scores[10] = {86.5, 92.1, 77.5};
    ```
  - creates an array of size 3 containing the specified values (size is determined from list).
    ```cpp
    float scores[] = {86.5, 92.1, 77.5};
    ```

Array access

- to access the value of any of the elements of the array individually as if it was a normal variable:
  ```cpp
  scores[2] = 89.5;
  ```
- scores[2] is a variable of type float
- use it anywhere a float variable can be used.
- rules about subscripts:
  - always start at 0, last subscript is size-1
  - must have type int but can be any expression
- watchout: brackets used both to declare the array and to access elements.
Arrays: operations

- **Valid operations over entire arrays:**
  - function call: `myFunc(scores, x);`
- **Invalid operations over structs:**
  - assignment: `array1 = array2;`
  - comparison: `array1 == array2`
  - output: `cout << array1;`
  - input: `cin >> array2;`

- Must do these element by element, probably using a for loop

Example: Processing arrays

Computing the average of an array of scores:

```c++
const int NUM_SCORES = 8;
int scores[NUM_SCORES];
cout << "Enter the " << NUM_SCORES
    << " programming assignment scores: " << endl;
for (int i=0; i < NUM_SCORES; i++) {
    cin >> scores[i];
}
int total = 0; //initialize accumulator
for (int i=0; i < NUM_SCORES; i++) {
    total = total + scores[i];
}
double average =
    static_cast<double>(total) / NUM_SCORES;
```

Example: Partially filled arrays

```c++
int sumList (int list[], int size) {//sums elements in list array
    int total = 0;
    for (int i=0; i < size; i++) {
        total = total + list[i];
    }
    return total;
}
const int CAPACITY = 100;
int main() {
    int scores[CAPACITY];
    int count = 0;              //tracks number of elems in array
    cout << "Enter the programming assignment scores:" << endl;
    cout << "Enter -1 when finished" << endl;
    int score;
    cin >> score;
    while (score != -1 && count < CAPACITY) {
        scores[count] = score;
        count++;
        cin >> score;
    }
    int sum = sumList(scores, count);
    pass count, not CAPACITY
}
```

Arrays as parameters

- In the **function definition**, the parameter type is a variable name with an empty set of brackets: `[]`
- Do NOT give a size for the array inside `[]`
  ```c++
  void showArray(int values[], int size)
  ```
- In the **prototype**, empty brackets go after the element datatype.
  ```c++
  void showArray(int[], int)
  ```
- In the **function call**, use the variable name for the array.
  ```c++
  showArray(numbers, 5)
  ```

- An array is **always** passed by reference.
Multidimensional arrays

- **multidimensional array**: an array that is accessed by more than one index

```c
int table[2][5]; // 2 rows, 5 columns
table[0][1] = 10; // puts 10 in first row, second column
```

- **Initialization**:

```c
int a[4][3] = {4,6,3,12,7,15,41,32,81,52,11,9};
- First row: 4,6,3
- Second row: 12, 7, 15
- etc.
```

Multidimensional arrays

- when using a 2D array as a parameter, you must specify the number of columns:

```c
void myfunction(int vals[][3], int rows) {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < 3; ++j)
            cout << vals[i][j] << " ";
        cout << "n";
    }
}
```

```
int main() {
    int a[4][3] = {4,6,3,12,7,15,41,32,81,52,11,9};
    ...
    myfunction(a,4);
    ...
}
```

Structures

- A structure stores a collection of objects of various types
- Each element in the structure is a member, and is accessed using the dot member operator.

```c
struct Student {
    int idNumber; // Defines a new data type
    string name;
    int age;
    string major;
};
```

```
Student student1, student2; // Defines new variables
student1.name = "John Smith";
Student student3 = {123456,"Ann Page",22,"Math"};
```

Structures: operations

- **Valid operations over entire structs**:
  - assignment: student1 = student2;
  - function call: myFunc(gradStudent,x);

<table>
<thead>
<tr>
<th>void myFunc(Student, int); //prototype</th>
</tr>
</thead>
</table>

- **Invalid** operations over structs:
  - comparison: student1 == student2
  - output: cout << student1
  - input: cin >> student2
  - Must do these member by member
Arrays of Structures

- You can store values of structure types in arrays.
  
  Student roster[40];  //holds 40 Student structs

- Each student is accessible via the subscript notation.
  
  roster[0] = student1;

- Members of structure accessible via dot notation
  
  cout << roster[0].name << endl;

Pointers

- **Pointer**: a variable that stores the address of another variable, providing indirect access to it.

- The **address operator** (&) returns the address of a variable.
  
  int x;
  cout << &x << endl;  // 0xbffffb0c

- An asterisk is used to define a pointer variable
  
  int *ptr;

- “ptr is a pointer to an int”. It can contain addresses of int variables.
  
  ptr = &x;

Pointers as Function Parameters

- Use pointers to implement pass by reference.

  ```cpp
  //prototype: void changeVal(int *);
  void changeVal (int *val) {
    *val = *val * 11;
  }
  
  int main() {
    int x;
    cout << "Enter an int " << endl;
    cin >> x;
    changeVal(&x);
    cout << x << endl;
  }
  ```

- How is it different from using reference parameters?
Pointers and Arrays

- You can treat an array variable as if it were a pointer to its first element.

```
int numbers[] = {10, 20, 30, 40, 50};
cout << "first: " << numbers[0] << endl;
cout << "first: " << *numbers << endl;
cout << &(numbers[0]) << endl;
cout << numbers << endl;
```

Output:
```
first: 10
first: 10
0xbffffb00
0xbffffb00
```

Pointer Arithmetic

- When you add a value \( n \) to a pointer, you are actually adding \( n \) times the size of the data type being referenced by the pointer.

```
int numbers[] = {10, 20, 30, 40, 50};
// sizeof(int) is 4.
// Let us assume numbers is stored at 0xbffffb00
// Then numbers+1 is really 0xbffffb00 + 1*4, or 0xbffffb04
// And numbers+2 is really 0xbffffb00 + 2*4, or 0xbffffb08
// And numbers+3 is really 0xbffffb00 + 3*4, or 0xbffffb0c
```

```
cout << "second: " << numbers[1] << endl;
cout << "second: " << *(numbers+1) << endl;
cout << "size: " << sizeof(int) << endl;
cout << numbers << endl;
cout << numbers+1 << endl;
```

Output:
```
second: 20
second: 20
size: 4
0xbffffb00
0xbffffb04
```

- Note: array[index] is equivalent to *(array + index)

Pointers and Arrays

- Pointer operations can be used with array variables.

```
int list[10];
cin >> *(list+3);
```

- Subscript operations can be used with pointers.

```
int list[] = {1,2,3};
int *ptr = list;
cout << ptr[2];
```

Pointers to structures

- We can define pointers to structures

```
Student s1 = {12345,"Jane Doe", 18, "Math"};
Student *ptr = &s1;
```

- To access the members via the pointer:

```
cout << *ptr.name << end; // ERROR: *(ptr.name)
```

- Dot operator has higher precedence, so use ():

```
cout << *(ptr).name << end;
```

- Or equivalently, use ->:

```
cout << ptr->name << end;
```