C++ References, Pointers, and Memory Management

References in C++

- A reference is an alias for an existing variable.
  - When a variable is declared as a reference, it becomes an alternative name for an existing variable.
- A variable can be declared as a reference by putting ‘&’ in the declaration.
- It must be initialized at the time of declaration

```
int a = 10;
int &ref = a;

cout<<"Directly accessing the value of a ="<<a<<endl;
cout<<"Accessing the value through reference, ref = "<<ref;
```

```
Directly accessing the value of a =10
Accessing the value through reference, ref = 10
```

Different Scenarios:

```
int a = 10;
int &ref;
&ref = a;
```

error: ‘ref’ declared as reference but not initialized

```
int a = 10;
int &ref = 20;
```

error: cannot bind non-const lvalue reference of type ‘int&’ to an rvalue of type ‘int’

```
int a = 10;
int b = 20;
int &ref = a;
&ref = b;
```

error: lvalue required as left operand of assignment

Reference Example

```
void increment(int &x) {
    x++;
}

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

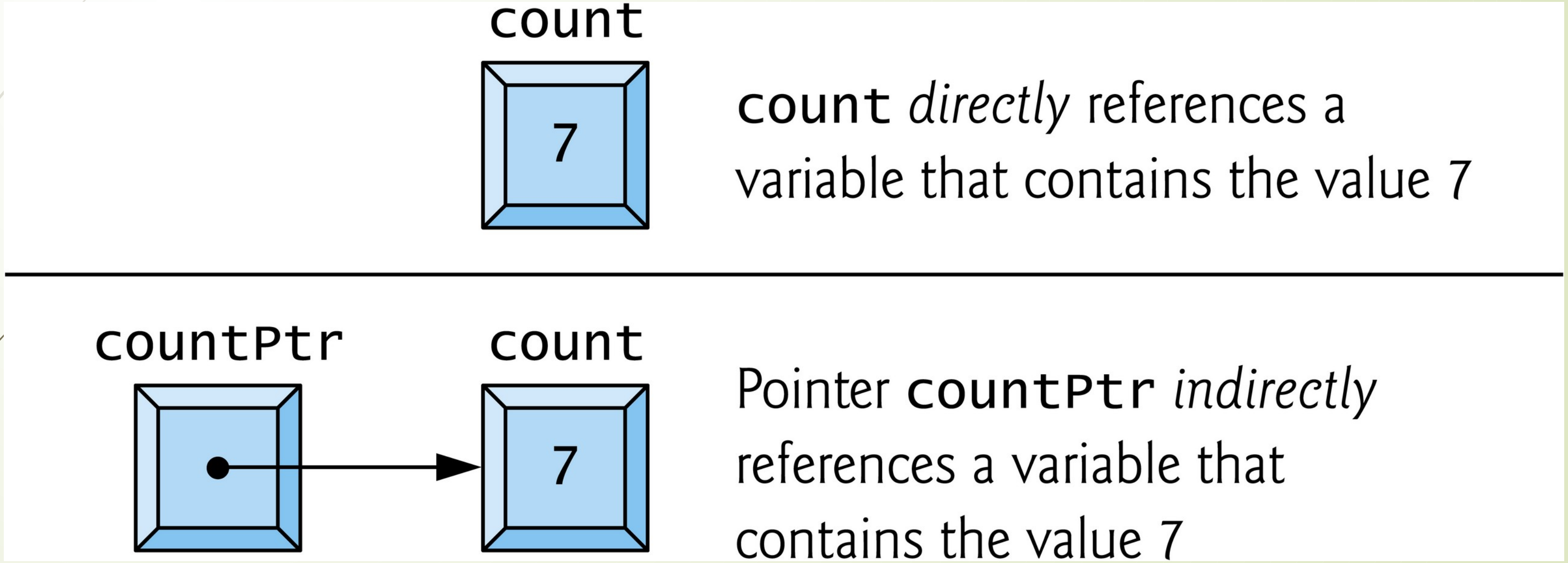
Pointers in C++

- A pointer is a variable that stores the memory address of another variable
- Declaration:

```
int a = 10;
int *p = &a;
```

- Key Characteristics:
  - Can be null (nullptr).
  - Can point to different variables.
  - Allow direct memory manipulation.

7.2 Pointer Variable Declarations and Initialization (1 of 2)



7.2 Pointer Variable Declarations and Initialization (2 of 2)

- `int* countPtr; // uninitialized "dangling pointer"`
- C++11 `nullptr`
  - `int* countPtr{nullptr}; // pointer to nothing`
- Null pointers before C++11
  - `0`
  - `NULL`

Example

```
int main(){
        int a = 10;
        int *p = &a;
        cout<<"Directly accessing the value of a ="<<a<<endl;
        cout<<"Accessing the value through pointer, *p = "<<*p<<endl;
        cout<<"Accessing the pointer itself, p = "<<p;
        return 0;
}
```

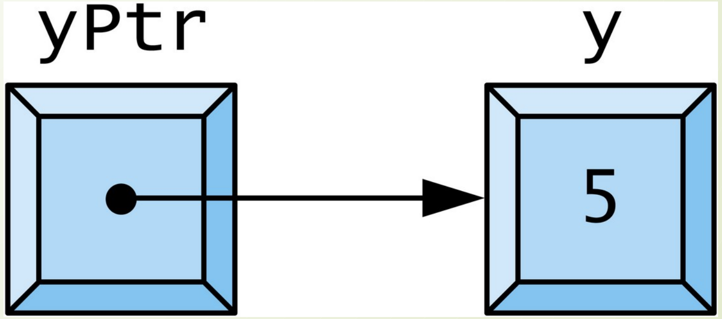
```
Directly accessing the value of a =10
Accessing the value through pointer, *p = 10
Accessing the pointer itself, p = 0x7fff581a756c
```

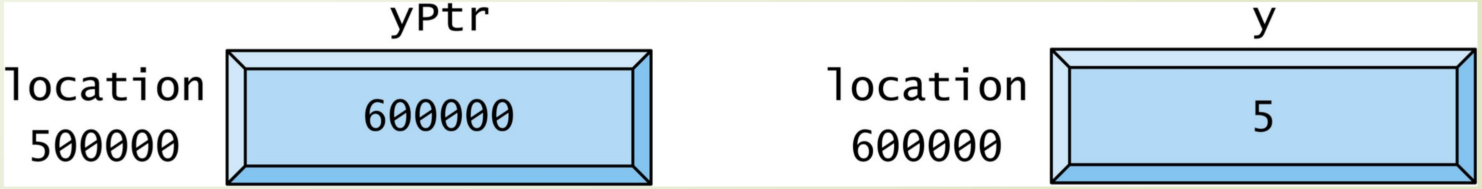
7.3 Pointer Operators (1 of 3)

- Address operator &
- Indirection operator *

7.3 Pointer Operators (2 of 3)

- `int y{5}; // declare variable y`
- `int* yPtr{nullptr}; // declare pointer variable yPtr`
- `yPtr = &y; // assign address of y to yPtr`





7.3 Pointer Operators (3 of 3)

- Applying unary * operator to a pointer results in an *lvalue* representing the object to which its pointer operand points
  - Known as the indirection or dereferencing operator
- Following statement displays y’s value (5):
  - `std::cout << *yPtr << '\n';`
- Using * in this manner is called dereferencing a pointer

Analogy: Pointers as Mailing Addresses:

- **Pointers as Mailing Addresses:**
  - Imagine that a **variable** in programming is like a **house**.
  - The **value** stored in the variable is the **person** who lives in the house.
  - A **pointer** is like the **mailing address** of the house.
- **Just like how:**
  - **A house's address** points to the actual location where the person (value) resides,
  - **A pointer** holds the memory address where the variable's value is stored.
- **You can:**
  - **Send a letter** to the address to communicate with the person (just like accessing a value via a pointer).
  - **Copy an address** to share it with someone else, allowing them to send something to the house (similar to copying a pointer to access the same memory location).
  - You don't need to move the person (value) directly. You just need the **address (pointer)** to interact with them.
- **Pointer Dereferencing:**
  - When you **dereference** a pointer (access the value it points to), it's like **visiting the house** using the address to meet the person (retrieve the value).
- **Pointer to Pointer Analogy:**
  - A **pointer to a pointer** is like having a **directory** that holds the mailing addresses of multiple houses. This directory helps you find the specific house address (pointer) and then visit the house (dereference the pointer).

Different Scenarios

```
int a = 10;
int *p;
p = &a;
```

No Error

```
int a = 10;
int *p = &a;
p = 20;
```

error: invalid conversion from ‘int’ to ‘int*’

```
int a = 10;
int *p = &a;
*p = 20;
```

No Error

```
int a = 10;
int b = 20;
int *p = &a;
*p = &b;
p = &b
```

error: invalid conversion from ‘int*’ to ‘int’

No Error

Pointers vs References

Pointers	References
Can be re-assigned to point elsewhere	Must be initialized when declared
Can be null (nullptr)	Cannot be null
Supports pointer arithmetic	Does not support pointer arithmetic
Explicit dereferencing using *	No dereferencing needed

Arrays and Pointers Relationship

- Arrays and pointers are closely related in C++.
- The name of an array acts as a pointer to the first element of the array.

```
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr;  // p points to the first
element of the array
cout << *p;    // Output: 1
```

- The name `arr` itself represents the memory address of the first element

Accessing Array Elements via Pointers

- You can use pointer arithmetic to access array elements.
- Moving to the next element in the array is equivalent to incrementing the pointer

```
int arr[3] = {10, 20, 30};
int *p = arr;

cout << *p;      // Output: 10
cout << *(p+1); // Output: 20
cout << *(p+2); // Output: 30
```

- *(p+1) accesses the second element of the array, and so on

Pointer Arithmetic in C++

- What is Pointer Arithmetic?
  - Pointer arithmetic refers to the ability to perform arithmetic operations on pointers, which allows for navigating through memory addresses.
  - The operations include incrementing (++), decrementing (--), addition (+), and subtraction (-) of pointer values.
- Key Concept:
  - The value of a pointer is a memory address.
  - When you increment or decrement a pointer, it moves to the next or previous memory location based on the data type size.

Pointer Arithmetic - Increment

- Pointer Increment:
  - When you increment a pointer, it moves to the next element in the memory based on the size of the data type.
- Example:

```
int arr[3] = {10, 20, 30};
int *p = arr;  // p points to the first element

cout << *p << " ";     // Output: 10
p++;
cout << *p << " ";     // Output: 20
p++;
cout << *p;            // Output: 30
```

- Explanation:
  - Initially, p points to `arr[0]`. After `p++`, it moves to `arr[1]`, and so on.

Pointer Arithmetic - Decrement

- Pointer Decrement:
  - Decrementing a pointer moves it to the previous memory location (or array element).
- Example:

```
int arr[3] = {10, 20, 30};
int *p = arr + 2;  // p points to the last element

cout << *p << " ";     // Output: 30
p--;
cout << *p << " ";     // Output: 20
p--;
cout << *p;              // Output: 10
```

- Explanation:
  - The pointer p starts at arr[2]. After p--, it moves to arr[1], and so on.

Pointer Addition and Subtraction

- Pointer Addition and Subtraction:
  - You can add or subtract integers from pointers, which moves them forward or backward in memory by the size of the data type.
- Example:

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;
p = p + 2;
cout << *p << " ";   // Output: 30
p = p + 2;
cout << *p << " ";   // Output: 50
p = p - 3;
cout << *p;          // Output: 20
```

- Explanation:
  - Addition of 2 takes p to the 3rd element, further addition of 2 takes p to the 5th element and subtraction of 3 takes it back to the 2nd element of the array

Pointer Difference

- Difference Between Two Pointers:
  - You can subtract one pointer from another to get the number of elements between them.
- Example:

```
int arr[5] = {10, 20, 30, 40, 50};
int *p1 = arr;          // Points to arr[0]
int *p2 = arr + 3;      // Points to arr[3]

cout << p2 - p1;       // Output: 3
```

- Explanation:
  - p2 - p1 returns the number of elements between p1 and p2, which is 3.

Pointer Comparison

- Comparing Pointers:
  - Pointers can be compared using comparison operators (e.g., ==, !=, <, >).
- Example:

```
int arr[3] = {10, 20, 30};
int *p1 = arr;
int *p2 = arr + 2;

if (p1 < p2) {
    cout << "p1 points to an earlier element than p2";
}
```

- Explanation:
  - Since p1 points to arr[0] and p2 points to arr[2], p1 < p2 evaluates to true.

Example - Traverse an Array Using Pointers

- Using Pointers to Traverse an Array:
  - You can traverse an entire array using pointer arithmetic instead of using array indexing.
- Example:

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;

while (p < arr + 5) {
    cout << *p << " ";
    p++;
}
```

- Explanation:
  - The pointer p starts at arr[0] and moves through the array using p++ until it reaches the end of the array.

Pointer Arithmetic with Multi-dimensional Arrays

- Pointers and 2D Arrays:
  - Pointer arithmetic can also be applied to multi-dimensional arrays.
- Example:

```
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
int (*p)[3] = arr;

cout << *(*(p + 1) + 2);  // Output: 6
```

- Explanation:
  - The pointer p points to the entire array, and *(p + 1) points to the second row ({4, 5, 6}). Adding 2 accesses the third element in that row.

Function Argument Pass by Reference Using Pointers

- The function takes a pointer (memory address) as an argument.
- Inside the function, the pointer is used to access and modify the original value by dereferencing the pointer.

```
void modifyValue(int *ptr) {
    *ptr = 100; // Dereferencing the pointer to modify the value
}

int main() {
    int number = 50;
    modifyValue(&number);
    cout << "After function call: " << number << endl;
    return 0;
}
```

Example: Pass by Reference and by Pointer

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

void modifyByReference(int &ref) {
    ref = 200;
}

void modifyByPointer(int *ptr) {
    *ptr = 300;
}

int main() {
    int number = 100;

    cout << "Original value: " << number << endl;

    modifyByReference(number);
    cout << "After modifyByReference: " << number << endl;

    modifyByPointer(&number);
    cout << "After modifyByPointer: " << number << endl;

    return 0;
}
```

The new Operator

- The new operator is used to dynamically allocate memory
- It returns the address of the allocated memory, which can be assigned to a pointer.
- Syntax:

```
pointer_variable = new data_type
```

- Code Example:

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

int main() {
    int *p = new int;   // Allocate memory for one int
    *p = 25;            // Assign value to allocated memory
    cout << "Value of *p: " << *p << endl;
    delete p;           // Free the allocated memory
    return 0;
}
```

Allocating Memory for Arrays Using new

- You can also dynamically allocate memory for arrays using the new operator. Memory can be allocated during runtime based on program needs
- Syntax: `pointer_variable = new data_type[size];`
- Code Example:

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

int main() {
    int *arr = new int[5];  // Allocate memory for an array of 5 integers
    for (int i = 0; i < 5; i++) {
        arr[i] = i * 2;  // Assign values
    }

    for (int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }

    delete[] arr;
    return 0;
}
```

- Memory is released using delete[].

Memory Deallocation: delete Operator

- delete Operator: Frees memory allocated using new.
- delete[] Operator: Used to free dynamically allocated arrays
- Memory Leaks: Forgetting to free dynamically allocated memory using delete.
- Double Deletion: Using delete twice on the same pointer can cause runtime errors.
- Dangling Pointers: Pointers that point to deleted memory. These should be set to nullptr after deletion.

```
delete p;
p = nullptr;
```

What is a Memory Leak?

- A memory leak occurs when a program allocates memory dynamically (usually from the heap) but fails to release it after it is no longer needed.
- The memory remains unavailable for the program or the system, even though it cannot be used anymore.
- Over time, if memory leaks accumulate,
  - they can lead to reduced system performance,
  - application crashes,
  - the system running out of memory.

A few Common Scenarios of Memory Leaks

- **Forgetting to use delete**: memory is allocated with new, but delete is never called.

```
Int* arr = new int[10];
// No delete[] arr; leads to memory leak
```

- **Exception Handling:** If an exception is thrown before delete is called, the allocated memory might never be released.

```
int* ptr = new int(10);
// If an exception occurs here, delete will not be reached
throw std::runtime_error("Error");
delete ptr;
```

- **Overwriting Pointers:** If a pointer is overwritten with a new memory address before releasing the memory it originally pointed to, the reference to the original memory block is lost, causing a leak.

```
int* ptr = new int(10);
ptr = new int(20);  // Memory for the first 'new int' is now leake
ddelete ptr;          // Only the second 'new int' is freed
```

Preventing Memory Leaks

- Manual delete: Always make sure that every new or new[] is paired with a corresponding delete or delete[].
- Smart Pointers: Use smart pointers (from the C++ Standard Library) such as std::unique_ptr or std::shared_ptr to automatically manage memory, ensuring that memory is freed when the pointer goes out of scope.

```
#include <memory>
int main() {
        std::unique_ptr<int> ptr = std::make_unique<int>(42);  // Automatically deleted when out of scope
        return 0;
}
```