Quick Reference: Structs, Pointers, Arrays, and Memory in C/C++

Structs: Grouping Data

A `struct` groups related variables under one name.

```
struct Point {
    int x;
    int y;
};
```

AS Syntax Pattern:

- Declare with `struct Name {...};`
- Use with `Point p; p.x = 10;`

CR Visual: Like a labeled box holding related slots: `x | y`

Bonus — C++ Style Struct with Constructor:

```
struct Point {
    int x, y;
    Point(int x_, int y_) : x(x_), y(y_) {}
};
```

Pointers: Memory Addresses

A pointer *stores the memory address* of a variable.

```
int a = 42;
int* ptr = &a;      // ptr points to a
std::cout << *ptr; // dereference = 42
```

AS Cheat Sheet:

- `*` declares a pointer: `int* p;`
- `&` gets the address: `&x`
- `*p` dereferences the pointer (gets the value)

CR Visual: `ptr → [42]` — like a cable connecting to a memory slot.

Arrays and Pointers

```
int arr[3] = {1, 2, 3};
int* p = arr;         // arr == &arr[0]
std::cout << p[1];    // prints 2
```

Arrays decay to pointers. You can treat the name of the array as a pointer.

Passing Pointers to Functions

Why: Let a function access/modify the original variable or structure.

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

int main() {
    int val = 10;
    increment(&val);  // pass pointer
    std::cout << val; // prints 11
}
```

AS Tip:

- Passing by pointer lets you modify values in-place

CR Analogy: You gave the function your variable's "address" — like handing over a key instead of a copy.

Pointers to Structs

```
Point pt(2, 3);
Point* ptr = &pt;
std::cout << ptr->x; // use '->' for member access
```

Syntax Rule: Use `->` to access members from a pointer to a struct.

Dynamic Memory: malloc/new

Allocate memory manually during runtime.

C Style:

```
int* data = (int*)malloc(sizeof(int) * 10);
data[0] = 42;
free(data);
```

C++ Style:

```
int* data = new int[10];
data[0] = 42;
delete[] data;
```

For structs:

```
Point* p = new Point(5, 6); // constructor call
delete p;
```

Important: Always free what you allocate!

Memory Layout Insight (CR Mental Image)

Imagine RAM as a giant warehouse with numbered lockers.

- `int x = 5;` puts 5 in locker #1000 (example).
- `int* p = &x;` gives you the key to locker #1000.
- `*p = 7;` writes 7 in locker #1000.

For arrays and structs: memory is continuous and packed.

Putting It All Together: Node Chaining Example

```
struct Node {
    std::string key;
    int value;
    Node* next;
};

Node* head = new Node{"key1", 123, nullptr};
head->next = new Node{"key2", 456, nullptr};
```

Memory layout:

$$[\text{head}] \rightarrow [\text{key1}, 123] \rightarrow [\text{key2}, 456] \rightarrow NULL$$

This is the core of chained hash tables!

Summary Cheatsheet

- `struct S { ... };` defines a custom type
- `S s;` or `S* sp = new S();`
- `*p` = value pointed to; `&x` = address of x
- `malloc/free` = C style, `new/delete` = C++ style
- Pass pointers to functions for in-place modification
- Use `->` to access members via pointers
- Arrays and pointers are deeply connected