Pointer uses -
1. Accessing Heap Memory
2. Accessing Resources
3. Parameter Passing

CPP Learning (chatgpt):
In C++, arrays do not have a built-in method or property like `arr.length` in JavaScript to determine their length. However, you can use the `sizeof` operator to find the size of an array in bytes and then divide it by the size of one element to get the number of elements. Here's an example:

```cpp
#include <iostream>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    std::cout << "Length of array: " << size << std::endl;
    return 0;
}
```

This code will output:

```
Length of array: 5
```

Here, `sizeof(arr)` gives the total size of the array in bytes, and `sizeof(arr[0])` gives the size of one element in the array. Dividing these two gives you the number of elements in the array.

Certainly! In C++, if you're using dynamic memory allocation with `malloc`, you won't have a direct way to determine the size of the allocated memory block, as `malloc` doesn't store that information.

However, you can manually keep track of the size yourself. Here's how you can modify your code to do that:

```cpp
#include <iostream>
#include <cstdlib> // for malloc and free

int main() {
    int *p;
    int size = 5; // Specify the size of the array
```

```
    p = (int *)malloc(size * sizeof(int));

    // Check if memory allocation was successful
    if (p == nullptr) {
        std::cout << "Memory allocation failed." << std::endl;
        return 1;
    }

    // Now you can use 'size' to keep track of the size of the array
    for (int i = 0; i < size; ++i) {
        p[i] = i + 1;
    }

    std::cout << "Elements in the array:" << std::endl;
    for (int i = 0; i < size; ++i) {
        std::cout << p[i] << " ";
    }
    std::cout << std::endl;

    // Don't forget to free the allocated memory
    free(p);
    return 0;
}
```

In this code, `size` is manually set to 5, which represents the number of elements in the dynamically allocated array. You can change `size` to any value you want. The rest of the code remains the same, but now it's explicitly keeping track of the array size using the `size` variable.

That's correct. When you allocate memory dynamically using `malloc` in C++, there's no direct built-in way to determine the size of the allocated memory block. The `sizeof` operator doesn't work for dynamically allocated memory because `malloc` doesn't store that information.

You either need to keep track of the size yourself using a separate variable, as shown in the example code, or consider using C++ standard library containers like `std::vector`, which automatically manage the size for you and provide methods to obtain the size. Using `std::vector` is generally preferred in modern C++ code because it's safer and more convenient than managing dynamic arrays manually.

Sure! Here's an example using `std::vector`:

```cpp
```

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec; // Create an empty vector

    // Add elements to the vector
    for (int i = 1; i <= 5; ++i) {
        vec.push_back(i);
    }

    // Output the elements in the vector
    std::cout << "Elements in the vector:" << std::endl;
    for (int i = 0; i < vec.size(); ++i) {
        std::cout << vec[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

In this code:

- `std::vector<int> vec;` creates an empty vector of integers.
- `vec.push_back(i);` adds elements to the vector.
- `vec.size()` returns the number of elements in the vector.

With `std::vector`, you don't need to worry about managing memory or keeping track of the size manually. The vector dynamically resizes itself as elements are added or removed, and you can always get the current size using the `size()` method.

Yes, `std::vector` in C++ dynamically allocates memory to store its elements. When you add elements to a `std::vector`, it automatically manages memory allocation for you. If the vector needs more space to accommodate additional elements, it will dynamically allocate a larger memory block, copy the existing elements to the new memory block, and deallocate the old memory block.

This dynamic memory management is one of the key advantages of using `std::vector`. It allows you to work with collections of elements without worrying about managing memory allocation and deallocation manually, as you would have to do with raw arrays or `malloc` in C or C++.

Yes, `std::vector` stores its elements on the heap. When you create a `std::vector`, it internally manages a dynamically allocated array on the heap

to hold its elements. This dynamic allocation allows the vector to resize itself as needed and efficiently manage memory.

The heap is a region of memory where dynamic memory allocation occurs. Unlike the stack, which has a limited size and is used for local variables and function call frames, the heap can grow and shrink dynamically during program execution, making it suitable for managing dynamically allocated memory, such as that used by `std::vector`.

Here's a chart showing the memory sizes of different data types in C++ on a typical system. Keep in mind that these sizes can vary depending on the compiler, architecture, and operating system:

Data Type	Size (in bytes)
char	1
unsigned char	1
signed char	1
short	2
unsigned short	2
int	4
unsigned int	4
long	4 or 8
unsigned long	4 or 8
long long	8
unsigned long long	8
float	4
double	8
long double	8 or more
bool	Typically 1, compiler-dependent

Please note that the sizes of `long`, `unsigned long`, and `long double` can vary depending on the system. On many modern systems, `long` and `unsigned long` are 4 bytes on 32-bit systems and 8 bytes on 64-bit systems, while `long double` can be 8, 12, 16, or even more bytes depending on the compiler and architecture.

Yes, C++ can be considered verbose due to its emphasis on explicitness and adherence to strict syntactic rules. Compared to higher-level languages such as Python or JavaScript, C++ requires developers to manage memory explicitly, declare variables before usage, and follow specific conventions for organizing code. Additionally, certain aspects of the C++ Standard Library, like date and time manipulation, require writing more code to achieve similar functionality found in other languages. While experienced developers become accustomed to this style, beginners often find C++ more complex and verbose initially.

Pointer topic codes:

pointers1.cpp

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>

using std::cout;
int main()
{

    int a = 10;
    int *p;                             // Declaration of pointer
    p = &a;                             // Assignment or Initialization (&a =
address of a)
    printf("using pointer: %i\n", p);  // value of p (prints address of a)
    printf("using pointer: %i\n", *p); // (dereferencing) value of p (prints
data of a)
    printf("using pointer: %i\n", &a); // Address of a
    // dereferencing - go to the adddress and get the value
    // &a is the address and p is pointing to it

    // Pointer to an array
    // Whenever you declare anything in your program it's going to be created
inside the stack
    int arr[5] = {2, 4, 6, 8, 10};
    int *q;
    q = arr;
    // if you want to use '&' ampercent
    // q = &arr[0];
    cout << arr << '\t' << &arr[0] << '\n'; // Note: the name of array itself
contains the addredss of its first index

    // traversing array
    for (int i = 0; i < 5; i++)
    {

        printf("%i, %i\n", arr[i], q[i]); // here pointer name is working like
array name both has same output
    }
    return 0;
}
```

pointers2.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
```

```
int main()
{
    int arr[5] = {10,
                  12,
                  18,
                  20,
                  98};
    // **create array in heap using pointer**
    int *p; // it's in the stack
    /* It will get the size of 5 integers sot it's an array of integers
malloc(size) - see notes */
    // p = (int *)malloc(5 * sizeof(int));     // it's in the heap
    p = new int[5]; // equivalent c++ code (works just like malloc)
    // initialize the values
    /*
        In C++, arrays do not have a built-in method or property like
arr.length in JavaScript to determine their length. However, you can use the
sizeof operator to find the size of an array in bytes and then divide it by the
size of one element to get the number of elements. Here's an exaple:
            int arr[] = {1, 2, 3, 4, 5};
            int size = sizeof(arr) / sizeof(arr[0]);
            std::cout << size;
    */
    int size = 5;
    // adding values to array inside heap
    for (int i = 0; i < size; i++)
    {
        std::cin >> p[i];
    }
    // using range based loop
    // for (int x : arr)
    // {
    //     std::cout << x << std::endl;
    // }
    for (int i = 0; i < size; i++)
    {
        std::cout << p[i] << std::endl;
    }

    // **delete the memory we used from heap**

    /* Whenever we are dynamically allocating memory we need to release the
memory   after using it. Can be ignored in small programs but necessary step
for big projects. */

    // use for 'malloc()' (c lang)
```

```
    // 'free' and 'delete' deallocates memory that is allocated in heap
    // free(p);
    // 'delete' used in c++ to free allocated memory in heap using 'new'
    delete[] p; // if any other type of variable write just the variable name
for array use square brackets
    return 0;

    // Once the program ends heap memory will be automatically(implicitly)
deleted
    // Implicit - what compiler does
    // Explicit - what programmer does
}
```

pointers3.cpp

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using std::cin;
using std::cout;
using std::endl;

// Defining structure

struct Rectangle
{
    int length;
    int breadth;
};

int main()
{
    int *p1;
    char *p2;
    float *p3;
    double *p4;
    // pointer of type structure
    struct Rectangle *p5;
    cout << sizeof(p1) << endl; // sizeof prints the size of data type in bytes
    cout << sizeof(p2) << endl;
    cout << sizeof(p3) << endl;
    cout << sizeof(p4) << endl;
    cout << sizeof(p5) << endl;
```

```
    // Whatever data type pointer is, pointer takes same amount of memory for some compilers it will be 4bytes for some 8bytes(Depends on compiler architecture 32/64). Size of pointer is independant of its data type.
    return 0;
}
```

Call by value and call by reference:

Certainly! Let's start with call by value and call by reference in both C and C++.

Call by Value:

C:

In call by value, a copy of the argument value is passed to the function. The function works with this copy, and any modifications made to the parameter inside the function do not affect the original value outside the function.

Example (C):

```c
#include <stdio.h>

void changeValue(int x) {
    x = 10; // Changing the value of the parameter
}

int main() {
    int num = 5;
    printf("Before function call: %d\n", num);
    changeValue(num); // Passing by value
    printf("After function call: %d\n", num); // num remains unchanged
    return 0;
}
```

Output:

```
Before function call: 5
After function call: 5
```

C++:

The concept of call by value in C++ is similar to C. A copy of the argument value is passed to the function, and modifications inside the function do not affect the original value outside the function.

Example (C++):
```cpp
#include <iostream>

void changeValue(int x) {
    x = 10; // Changing the value of the parameter
}

int main() {
    int num = 5;
    std::cout << "Before function call: " << num << std::endl;
    changeValue(num); // Passing by value
    std::cout << "After function call: " << num << std::endl; // num remains
unchanged
    return 0;
}
```

Output:
```
Before function call: 5
After function call: 5
```

Call by Reference:

C:
In C, there's no native support for call by reference like in C++. However, it can be emulated using pointers. By passing the address of a variable, the function can directly modify the value stored at that address.

Example (C):
```c
#include <stdio.h>

void changeValue(int *x) {
    *x = 10; // Changing the value at the address stored in x
}

int main() {
    int num = 5;
    printf("Before function call: %d\n", num);
    changeValue(&num); // Passing by reference (via pointer)
    printf("After function call: %d\n", num); // num is modified
    return 0;
```

```
}
```

Output:
```
Before function call: 5
After function call: 10
```

C++:
In C++, call by reference is a native feature. When a reference is passed to a function, any modifications made to the parameter inside the function directly affect the original value outside the function.

Example (C++):
```cpp
#include <iostream>

void changeValue(int &x) {
    x = 10; // Changing the value of the reference
}

int main() {
    int num = 5;
    std::cout << "Before function call: " << num << std::endl;
    changeValue(num); // Passing by reference
    std::cout << "After function call: " << num << std::endl; // num is
modified
    return 0;
}
```

Output:
```
Before function call: 5
After function call: 10
```

Explanation:
- **Call by Value**: In call by value, a copy of the actual parameter is passed to the function. The function works with this copy, and any modifications made to the parameter inside the function do not affect the original value outside the function.

- **Call by Reference**: In call by reference, the memory address of the actual parameter is passed to the function. This allows the function to directly

access and modify the original value of the parameter, affecting it outside the function.

In C, call by reference can be simulated using pointers, while in C++, it's directly supported through references. Using call by reference can be more efficient when working with large data structures, as it avoids unnecessary copying of values.

In C++, a reference variable doesn't allocate extra memory. It's essentially an alias or another name for an existing variable. When you create a reference variable, it's bound to the memory location of the variable it references. Therefore, modifying the reference variable affects the original variable directly because they both refer to the same memory location.

Let's illustrate this with an example:

```cpp
#include <iostream>

int main() {
    int num = 5;
    int &ref = num; // Reference variable ref refers to num

    std::cout << "num: " << num << std::endl;
    std::cout << "ref: " << ref << std::endl;

    // Modify the value using the reference
    ref = 10;

    std::cout << "After modification:" << std::endl;
    std::cout << "num: " << num << std::endl;
    std::cout << "ref: " << ref << std::endl;

    return 0;
}
```

In this example, `ref` is a reference variable that refers to the variable `num`. Modifying `ref` directly affects `num` because they both refer to the same memory location. There's no extra memory allocated for `ref`; it just acts as an alias for `num`.

Output:
```
num: 5
ref: 5
```

```
After modification:
num: 10
ref: 10
```

As you can see, changing the value of `ref` also changes the value of `num`. This behavior demonstrates that reference variables in C++ don't allocate extra memory but instead provide a way to work with the same memory location using a different name or reference.

In C++, you cannot directly print the memory address of a reference variable using the address-of operator (`&`) as you would with a regular variable. This is because a reference variable doesn't have its own memory address; it is simply an alias for another variable's memory address.

However, you can print the memory address of the variable that the reference is referring to. Since the reference variable points to the same memory location as the original variable, printing the address of the original variable will effectively show the memory address of the reference variable indirectly.

Here's an example:

```cpp
#include <iostream>

int main() {
    int num = 5;
    int &ref = num;

    std::cout << "Memory address of num: " << &num << std::endl;
    std::cout << "Memory address of ref: " << &ref << std::endl;

    return 0;
}
```

In this example, `&num` and `&ref` will print the same memory address because `ref` is simply another name for `num`.

Output:
```
Memory address of num: 0x7ffd48f3c62c
Memory address of ref: 0x7ffd48f3c62c
```

As you can see, the memory addresses of `num` and `ref` are the same. This demonstrates that a reference variable doesn't have its own memory address but rather refers to the memory address of another variable.

References are commonly used in parameter passing in C++ for various reasons, including efficiency, readability, and to modify variables within a function. Here's an example to illustrate the usefulness of references in parameter passing:

```cpp
#include <iostream>

// Function to swap two integers using call by reference
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int num1 = 5, num2 = 10;

    std::cout << "Before swapping: num1 = " << num1 << ", num2 = " << num2 << std::endl;

    // Call the swap function, passing num1 and num2 by reference
    swap(num1, num2);

    std::cout << "After swapping: num1 = " << num1 << ", num2 = " << num2 << std::endl;

    return 0;
}
```

In this example, the `swap` function takes two integer references (`int &a` and `int &b`) as parameters. These references allow the function to directly modify the original variables `num1` and `num2` that are passed to it without needing to return any value.

Advantages of using references in parameter passing:

1. **Efficiency**: Passing by reference avoids making a copy of the arguments, which can be more efficient, especially for large objects.

2. **Modifying Variables**: References allow functions to modify variables passed as arguments, which is not possible with call by value.

3. **Readability**: Using references can make function calls more readable, as it's clear that the function may modify the original variables.

4. **Avoiding Pointer Syntax**: References provide a more intuitive syntax compared to pointers, especially for beginners.

Output:
```
Before swapping: num1 = 5, num2 = 10
After swapping: num1 = 10, num2 = 5
```

As you can see, the `swap` function successfully swaps the values of `num1` and `num2`, and the changes are reflected outside the function. This demonstrates how references can be useful in parameter passing in C++.

If we don't use references in the `swap` function and instead pass the parameters by value (i.e., using call by value), the function would receive copies of the original variables `num1` and `num2`. Any modifications made to these copies inside the function would not affect the original variables outside the function.

Let's see what happens if we implement the `swap` function using call by value:

```cpp
#include <iostream>

// Function to swap two integers using call by value
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int num1 = 5, num2 = 10;

    std::cout << "Before swapping: num1 = " << num1 << ", num2 = " << num2 << std::endl;

    // Call the swap function, passing num1 and num2 by value
    swap(num1, num2);
```

```
    std::cout << "After swapping: num1 = " << num1 << ", num2 = " << num2 <<
std::endl;

    return 0;
}
```

Output:
```
Before swapping: num1 = 5, num2 = 10
After swapping: num1 = 5, num2 = 10
```

As you can see, the `swap` function does not swap the values of `num1` and `num2` outside the function because the function is working with copies of these variables. The changes made to `a` and `b` inside the function do not affect the original variables `num1` and `num2`.

Using call by value in this case prevents the `swap` function from achieving its intended purpose of swapping the values of the variables. This demonstrates the problem that we would face if we don't use references in parameter passing.

Yes, that's correct. In C and C++, the `malloc` function is used to dynamically allocate memory on the heap. It allocates a block of memory of the specified size and returns a pointer to the beginning of that block. This pointer can then be used to access and manipulate the allocated memory.

Here's a simple example in C:

```c
#include <stdlib.h>

int main() {
    // Dynamically allocate memory for an array of 10 integers
    int *ptr = (int *)malloc(10 * sizeof(int));

    if (ptr == NULL) {
        // Check if memory allocation was successful
        // Handle error
        return 1;
    }

    // Use the allocated memory
    ptr[0] = 10;
    ptr[1] = 20;
```

```
    // Free the allocated memory when done
    free(ptr);

    return 0;
}
```

In this example, `malloc` allocates memory for an array of 10 integers (`sizeof(int)` bytes each) on the heap. It returns a pointer to the allocated memory block, which is stored in the variable `ptr`. The allocated memory can then be accessed and used through this pointer. Finally, `free` is used to release the allocated memory once it's no longer needed, preventing memory leaks.

Yes, heap memory is allocated at runtime in C++ using dynamic memory allocation functions like `new` and `malloc`. Unlike stack memory, which is allocated and deallocated automatically as part of function calls, heap memory allocation is explicitly controlled by the programmer.

Here's an example to illustrate how heap memory is allocated at runtime:

```cpp
#include <iostream>

int main() {
    // Allocate memory for an integer on the heap
    int *ptr = new int;

    // Check if memory allocation was successful
    if (ptr == nullptr) {
        std::cout << "Memory allocation failed." << std::endl;
        return 1;
    }

    // Assign a value to the dynamically allocated memory
    *ptr = 42;

    // Print the value
    std::cout << "Dynamically allocated value: " << *ptr << std::endl;

    // Deallocate the memory
    delete ptr;

    return 0;
}
```

```

In this example:

1. We dynamically allocate memory for an integer on the heap using the `new` operator: `int *ptr = new int;`.

2. We check if the memory allocation was successful by verifying if the pointer `ptr` is `nullptr`.

3. We assign a value (`42` in this case) to the dynamically allocated memory location using the dereferencing operator `*ptr`.

4. We print the value stored in the dynamically allocated memory.

5. Finally, we deallocate the memory using the `delete` operator: `delete ptr;`.

Heap memory is managed manually by the programmer, and it's essential to deallocate memory when it's no longer needed to avoid memory leaks. Unlike stack memory, which is automatically deallocated when the function returns, heap memory needs to be explicitly deallocated by the programmer using `delete` or `free` to release the allocated memory back to the system. Failure to deallocate heap memory can lead to memory leaks, where memory is allocated but never released, resulting in inefficient memory usage and potential program instability.
—————————————————————————————————————————
A code activation record, also known as a function activation record or stack frame, is a data structure used by compilers and operating systems to manage the execution of functions or procedures. It contains information about the function's execution state, including parameters, local variables, return address, and other necessary data.

When a function is called, a new activation record is created and pushed onto the call stack. This activation record contains all the information needed for the function to execute properly. Once the function completes its execution, its activation record is removed from the stack.

Here's a breakdown of the typical components found in a code activation record:

1. **Return Address**: This is the address in memory where the program should return after the function completes its execution.

2. **Parameters**: The values passed to the function as arguments. These are typically stored in specific locations within the activation record.
```

3. **Local Variables**: Variables declared within the function. They are stored in memory locations reserved within the activation record.

4. **Saved Registers**: Registers that need to be preserved across function calls are saved in the activation record.

5. **Frame Pointer (FP)**: A pointer to the base of the activation record. It allows efficient access to parameters and local variables.

6. **Previous Frame Pointer (PFP)**: A pointer to the base of the activation record of the calling function. It enables the function to access its caller's activation record.

7. **Exception Handling Information**: Information needed for handling exceptions, such as try-catch blocks.

8. **Function Return Value**: The location where the function places its return value before returning.

The structure and organization of the activation record can vary depending on the architecture, compiler, and language used. However, the fundamental purpose remains the same: to manage the execution of functions and procedures within a program. Understanding activation records is essential for debugging, performance optimization, and understanding program execution flow.
—--
Certainly! Here's a chart illustrating the precedence and associativity of some common operators in C++:

```
---------------------------------------------------------------------------
| Operator         | Description                      | Associativity
|
---------------------------------------------------------------------------
| ()               | Function call                    | Left to Right
|
| []               | Array subscript                  | Left to Right
|
| .                | Member selection (object)        | Left to Right
|
| ->               | Member selection (pointer)       | Left to Right
|
| ++ --            | Postfix increment/decrement      | Left to Right
|
| ++ --            | Prefix increment/decrement       | Right to Left
|
```

| + - | Unary plus/minus | Right to Left
|
| ! ~ | Logical NOT/Bitwise NOT | Right to Left
|
| * | Multiplication | Left to Right
|
| / % | Division/Modulus | Left to Right
|
| + - | Addition/Subtraction | Left to Right
|
| << >> | Bitwise shift | Left to Right
|
| < <= > >= | Relational operators | Left to Right
|
| == != | Equality operators | Left to Right
|
| & | Bitwise AND | Left to Right
|
| ^ | Bitwise XOR | Left to Right
|
| | | Bitwise OR | Left to Right
|
| && | Logical AND | Left to Right
|
| || | Logical OR | Left to Right
|
| ?: | Ternary conditional | Right to Left
|
| = += -= *= /= | Assignment and compound assignment| Right to Left
|
| , | Comma | Left to Right
|

```

- **Precedence**: Higher precedence operators are evaluated before lower precedence operators.
- **Associativity**: Defines the order of evaluation when multiple operators of the same precedence appear in an expression. Left to right means the leftmost operation is evaluated first, while right to left means the rightmost operation is evaluated first.

This chart serves as a general guide, but it's always a good practice to use parentheses `()` to explicitly specify the order of evaluation if there's any ambiguity.
—---
```