Pointers

Introduction to pointers

Pointers are variable that stores memory address of another variable.

`int *p1 = NULL` <-- This line declares the integer pointer variable and set it to NULL.

`p1 = &a` will reset the pointer p1 to the memory location of the variable "a" from NULL. Operator '&' before any variable gives the memory location to the variable.

- Dereferencing the Pointer - *p1

*p1 Outputs the value stored in the position p1. If the pointer is of the type int then it will read out the next four bytes. Thats why pointers have strict types.

To change the pointer we have to explicitly change the pointer. `*p1 = b` will only change the value at the location to b.

```
    *p1 = b;
    *p1 = 8; // This will change the variable from b to 8
```

To change the pointer we have to change the pointer explicitly like so,

```
    p1 = &b;
```

Pointer type, Void Pointer and Pointer Arithmetic

Pointers have strong type. This means that we have to initiate pointers to a specific type only. This will help us to deference pointers as well as use type specific operations on pointer. We will also be able to call the pointers in the function with type specified.

Pointer initiation	Pointer Type
*int	Integer
*char	Character

- Void Pointer - Have no value associated with hance no dereferencing.

```
void *P0;
P0 = __SOME_POINTER__
```

pointing the void pointer is still allowed. But not calling.

- Pointer arithmetic

```
int *p1, *p2;
int a;
char c;
p1 = &a;

printf (p1+1);
printf (p2+2);
```

p1 + 1 is p1 plus 4 bytes

p2 + 1 is just 1 byte

because the pointer p1 is integer type an dp2 is character type. only arithmetic on the pointer possible are just addition and subtraction.

- Type casting

```
p1 = (char*) p2;
```

this is called Typecasting. The pointer p1 is integer type which is casted to p2 a character type. The character type pointer will only look at the first byte of the integer type pointer.

also

```
p1 = p2
```

will give the compiler error because these are different type of variables.

Pointers to Pointer

In C++ we can have a pointer pointing to another pointer and so on. Pointers have a strict type and hance they can only point to the type. For example:

```
    int a = 9;
    int *p = &a;
    int **q = &p;
    int ***r = &q;
```

for manipulation, `***r = **q + 30` will change the value of a to 39. furthermore, `**r` is `p`

As a thumb rule

> ptr *(type)

This translate to following, If the type is pointer itself then `ptr *(*ptr)` should be used to create a pointer to the pointer and so on.

Pointers as a Function Argument

We can also use a pointer as a Function argument. There are some benefits of using pointer as a function arguments.

```
void Increment(int *a)  // call by reference
{
    *a = *a + 1;
}

int main() {
    int a = 10;
    Increment (&a);
    cout << a;
}
```

In this code snippet, We passed the location of the variable `a` not the `a` itself. If we would've just passed `a`, this code will not run because `a` in the function `Increment()` is local and will be destroyed from the stack once the function is executed. Hance passing `a` as a reference will be useful.

Pointers and Arrays

Arrays are the data structure stored in the block of memory. Individual elements are stored next to each other.

- **Element at index i -**
  - **address - `&A[i]` or `(A+i)`**
  - **value - `A[i]` or `*(A+i)`**

`A` will just give us the **base address** of the array. **That is address of the first element.**

When arrays are passed to a functions they are explicitly passed as a reference not as a array. Arrays can be very big hance copying it wastes resources.

Arrays as a Function Argument

When passing the Arrays as a function arguments we need to take care to not copy the whole array into the new array. Because this will cause unnecessary memory usage.

The solution is to pass an array's first element address.

```
void __SOMEFUNCTION__ (int a[][2][2]){}

int main() {
    int a[3][2][2];
    __SOMEFUNCTION__(int a[][2][2])
}
```

For example in the previous code, we have passed the head pointer of the first array. The head pointer to multidimensional array follows like so,`A[][i][j][k]......` and de referencing will result in the first element of the array. any pointer arithmetic will result in moving through the array.

Character Arrays

Strings in C are just character arrays.

```
char a[5];
a[1]= 'J'
a[2]= 'O'
a[3]= 'H'
a[4]= 'N'
a[5]= \0
```

Strings have special termination rule. They must terminate with the `\0` and hance a four character string will always have a length = 5.

When character arrays are initialized implicitly like in the previous example the termination should also be stated implicitly. but if it is initialized explicitly the termination is explicit.

In C++ the strings are not NUL terminated. They have these functionalities built in.

```
#include<string>
int main(){
    std::string a = "alpha";
    cout << a << std::endl;  // alpha
}
```

String are of objects of Standard template library (STD)

Pointer and Multidimensional array

Multiple Dimensional array is arrays inside an array.

- `int A[2]` is how you initiate array in C++.
- `int B[2][3];` is how you initiate multidimensional arrays in C++.

```
    int A[5];
    int *p = A;
    std::cout << A; // print memory location of the first element in the
memory.

    // *p is same as A.
```

As arrays are in contiguous block of memory, We can use the first element to traverse through the array. This is handy and useful. **Just the name of the array can be passed as a pointer to the first element**. To retrieve the element from the array we just have to dereference the array. (`*A` for first element and `*(A+i)` for ith element).

While assigning the pointer to an array, this does not work, `A = alpha` because the base pointer is constant pointer because if that changes the whole array need to move to the new location, not a smart idea.

Here is how you assign a pointer to a multidimensional array :

```
    int B[2][3];
    int (*beta)[3] = B; // int (*ptr)[i] is the integer array pointer of i
elements and it points to the first element of that multidimensional array.
```

The thing to note is to use the correct pointer type for the elements. B as it is, is the address of the first element. In this case it would be a integer array of 3 elements. so the type of pointer should be integer array pointer of 3 element.

Pointer and Dynamic Memory

Application when run have some memory assigned to it by the operating system. There are 4 parts of memory allocated to application.

- Code -- text of code
- Static/Global -- Global variable, Stored for the life time of the programme
- Stack -- Local variable and methods, Stored only for the life time of the method. Memory is freed automatically and as soon as the method returns. Fixed memory size
- Heap -- Stores variables through pointer and only when explicitly stated. must be deleted manually. Memory can grow as long as machine have memory.

To allocate variables in the Heap you use `new` and to delete you use `delete` syntax in C++.

```
int a;
int *p;
p = new int;
*p = 10
delete p;

p new int [20];
delete[] p;
```

`New` will reserve the appropriate size and returns the pointer to the first byte of the storage. `delete` free the storage in the heap at the location. To write at the location we have to use the reference to the location.

Pointers as a Function return

Functions can return pointers just like they can return any other data type. This is useful when you want the function to provide a way to access or modify data that is stored outside of the function's local scope.

But returning a pointer to a local variable within a function can be dangerous because the memory used by the local variable is deallocated when the function returns. This can lead to the pointer pointing to invalid memory, causing program crashes or unexpected behavior.

To avoid this use pointer to store the returned variable to the heap and not stack. In heap the memory allocation is protected and can not be over written. Also programmer must take care and always deallocate the memory used in heap with function like `delete` in C++.

This code snippet shows the similar concept.

```
#include <iostream>

// Function to add two integers and return a pointer to the sum
int* add(int a, int b) {
  // Allocate memory on the heap to store the sum
  int* sum = new int;
  *sum = a + b; // Store the sum in the allocated memory

  // Return the pointer to the allocated memory
  return sum;
}

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

  // Call the add function and store the returned pointer
  int* result = add(num1, num2);

  // Access the sum using the dereferenced pointer
  std::cout << "Sum: " << *result << std::endl;

  // Important: Deallocate the memory after use
  delete result;

  return 0;
}
```

Function Pointers

concept

- Function pointers are special variables that store memory addresses of functions.
- They allow you to call functions indirectly through the pointer variable.

Declaring a Function Pointer

```
// Function pointer declaration with return type and arguments
int (\*function_pointer_name)(int, int);
```

- `int` : Return type of the function the pointer points to.
- `(*function_pointer_name)` : Function pointer variable name enclosed in parenthesis.
- `(int, int)` : Argument types of the function the pointer points to.

Initializing a Function Pointer

```
function_pointer_name = &function_name;
```

- `&` : Address-of operator.
- `function_name` : Name of the function whose address is being assigned.

Dereferencing a Function Pointer

```
(\*function_pointer_name)(arguments);
```

- `*` : Dereference operator.
- `function_pointer_name` : Function pointer variable.
- `(arguments)` : Arguments passed to the function.

Example Code:

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int main() {
// Function pointer declaration
    int (\*p)(int, int);

// Initialize the pointer to point to the add function
    p = &add;

// Dereference the pointer to call the add function
    int result = (\*p)(2, 3);

// Print the result
    printf("The sum is: %d\n", result);

return 0;
}
```

Function Pointers and Callbacks in C++

Concepts

- **Function Pointers**: Variables that store memory addresses of functions.
- **Callbacks**: Functions passed as arguments to other functions, allowing them to be called later.

Example: Sorting with Callbacks

This example demonstrates sorting a list of integers using function pointers and callbacks.

1. Define the data structure

```
#include <vector>
std::vector<int> data = {5, 2, 9, 1, 8};
```

2. Define the comparison function

This function takes two integers and returns a negative value if the first element is less than the second, zero if they are equal, and a positive value otherwise.

```
bool compareAscending(int a, int b) {
  return a < b;
}
// Similar functions can be defined for descending order or other criteria.
```

3. Sorting function with callback

This function takes the data vector and a comparison function pointer as arguments. It uses the standard library `std::sort` function with the provided comparison function.

```
void sortByPointer(std::vector<int>& data, bool (*comparison)(int, int))
{
  std::sort(data.begin(), data.end(), comparison);
}
```

4. Usage

```
// Sort in ascending order (using the default comparison function)
sortByPointer(data, compareAscending);

// Sort in descending order (using a modified comparison function)
bool compareDescending(int a, int b) {
  return a > b;
}
sortByPointer(data, compareDescending);
```

Explanation

- The sortByPointer function takes the data vector and a comparison function pointer (comparison).
- Inside the function, `std::sort` is used to sort the data.
- The provided comparison function (compareAscending or compareDescending) is used to determine the order during sorting.