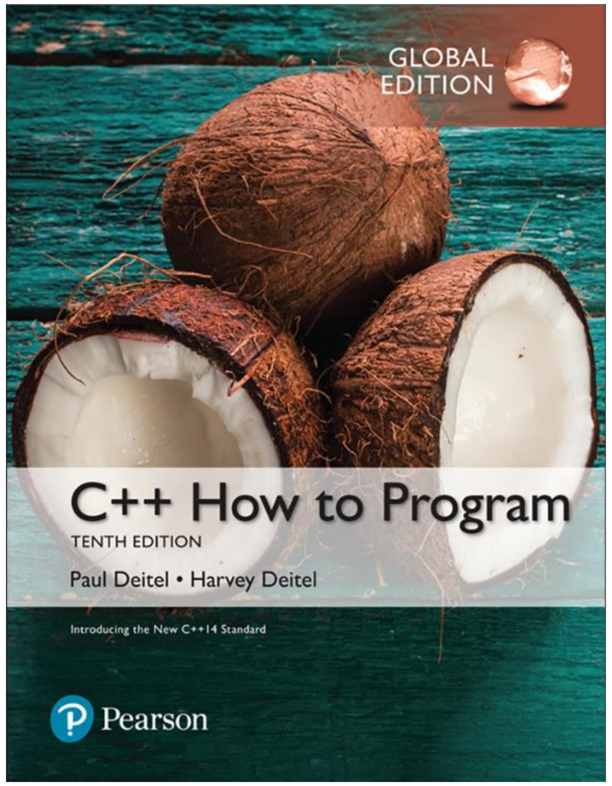
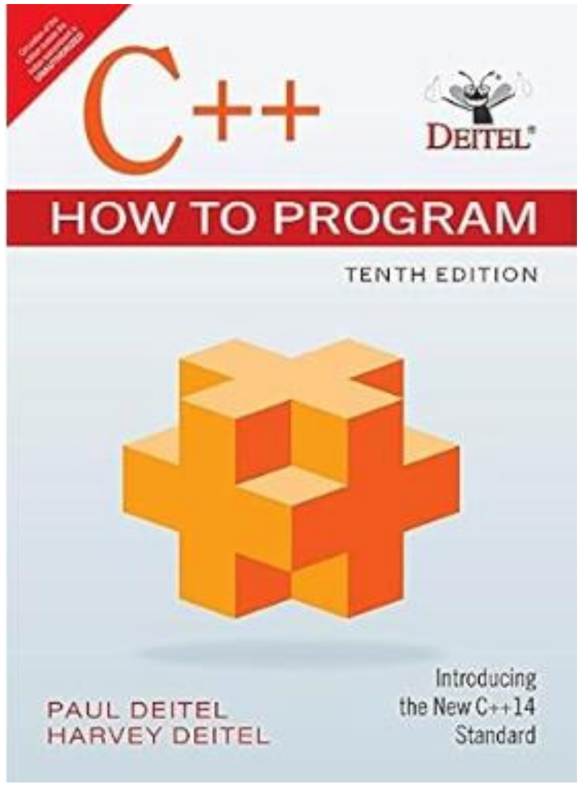
# Object-Oriented Programming

Lecture 6: Class Templates Array and vector
Pointers

# OOP Book

C++ How to Program (10th Edition)
HARVEY M. DEITEL PAUL DEITEL (Author)


GLOBAL EDITION
C++ How to Program
TENTH EDITION
Paul Deitel • Harvey Deitel
Introducing the New C++14 Standard
Pearson

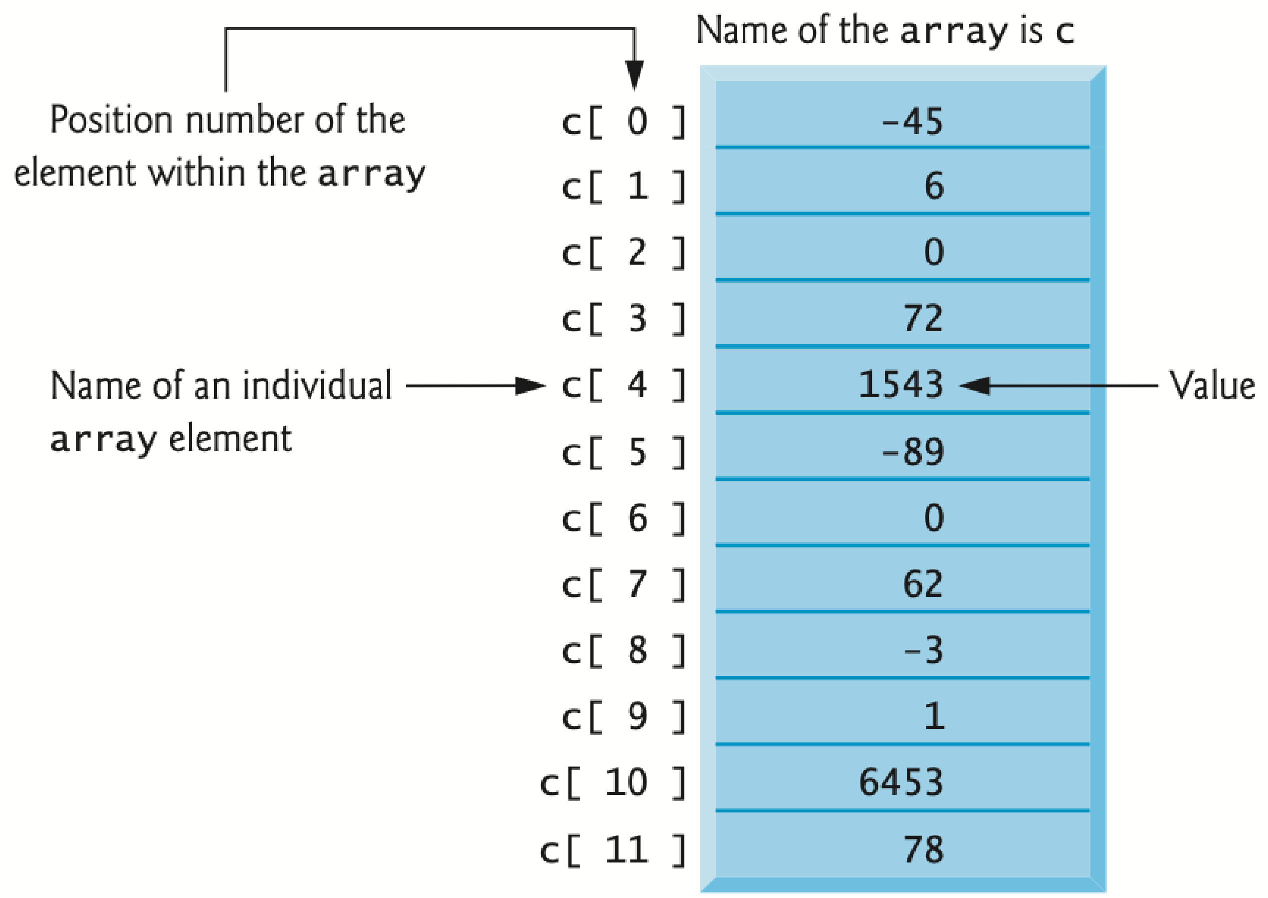


## Array

# Array

- An array is a contiguous group of memory locations that all have the same type.
- To refer to a particular location or element in the array, we specify the name of the array and the position number of the particular element in the array.
- Declaring array:

  array<type, size> arrayName;
  data_type arrayName[size];
- Accessing array elements
  Using square brackets: element = array_nan
  Important: Ensure index is within valid
 range (0 to size-1).



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

int main() {
   array<int, 5> n; // n is an array of 5 int values

   // initialize elements of array n to 0
   for (size_t i{0}; i < n.size(); ++i) {
      n[i] = 0; // set element at location i to 0
   }

   cout << "Element" << setw(10) << "Value" << endl;

   // output each array element's value
   for (size_t j{0}; j < n.size(); ++j) {
      cout << setw(7) << j << setw(10) << n[j] << endl;
   }
}
```

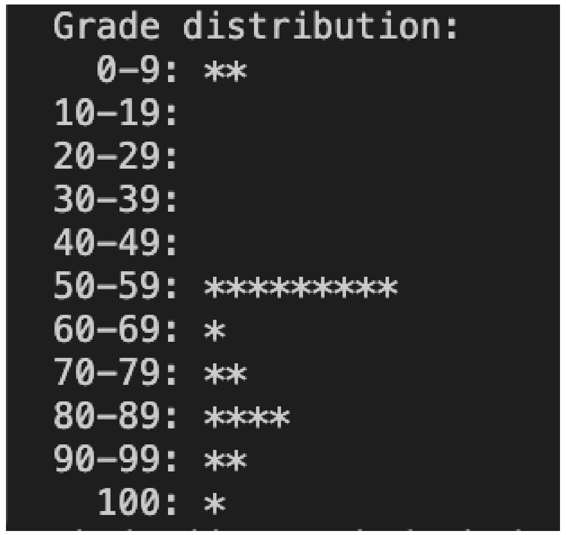
```
array<int, 5> n{32, 27, 64, 18, 95}; // list initializer

const size_t arraySize{5}; // must initialize in declaration

array<int, arraySize> values; // array values has 5 elements
```

```
Element     Value
      0         0
      1         0
      2         0
      3         0
      4         0
```

```
Element     Value
      0        32
      1        27
      2        64
      3        18
      4        95
```

size_t: This is an unsigned integer type that is typically used for array indexing and loop counting. It is guaranteed to be big enough to contain the size of the largest possible object of any type (including arrays).

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

int main() {
    const size_t arraySize{11};
    array<unsigned int, arraySize> n{2, 0, 0, 0, 0, 9, 1, 2, 4, 2, 1};

    cout << "Grade distribution:" << endl;

    // for each element of array n, output a bar of the chart
    for (size_t i{0}; i < n.size(); ++i) {
        // output bar labels ("0-9:", ..., "90-99:", "100:")
        if (0 == i) {
            cout << "  0-9: ";
        } else if (10 == i) {
            cout << "  100: ";
        } else {
            cout << setw(2) << i * 10 << "-" << setw(2) << (i * 10) + 9 << ": ";
        }

        // print bar of asterisks
        for (unsigned int stars{0}; stars < n[i]; ++stars) {
            cout << '*';
        }

        cout << endl; // start a new line of output
    }

    return 0;
}
```



```
Grade distribution:
  0-9: **
10-19: 
20-29: 
30-39: 
40-49: 
50-59: *********
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

## Using the Elements of an array as Counters

random number generator provided by the C++ Standard Library

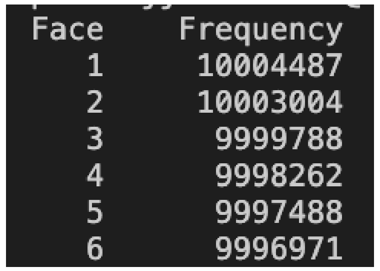
template class that generates random integers.

seeding the random number generator

function call returns the current time

specifies the type of number to generate

```
#include <iostream>
#include <iomanip>
#include <array>
#include <random>
#include <ctime>

using namespace std;

int main() {
    // Random number engine and distribution
    default_random_engine engine(static_cast<unsigned int>(time(0)));
    uniform_int_distribution<unsigned int> randomInt(1, 6);

    const size_t arraySize{7}; // Size of the array
    array<unsigned int, arraySize> frequency{}; // Initialize frequencies to 0

    // Roll die 60,000,000 times and record the frequency
    for (unsigned int roll{1}; roll <= 60000000; ++roll) {
        ++frequency[randomInt(engine)];
    }

    // Display the frequency of each face
    cout << "Face" << setw(13) << "Frequency" << endl;

    for (size_t face{1}; face < frequency.size(); ++face) {
        cout << setw(4) << face << setw(13) << frequency[face] << endl;
    }

    return 0;
}
```



```
Face    Frequency
   1     10004487
   2     10003004
   3      9999788
   4      9998262
   5      9997488
   6      9996971
```

## Using arrays to Summarize Survey Results

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

int main() {
   // define array sizes
   const size_t responseSize{20}; // size of array responses
   const size_t frequencySize{6}; // size of array frequency

   // place survey responses in array responses
   const array<unsigned int, responseSize> responses{
      1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5};

   // initialize frequency counters to 0
   array<unsigned int, frequencySize> frequency{};

   // for each answer, select responses element and use that value
   // as frequency subscript to determine element to increment
   for (size_t answer{0}; answer < responses.size(); ++answer) {
      ++frequency[responses[answer]];
   }

   cout << "Rating" << setw(12) << "Frequency" << endl;

   // output each array element's value
   for (size_t rating{1}; rating < frequency.size(); ++rating) {
      cout << setw(6) << rating << setw(12) << frequency[rating] << endl;
   }
}
```

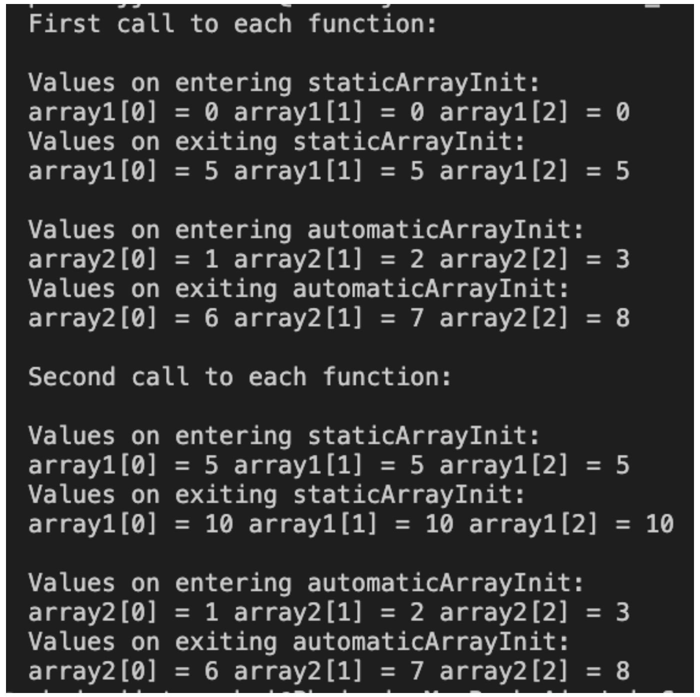
```
Rating   Frequency
     1           3
     2           5
     3           7
     4           2
     5           3
```

## Static Local arrays and Automatic Local arrays

**Static Storage Duration**
An array with static storage duration is allocated when the program starts and deallocated when the program ends. Its lifetime is the entire execution of the program. This means:
- **Initialization**: A static array is initialized only once, and this occurs before program execution begins. If explicitly initialize a static array, it is automatically initialized to zero.
- **Memory Location**: Static arrays are typically stored in a specific area of memory (often called the "data segment").
- **Persistence**: They retain their value between function calls. If a function with a static array is called multiple times, the array keeps its data from the previous calls.
- **Usage**: Static arrays are useful when you need to maintain state between function calls, but they should be used judiciously to avoid excessive memory usage or unintended side-effects between calls.

**Automatic Storage Duration**
An array with automatic storage duration is allocated when the block in which it is defined is entered and deallocated when that block is exited. In simpler terms:
- **Lifetime**: The lifetime of an automatic array is limited to the scope of the block in which it's defined, typically a function.
Initialization: If not explicitly initialized, its values are indeterminate.
- **Memory Location**: They are usually stored on the stack, making them faster to allocate and deallocate.
- **Reset on Every Call**: Each time a function with an automatic array is called, the array is re-created; it does not retain values between calls. This makes them suitable for functions that need a fresh array on each call.

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

void staticArrayInit(); // function prototype
void automaticArrayInit(); // function prototype

const size_t arraySize{3};

int main() {
    cout << "First call to each function:\n";
    staticArrayInit();
    automaticArrayInit();

    cout << "\n\nSecond call to each function:\n";
    staticArrayInit();
    automaticArrayInit();
    cout << endl;
}
```



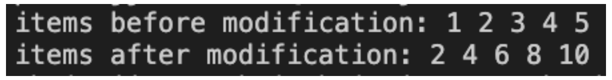
```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0 array1[1] = 0 array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5 array1[1] = 5 array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1 array2[1] = 2 array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5 array1[1] = 5 array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10 array1[1] = 10 array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1 array2[1] = 2 array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8
```

```cpp
// function to demonstrate a static local array
void staticArrayInit() {
    // initializes elements to 0 first time function is called
    static array<int, arraySize> array1; // static local array

    cout << "\nValues on entering staticArrayInit:\n";

    // output contents of array1
    for (size_t i{0}; i < array1.size(); ++i) {
        cout << "array1[" << i << "] = " << array1[i] << " ";
    }

    cout << "\nValues on exiting staticArrayInit:\n";

    // modify and output contents of array1
    for (size_t j{0}; j < array1.size(); ++j) {
        cout << "array1[" << j << "] = " << (array1[j] += 5) << " ";
    }
}

// function to demonstrate an automatic local array
void automaticArrayInit() {
    // initializes elements each time function is called
    array<int, arraySize> array2{1, 2, 3}; // automatic local array

    cout << "\n\nValues on entering automaticArrayInit:\n";

    // output contents of array2
    for (size_t i{0}; i < array2.size(); ++i) {
        cout << "array2[" << i << "] = " << array2[i] << " ";
    }

    cout << "\nValues on exiting automaticArrayInit:\n";

    // modify and output contents of array2
    for (size_t j{0}; j < array2.size(); ++j) {
        cout << "array2[" << j << "] = " << (array2[j] += 5) << " ";
    }
}
```

## Range-Based for Statement

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

int main() {
   array<int, 5> items{1, 2, 3, 4, 5};

   // Display items before modification
   cout << "items before modification: ";
   for (int item : items) {
      cout << item << " ";
   }

   // Multiply the elements of items by 2
   for (int& itemRef : items) {
      itemRef *= 2;
   }

   // Display items after modification
   cout << "\nitems after modification: ";
   for (int item : items) {
      cout << item << " ";
   }

   cout << endl;
}
```

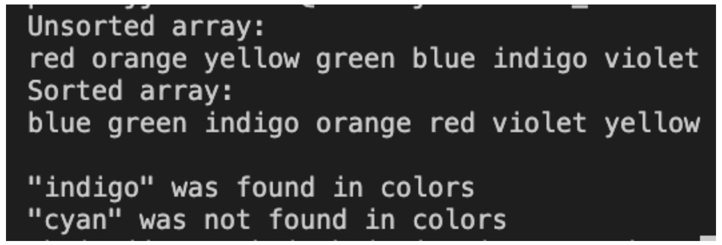
reference, not a pointer
work with it as if it were the original variable



```
items before modification: 1 2 3 4 5
items after modification: 2 4 6 8 10
```

## Sorting and Searching arrays

```
#include <iostream>
#include <iomanip>
#include <array>
#include <string>
#include <algorithm> // contains sort and binary_search
using namespace std;

int main() {
   const size_t arraySize{7}; // size of array colors
   array<string, arraySize> colors{"red", "orange", "yellow", "green",
"blue", "indigo", "violet"};

   // Output original array
   cout << "Unsorted array:\n";
   for (const string& color : colors) {
      cout << color << " ";
   }
```



```
Unsorted array:
red orange yellow green blue indigo violet
Sorted array:
blue green indigo orange red violet yellow

"indigo" was found in colors
"cyan" was not found in colors
```

```
   // Sort contents of colors
   sort(colors.begin(), colors.end());

   // Output sorted array

   cout << "\nSorted array:\n";

   for (const string& item : colors) {

      cout << item << " ";

   }

   // Search for "indigo" in colors

   bool found = binary_search(colors.begin(), colors.end(), "indigo");

   cout << "\n\n\"indigo\" " << (found ? "was" : "was not") << " found in colors"
<< endl;


   // Search for "cyan" in colors
   found = binary_search(colors.begin(), colors.end(), "cyan");
   cout << "\"cyan\" " << (found ? "was" : "was not") << " found in colors" <<
endl;

}
```

## Sorting Array

```
sort(colors.begin(), colors.end());
```

returns an iterator pointing to the first element and "past-the-end" of the colors array.

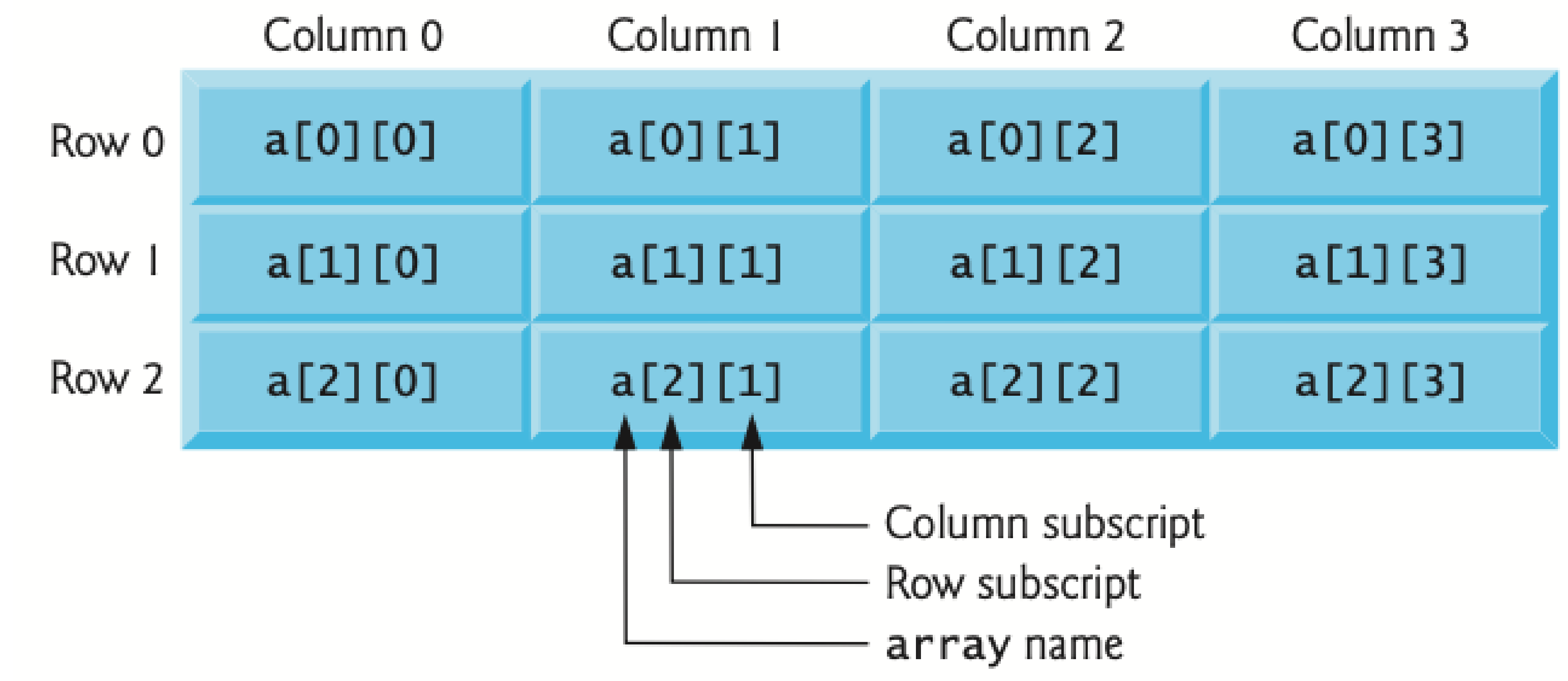
a conceptual position in a container that is one position beyond the last actual element.

**sort(startIterator, endIterator).** It sorts the elements in the range [startIterator, endIterator).
Iterators:
**startIterator** points to the beginning of the range to be sorted.
**endIterator** points to one past the end of the range to be sorted (the "past-the-end" iterator).
Sorting Order: By default, sort arranges the elements in ascending order. It does this by comparing pairs of elements using the < (less than) operator.
To sort in descending order,

```
std::sort(arr.begin(), arr.end(), std::greater<int>());
```

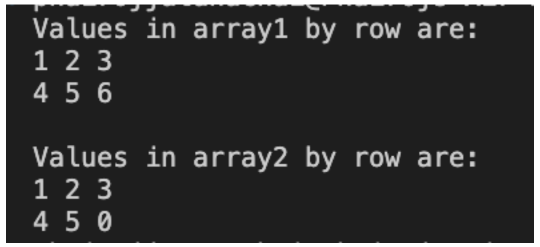
This is a function object (also known as a functor) defined in the <functional> header of the C++ Standard Library.
std::greater<T> is a comparison functor that returns true if the first argument is greater than the second.
In this case, std::greater<int> is used to compare integers. It is called repeatedly to compare pairs of elements in the container being sorted
By passing std::greater<int> as the third argument to std::sort, you instruct the function to sort the array in descending order (from largest to smallest).

## Multidimensional arrays



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

const size_t rows{2};
const size_t columns{3};
void printArray(const array<array<int, columns>, rows>&); // function
prototype

int main() {                    3         2
   array<array<int, columns>, rows> array1{{1, 2, 3, 4, 5, 6}};
   array<array<int, columns>, rows> array2{{1, 2, 3, 4, 5}};

   cout << "Values in array1 by row are:" << endl;
   printArray(array1);

   cout << "\nValues in array2 by row are:" << endl;
   printArray(array2);
}

// Output array with two rows and three columns
void printArray(const array<array<int, columns>, rows>& a) {
   // Loop through array's rows
   for (auto const& row : a) {
      // Loop through columns of current row
      for (auto const& element : row) {
         cout << element << ' ';
      }
      cout << endl; // Start new line of output
   }
}
```

```
for (size_t row{0}; row < a.size(); ++row) {
   for (size_t column{0}; column < a[row].size(); ++column) {
      cout << a[row][column] << ' '; }
   cout << endl;
}
```



```
Values in array1 by row are:
1 2 3
4 5 6

Values in array2 by row are:
1 2 3
4 5 0
```

## Vector

## Class Template vector

- What are vectors?
  - Dynamic arrays that can grow and shrink as needed.
  - Part of the C++ Standard Template Library (STL).
  - Store elements of the same data type.
- Why use vectors?
  - Flexibility in managing collections of data.
  - Efficient for insertion and deletion of elements.
  - Access elements directly using indices.

## Declaring and Initializing Vectors

- Syntax:
  vector<type> name; // Declares an empty
  vector vector<type> name(size); // Declares a vector with specified initial size
  vector<type> name(otherVector); // Copy constructor
  vector<type> name(start, end); // Initializer list from iterators
- Examples:
  vector<int> numbers; vector<string> names(5);

## Accessing Vector Elements

- Using square brackets:
  element = name[index];
  name[index] = value;
- Using the at() method:
  element = name.at(index); // Throws out_of_range exception for invalid indices

## Common Vector Operations

- size(): Returns the number of elements.
- push_back(element): Adds an element to the end.
- pop_back(): Removes the last element.
- clear(): Removes all elements.
- empty(): Checks if the vector is empty.
- insert(iterator, element): Inserts an element at a specific position.
- erase(iterator): Removes an element at a specific position.

## Vector Iterators

- Begin() and end(): Iterators pointing to the first and one-past-the-last elements.
- Using iterators for traversal and manipulation:

```
for (auto it = name.begin(); it != name.end(); ++it) {
    // Process each element using *it
}
```

## Introduction to Lists in C++

- A list is a collection of elements stored in a sequence.
- C++ provides std::list from the Standard Template Library (STL).
- Lists can be singly linked or doubly linked.
- Key Features of Lists:
  - Dynamic size
  - Efficient insertions and deletions
  - No direct access to elements (unlike arrays or vectors)

## Introduction to Lists in C++

- A list is a collection of elements stored in a sequence.
- C++ provides std::list from the Standard Template Library (STL).
- Lists can be singly linked or doubly linked.
- Key Features of Lists:
  - Dynamic size
  - Efficient insertions and deletions
  - No direct access to elements (unlike arrays or vectors)

## Introduction to Lists in C++

- A list is a collection of elements stored in a sequence.
- C++ provides std::list from the Standard Template Library (STL).
- Lists can be singly linked or doubly linked.
- Key Features of Lists:
  - Dynamic size
  - Efficient insertions and deletions
  - No direct access to elements (unlike arrays or vectors)

## Introduction to Lists in C++

- A list is a collection of elements stored in a sequence.
- C++ provides std::list from the Standard Template Library (STL).
- Lists can be singly linked or doubly linked.
- Key Features of Lists:
  - Dynamic size
  - Efficient insertions and deletions
  - No direct access to elements (unlike arrays or vectors)

## Introduction to Lists in C++

- A list is a collection of elements stored in a sequence.
- C++ provides std::list from the Standard Template Library (STL).
- Lists can be singly linked or doubly linked.
- Key Features of Lists:
    - Dynamic size
    - Efficient insertions and deletions
    - No direct access to elements (unlike arrays or vectors)

## Introduction to Lists in C++

- A list is a collection of elements stored in a sequence.
- C++ provides std::list from the Standard Template Library (STL).
- Lists can be singly linked or doubly linked.
- Key Features of Lists:
  - Dynamic size
  - Efficient insertions and deletions
  - No direct access to elements (unlike arrays or vectors)

## Introduction to Lists in C++

- A list is a collection of elements stored in a sequence.
- C++ provides std::list from the Standard Template Library (STL).
- Lists can be singly linked or doubly linked.
- Key Features of Lists:
  - Dynamic size
  - Efficient insertions and deletions
  - No direct access to elements (unlike arrays or vectors)

## Introduction to Doubly Linked Lists

- A doubly linked list (DLL) is a data structure consisting of nodes.
- Each node contains:
  - Data (value stored in the node)
  - Pointer to the next node (next)
  - Pointer to the previous node (prev)
- Unlike singly linked lists, DLLs allow bidirectional traversal.

# Introduction to Doubly Linked Lists

- Key Points:
  - The first node's prev is nullptr.
  - The last node's next is nullptr.
  - Middle nodes connect in both directions.

```
NULL <- [10] <-> [20] <-> [30] -> NULL
       (Head)            (Tail)
```

## Advantages of Doubly Linked Lists

- Efficient Insertions and Deletions: Can add/remove nodes at the front, middle, or end efficiently.
- Bidirectional Traversal: Can move forward and backward through the list.
- More Memory Overhead: Requires extra space for prev pointers.

## Introduction to Doubly Linked Lists Using std::list

- C++ provides std::list, which is implemented as a doubly linked list.
- Features of std::list
  - Bidirectional Iteration (Forward & Backward)
  - Fast Insertions/Deletions at any position
  - More Memory Overhead due to extra pointers

```
#include <iostream>
#include <list>

int main() {
    std::list<int> dll = {10, 20, 30};
    dll.push_back(40);  // Insert at the end
    dll.push_front(5);  // Insert at the front

    // Forward Traversal
    for (int num : dll) {
        std::cout << num << " ";
    }
    return 0;
}
```

## Operations on std::list

Insertion

- push_front(value) → Inserts an element at the front.
- push_back(value) → Inserts an element at the end.
- insert(iterator, value) → Inserts an element at a specific position.

Deletion

- pop_front() → Removes the first element.
- pop_back() → Removes the last element.
- erase(iterator) → Removes an element at a specific position.

Traversal

- Forward Traversal: Using iterators or range-based for loop.
- Reverse Traversal: Using rbegin() and rend().

Sorting & Reversing:

- sort() → Sorts the list in ascending order.
- reverse() → Reverses the order of elements.

| Feature/Aspect | std::list | std::array | std::vector |
|---|---|---|---|
| **Type** | Doubly Linked List | Fixed-size array | Dynamic array |
| **Header** | <list> | <array> | <vector> |
| **Size Flexibility** | Dynamic (can grow and shrink) | Fixed (size must be known at compile-time) | Dynamic (can grow and resize) |
| **Element Access** | Slow (sequential access) | Fast (direct access) | Fast (direct access) |
| **Insertions/Deletions** | Fast (anywhere in the list) | Not applicable (fixed size) | Fast at the end, slow in the middle or beginning |
| **Memory Allocation** | Non-contiguous (each element separate) | Contiguous (fixed block of memory) | Contiguous (may resize and move elements) |
| **Use Case** | When frequent insertions/deletions | When size is fixed and known | When size may vary or need to access by index |
| **Iterators** | Bidirectional | Random access | Random access |
| **Sorting** | Can be sorted with list::sort() | Requires std::sort | Can be sorted with std::sort |

## Multi-dimension: List, Area, Vector

```
#include <array>

std::array<std::array<int, 3>, 3> matrix = {{
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
}};
```

```
#include <vector>

std::vector<std::vector<int>> matrix = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
};
```

```
#include <list>

std::list<std::list<int>> matrix = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
};
```

## Pointers

## Agenda

- Introduction
- Pointer Variable Declarations and Initialization
- Pointer Operators
- Pass-by-Reference with Pointers
- Built-In Arrays
- Using const with Pointers
- sizeof Operator
- Pointer Expressions and Pointer Arithmetic
- Relationship Between Pointers and Built-In Arrays
- Pointer-Based Strings

## Introduction to Pointers

- A pointer in C++ is a variable that holds the memory address of another variable. This allows direct access and manipulation of the memory location.
- Type-specific: A pointer is declared with a specific data type and can only point to variables of that type.
- Memory Address: It stores the address of the variable it points to.
- Indirection: Through dereferencing (using the * operator), you can access or modify the value at the memory address.

## Pointers vs. References

A pointer is a variable that holds the memory address of another variable. It allows for direct memory access and manipulation.

```
int x = 10;    // an integer variable
int *p = &x;   // a pointer variable that holds the address
of x

// Accessing the value at the pointed location
cout << "Value of x: " << *p << endl;  // Output: Value of
x: 10

 *p = 20;  // Modifying the value at the memory address
of x

 cout << "After modification: x = " << x << endl;  //
Output: x = 20
```

A reference is an alias for another variable. It must be initialized to a variable when declared and cannot be changed to reference another variable later.

```
int x = 10;   // an integer variable
int &ref = x; // ref is a reference to x

// Modifying the value of x through the reference
ref = 20;
cout << "Value of x: " << x << endl;  // Output: Value of
x: 20
```

## Comparing Pointers and References

**1. Initialization:**

- Pointers can be initialized to nullptr or to the address of a variable.
- References must be initialized to an existing variable when declared.

**2. Reassignment:**

- Pointers can be reassigned to point to different variables.
- References cannot be reassigned; they always refer to the initial object.

**3. Syntax:**

- Pointers use * for dereferencing and & for getting the address.
- References use plain syntax as if dealing with the original variable.

**4. Use Cases:**

- Pointers are used for dynamic memory allocation, implementing data structures, and more complex tasks.
- References are used when an alias of a variable is needed, often in function arguments to avoid copying.

## Pointers vs. References

Pointer Example:

```
void increment(int *ptr) {
   if(ptr) *ptr += 1; // Increment only if ptr is not null
}

int main() {
   int x = 5;
   increment(&x);
   cout << x << endl;  // Output: 6
}
```

Reference Example:

```
void increment(int &ref) {
   ref += 1;
}

int main() {
   int x = 5;
   increment(x);
   cout << x << endl;  // Output: 6
}
```

# Key Differences Between Pointers and References

| Feature | Pointer (*p) | Reference (&ref) |
|---|---|---|
| **Stores Address?** | Yes (`int *p = &x;`) | No, it is an alias |
| **Can be Null?** | Yes (`p = nullptr;`) | No (`ref` must be initialized) |
| **Can be Reassigned?** | Yes (`p = &y;`) | No (`ref` always refers to `x`) |
| **Needs Dereferencing?** | Yes (`*p = 20;`) | No (`ref = 20;`) |
| **Memory Access** | Requires extra lookup (`*p`) | Direct access (faster) |
| **Usage** | Dynamic memory, function pointers, optional indirection | Always bound to one variable |

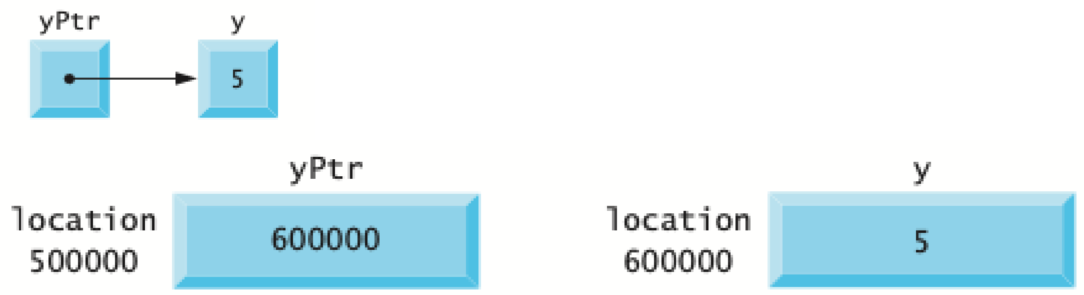
## Which One to Use

Use References (&ref) if:
- You need a permanent alias to a variable.
- You don't want the ability to reassign the reference.
- You want slightly better performance (no extra memory lookup).

Use Pointers (*p) if:
- You need to dynamically allocate memory (new/delete).
- You need to change what the pointer points to.
- You want optional (nullptr) behavior.

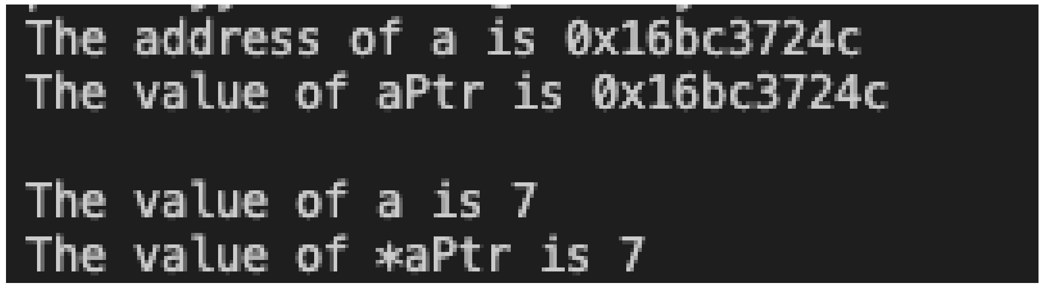
## Pointer Variable Declarations and Initialization

- Declaring pointers
  int* countPtr, count=7;
  Initialization pointers
  int* ptr = nullptr;
- Address (&) Operators
  int y{5};
  Int* yPtr{nullptr};
  yptr = &y;
- Indirection (*) Operators
  cout << *yPtr << endl;
  cout << y << endl;
  *yPtr = 9;
  cin >> *yPtr;



## Address (&) and Indirection (*) Operators

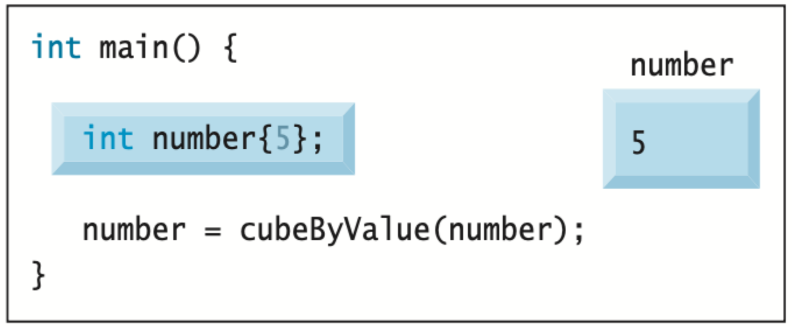
```
#include <iostream>
using namespace std;
int main() {
   int a{7}; // initialize a with 7
   int* aPtr = &a; // initialize aPtr with the address of int variable a

   cout << "The address of a is " << &a
   << "\nThe value of aPtr is " << aPtr;
   cout << "\n\nThe value of a is " << a
   << "\nThe value of *aPtr is " << *aPtr << endl;
}
```



```
The address of a is 0x16bc3724c
The value of aPtr is 0x16bc3724c

The value of a is 7
The value of *aPtr is 7
```

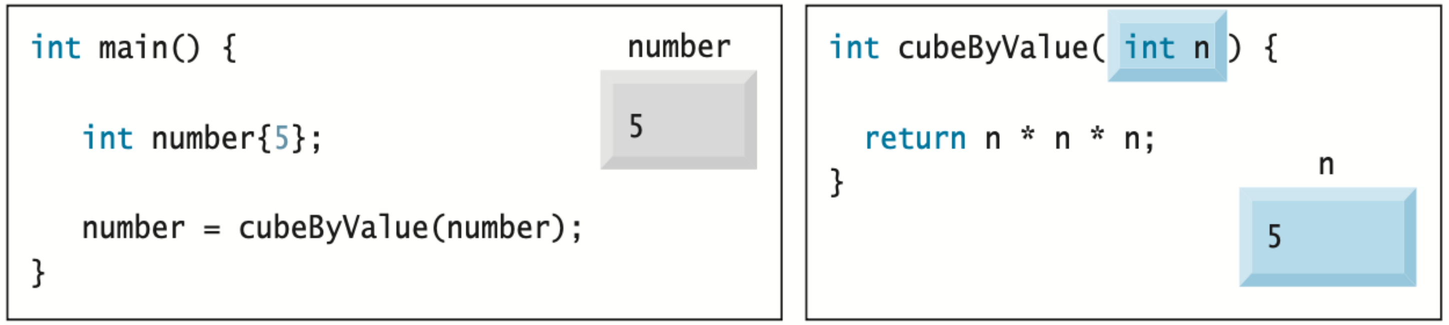
## Pass-by-Value

Step 1: Before main calls cubeByValue:



```
int main() {

   int number{5};

   number = cubeByValue(number);
}
```

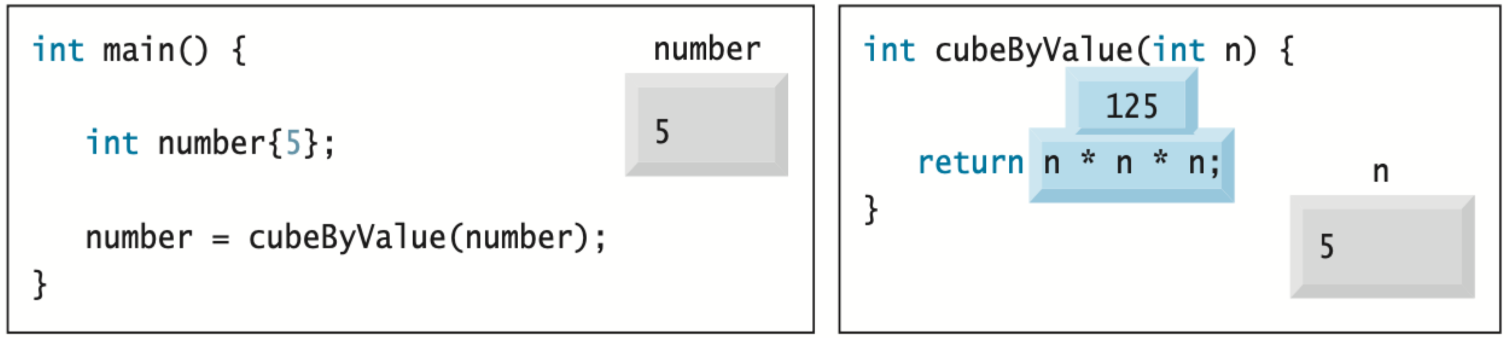
Step 2: After cubeByValue receives the call:



```
int main() {

   int number{5};

   number = cubeByValue(number);
}
```

```
int cubeByValue( int n ) {

   return n * n * n;
}
```

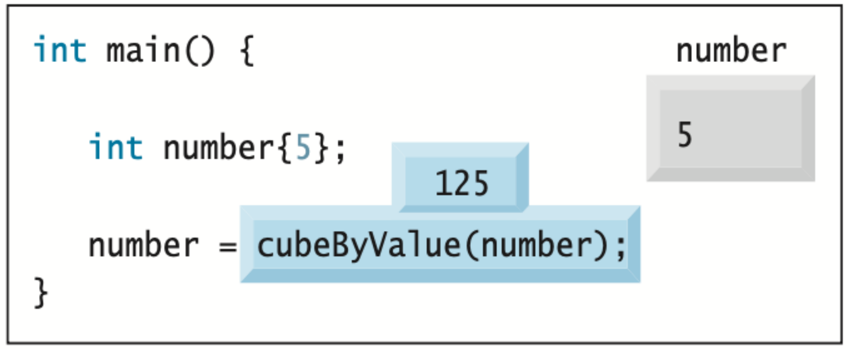
Step 3: After cubeByValue cubes parameter n and before cubeByValue returns to main:



```
int main() {

   int number{5};

   number = cubeByValue(number);
}
```

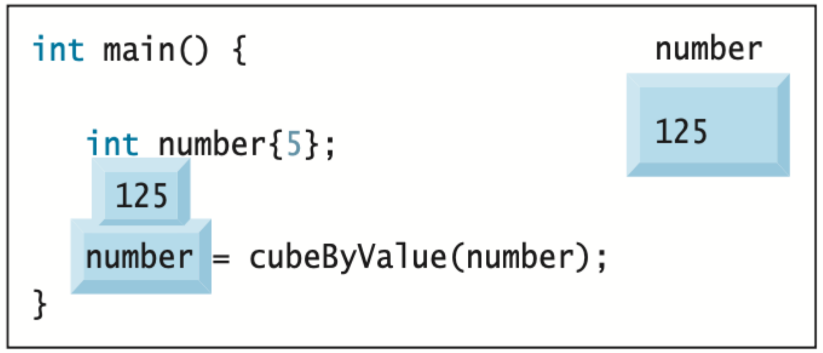
```
int cubeByValue(int n) {

   return n * n * n;
}
```

Step 4: After cubeByValue returns to main and before assigning the result to number:



```
int main() {

   int number{5};

   number = cubeByValue(number);
}
```

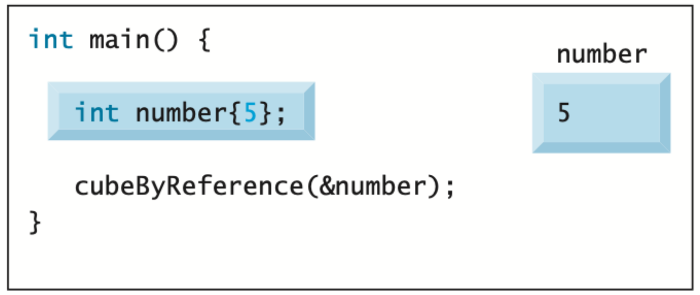
Step 5: After main completes the assignment to number:



```
int main() {

   int number{5};

   number = cubeByValue(number);
}
```

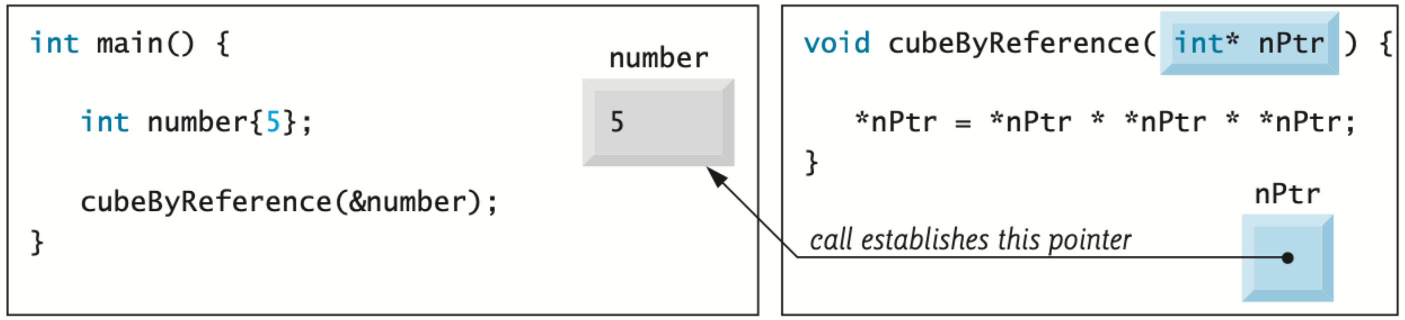
# Pass-by-Reference (pointer)

Step 1: Before `main` calls `cubeByReference`:



```
int main() {
   int number{5};

   cubeByReference(&number);
}
```

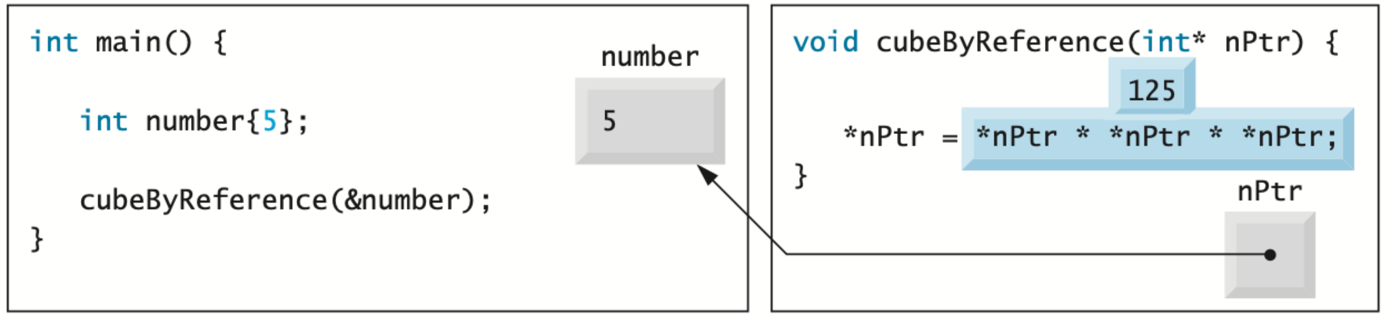
Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:



```
int main() {
   int number{5};

   cubeByReference(&number);
}
```

```
void cubeByReference( int* nPtr ) {
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

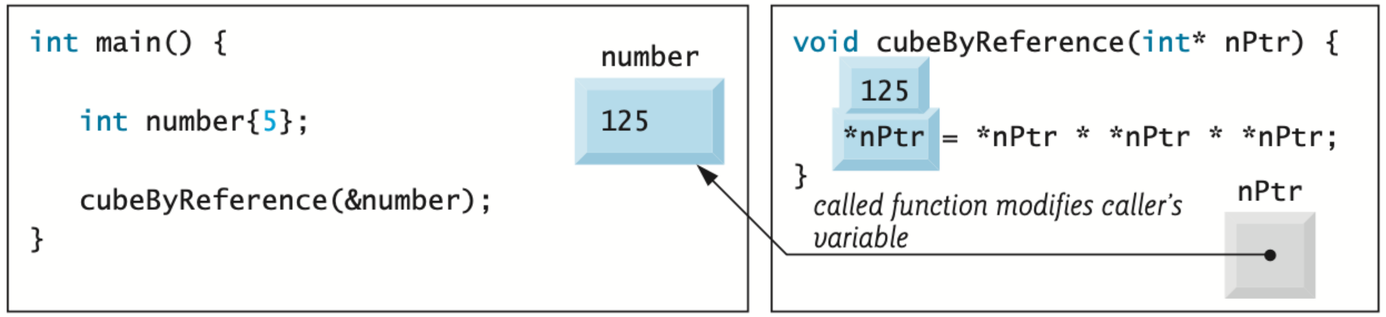
Step 3: Before`*nPtr` is assigned the result of the calculation `5 * 5 * 5`:



```
int main() {
   int number{5};

   cubeByReference(&number);
}
```

```
void cubeByReference(int* nPtr) {
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

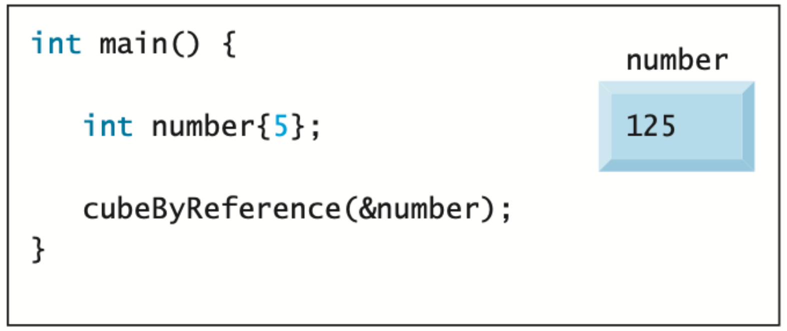
Step 4: After `*nPtr` is assigned 125 and before program control returns to `main`:



```
int main() {
   int number{5};

   cubeByReference(&number);
}
```

```
void cubeByReference(int* nPtr) {
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

Step 5: After `cubeByReference` returns to `main`:



```
int main() {
   int number{5};

   cubeByReference(&number);
}
```

## Built-In Arrays

- Declaration and initializing

```
type  arrayName[arraySize];
int c[12];
int n[5] {60, 20, 30, 40, 70};
int n[] {60, 20, 30, 40, 70};
```

- Passing built-in array to function

```
void func1(int arr[], int size) {}    //  or void func1(int* arr, int size)
    . . .
int myArr[] {1,2,3};
int size = sizeof(myArr) / sizeof(myArr[0]);
func1(myArr, size);
```

-

## Standard Library Functions begin and end

```
#include <iostream>
#include <algorithm> // Required for std::sort
#include <iterator>  // Required for std::begin, std::end

int main() {
    int n[] = {4, 2, 5, 3, 1};

    // Sort the array using std::sort
    std::sort(std::begin(n), std::end(n));   // ??? can I use ‘sort(n.begin(), n.end());’ ???

    // Print the sorted array
    for (int i : n) {                        // ??? Is it the same ‘for (int i = 0; i < sizeof(n); i++) {‘  ???
        std::cout << i << ' ';
    }

    return 0;
}
```

## Built-In Array Limitations

• They cannot be compared using the relational and equality operators—you must use a loop to compare two built-in arrays element by element.
• They cannot be assigned to one another—an array name is effectively a pointer that is const.
• They don't know their own size—a function that processes a built-in array typically receives both the built-in array's name and its size as arguments.
• They don't provide automatic bounds checking—you must ensure that array-access expressions use subscripts that are within the built-in array's bounds.

## Using const with Pointers

- const enables you to inform the compiler that the value of a particular variable should not be modified.
- The built-in array's size is used in the function's body to determine the highest subscript so the loop can terminate when the processing completes. The size does not need to change in the function body, so it should be declared const to ensure that it will not change.
- There are four ways to pass a pointer to a function:
    - a nonconstant pointer to nonconstant data.
    - a nonconstant pointer to constant data.
    - a constant pointer to nonconstant data.
    - a constant pointer to constant data.

# Nonconstant Pointer to Nonconstant Data

Allows the function to modify both the pointer and the data the pointer is pointing to.

```
void modifyBoth(int* ptr) {
    *ptr = 5; // Modify the data pointed to by ptr
    ptr = nullptr; // Modify the pointer itself
}
```

## Nonconstant Pointer to Constant Data

Allows the function to modify the pointer but not the data it points to.

```
void modifyPointer(const int* ptr) {
   // *ptr = 5; // Error: cannot modify the data
   ptr = nullptr; // OK: can modify the pointer itself
}
```

## Constant Pointer to Nonconstant Data

Allows the function to modify the data pointed to by the pointer, but not the pointer itself.

```
void modifyData(int* const ptr) {
    *ptr = 5; // OK: can modify the data
    // ptr = nullptr; // Error: cannot modify the constant pointer itself
}
```

## Constant Pointer to Constant Data

Not allow the function to modify either the pointer or the data it points to.

```
void modifyNone(const int* const ptr) {
   // *ptr = 5; // Error: cannot modify the data
   // ptr = nullptr; // Error: cannot modify the constant pointer itself
}
```

## sizeof Operator

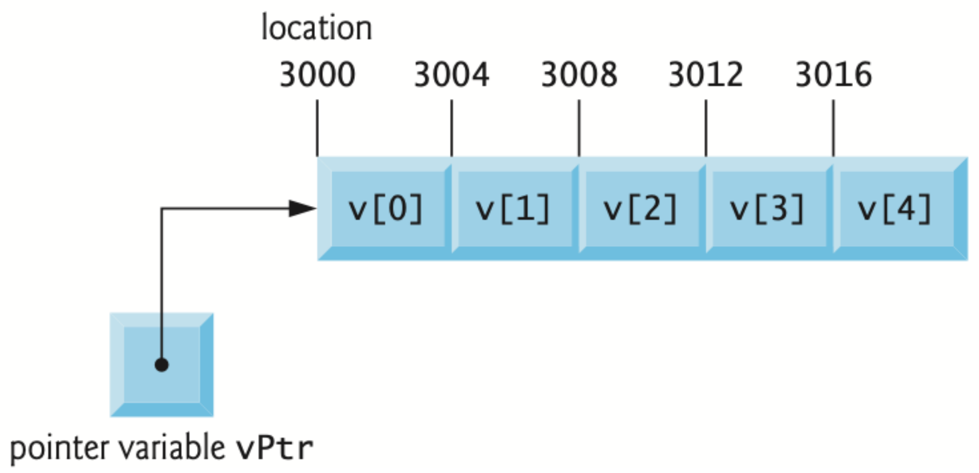
- The compile time unary operator sizeof determines the size in bytes of a built-in array or of any other data type, variable or constant.
- When applied to a built-in array's name, sizeof returns the total number of bytes in the built-in array as a value of type size_t.

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

size_t getSize(double*);

int main() {
  double numbers[20];

   std::cout << ”array size: " << sizeof(numbers);
   std::cout << "\nnumber returned by getSize is " << getSize(numbers) << std::endl;
}

size_t getSize(double* ptr) {
   return sizeof(ptr);
}
```

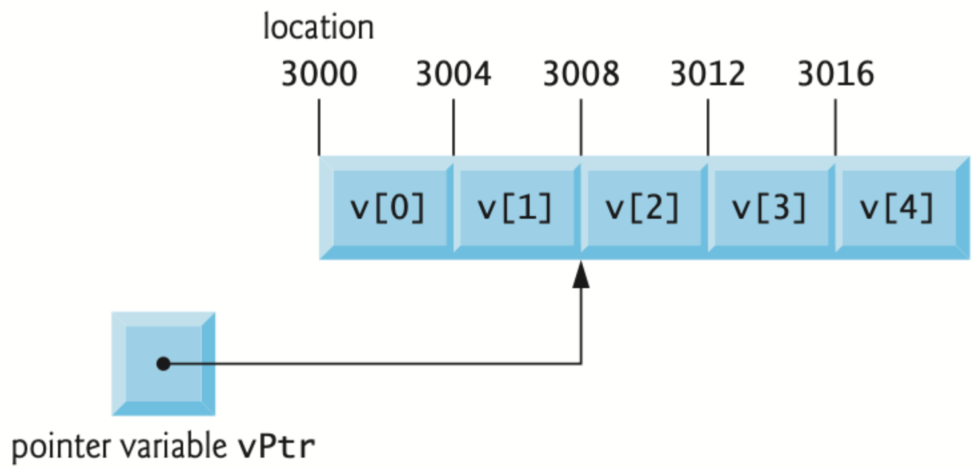
??? Size_t
??? Output line 1
??? Outpit line 2

# Pointer Expressions and Pointer Arithmetic

```
int  v[5];
int* vPtr{v};
int* vPtr{&v[0]};
```



```
vPtr += 2;
```



*There's no bounds checking on pointer arithmetic. You must ensure that every pointer arithmetic operation that adds an integer to or subtracts an integer from a pointer results in a pointer that references an element within the built-in array's bounds.*

## Subtracting Pointers

```
#include <iostream>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int* ptr1 = &arr[1]; // Pointer to second element
    int* ptr2 = &arr[4]; // Pointer to fifth element

    std::ptrdiff_t diff = ptr2 - ptr1;
    std::cout << "The pointers are " << diff << " elements apart." << std::endl;

    return 0;
}
```

??? result

## Pointer Assignment

```
#include <iostream>

int main() {
    int value = 5;
    int* ptr1 = &value;
    int* ptr2 = ptr1; // ptr2 is now pointing to the same location as ptr1

    std::cout << "Value of *ptr2: " << *ptr2 << std::endl;

    return 0;
}
```

## Cannot Dereference a void*

```
#include <iostream>

int main() {
    int value = 5;
    void* ptr = &value;

    // std::cout << *ptr << std::endl; // Error: cannot dereference a void pointer

    // Correct way: Cast void* to int* before dereferencing
    std::cout << "Value of *ptr: " << *(static_cast<int*>(ptr)) << std::endl;

    return 0;
}
```

## Comparing Pointers

```
#include <iostream>

int main() {
    int arr[] = {10, 20, 30};
    int* ptr1 = &arr[0];
    int* ptr2 = &arr[0];
    int* ptr3 = &arr[1];

    if (ptr1 == ptr2) {
        std::cout << "ptr1 and ptr2 point to the same memory location." << std::endl;
    }

    if (ptr1 != ptr3) {
        std::cout << "ptr1 and ptr3 do not point to the same memory location." << std::endl;
    }

    return 0;
}
```

## Pointer-Based Strings

- a "Pointer-Based String" is an array of characters terminated by a null character ('\0’).
- A pointer-based string uses a pointer to refer to the first character of the string.

```
const char *str = "Hello, World!";
```

- Declares str as a pointer to a const char.
- The string literal "Hello, World!" is typically placed in read-only memory.
- str points to the first character of this string literal.
- The characters of the string cannot be modified through str (attempting to do so would be undefined behavior).
- The pointer str can be reassigned to point to another string literal or array of characters.

```
char str[] = "Hello, World!";
```

- Declares str as an array of char, with size automatically determined by the number of characters in the string literal, including the null terminator.
- The string literal is copied into the array str, which is stored in the stack (for local variables) or in static/global memory (for global variables).
- The contents of the array can be modified since they are not const.
- The array str cannot be reassigned to point to another string or character array.

## Pointer-Based Strings

- Character constants as initializers

```
char str[] = "Hello, World!";
char str[] = {'H','e','l','l','o',',',' ','W','o','r','l','d','!','\0’};
```

- Accessing characters in a C String

```
const char *str {"Hello, World!"};
std::cout << str[1];
```

- Reading strings into built-in array of char with cin

```
const char* word;
std::cin >> word;
```

```
char word[20];
std::cin >> word;
```

```
char word[20];
std::cin >> std::setw(20) >> word;
```

```
char word[20];
std::cin.getline(word, sizeof(word));
```

## Queue

- queue container adapter that models the FIFO (first-in, first-out) strategy. It is included in the <queue> header.
- push: Add an element to the end of the queue.
- pop: Remove the element at the front of the queue.
- front: Access the element at the front of the queue.
- back: Access the element at the back of the queue.
- empty: Check if the queue is empty.
- size: Get the number of elements in the queue.

```
#include <iostream>
#include <queue>

int main() {
   std::queue<int> myQueue;

   // Adding elements to the queue
   myQueue.push(10);
   myQueue.push(20);
   myQueue.push(30);

   // Now the queue is {10, 20, 30}, with 10 at the front

   // Accessing the front element
   std::cout << "Front element: " << myQueue.front() << std::endl;  //
Outputs
```

## Queue Benefit

• **Simplicity and Clarity:** A queue provides a clear and simple FIFO (First-In, First-Out) data structure that models real-world scenarios like lines and task scheduling, making the code easier to understand and maintain.
• **Encapsulation:** By encapsulating the FIFO behavior, a queue reduces the complexity of the code, as the user does not need to manage the underlying data structure for adding and removing elements.
• **Safety:** The queue interface restricts operations to only a few safe actions (enqueue, dequeue, front access, etc.), which prevents errors that could arise from manipulating the data structure in unintended ways.
• **Consistent Performance:** std::queue guarantees constant time complexity for insertions and removals, which is critical for real-time and high-performance applications where predictable execution time is necessary.
• **Concurrency Support:** Queues are commonly used in multithreaded programming for task scheduling and synchronization. The clear semantics of a queue make it easier to design clean inter-thread communication with less chance for deadlock or race conditions.

## Q & A

## Mini Projects

- Score 20%
- Team of 2 students
- Submit Proposal: Jan 19th, 2025 18:00
- Progress Report: Feb 20th, 2025  18:00
- Final submission: Mar 28th, 2025 18:00
- Present Project: 3rd April , 2025 start at 9:00 am
- Presentation time: 15 minutes

## Mini Projects Topics

**Project Topics: Choose one from the following topics:**
- Financial Calculator: Design a program that performs various financial calculations.
- Weather Data Analysis: Analyze and display weather data in a user-friendly format.
- Educational Tool for Kids: Develop an interactive learning tool for children.
- E-commerce Store Simulation: Create a simulation of an online store's backend.
- Social Media Dashboard: Develop a dashboard to interact with social media APIs.
- Fitness Tracker: Design a program to track and analyze fitness activities.
- Library Management System: Build a system to manage book inventories and user data.
- Music Player: Create a simple music player application.
- Artificial Intelligence (AI) Chatbot: Develop a basic AI chatbot.
- Healthcare Management System: Design a system to manage patient records.
- Custom Project: Propose your unique project idea (subject to approval).

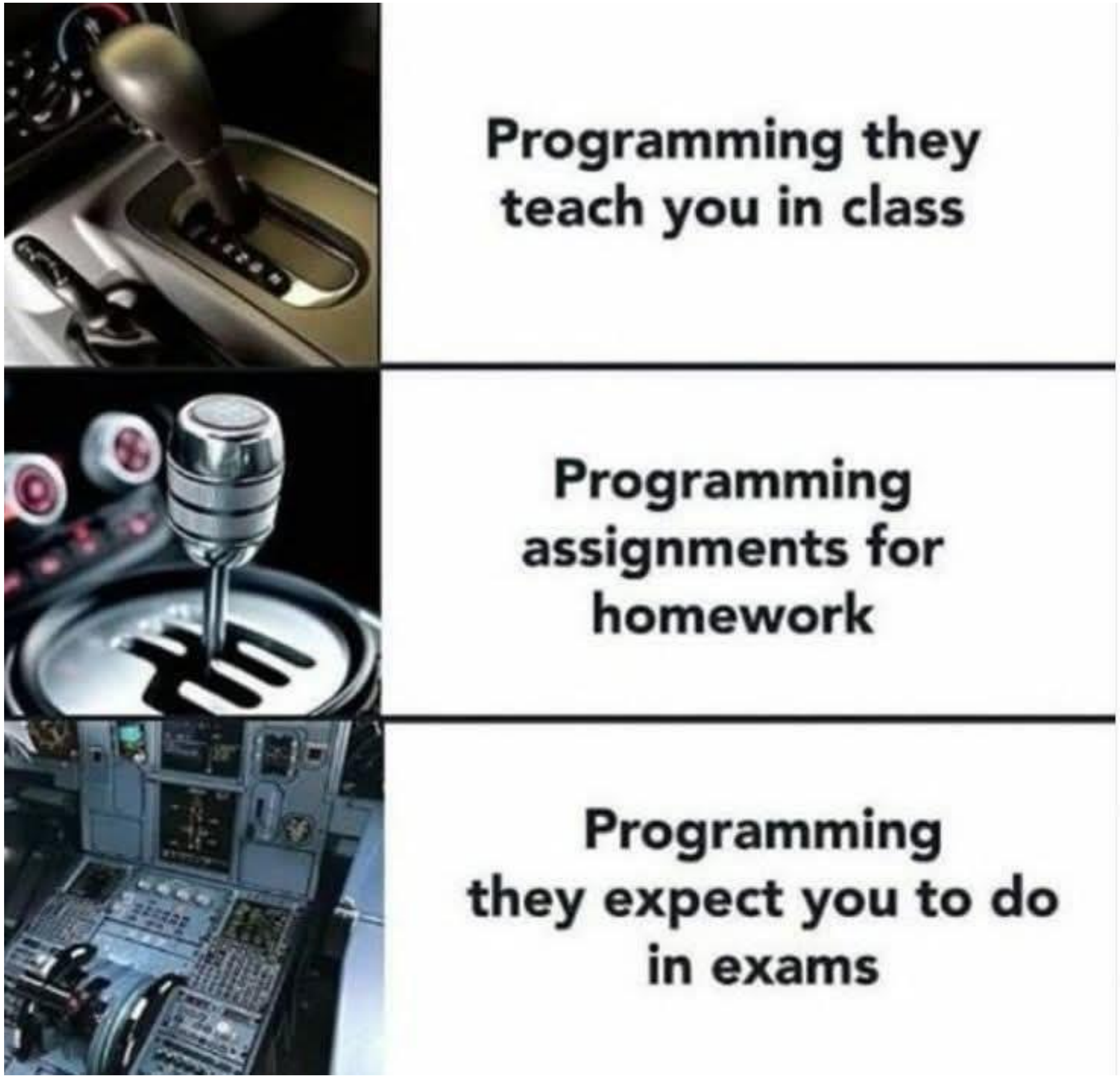
*** No graphics engine allowed ***

## Deliverables

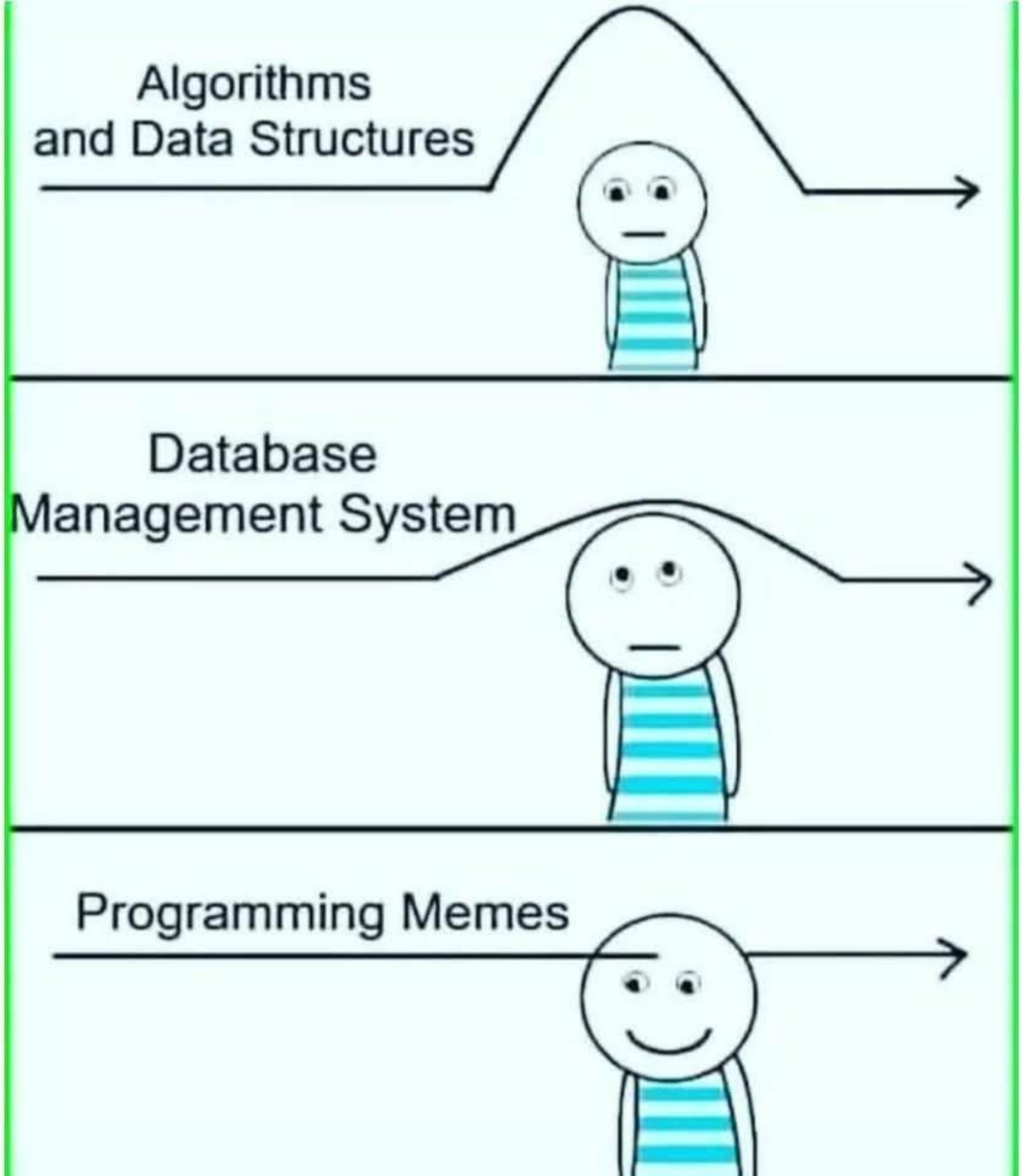
- Project Proposal: Submit a brief proposal outlining your chosen topic, objectives, and initial plan.
- Progress Report: Submit a progress report detailing what has been accomplished and what remains to be done.
- Final Submission: Include the following in your final submission:
    - Source code with proper documentation.
    - A report detailing the project's design, implementation, challenges faced, and how they were overcome.
    - A short video presentation demonstrating the functionality of your project.

## Evaluation Criteria

- Adherence to OOP principles. (at least 50% of code in C++)
- Code quality and documentation.
- Creativity and practical application of the project.
- Teamwork and collaboration.
- Presentation and demonstration of the project.


Programming they teach you in class
Programming assignments for homework
Programming they expect you to do in exams

## Q & A


Algorithms and Data Structures
Database Management System
Programming Memes