Advanced Programming

Pointers

Pointers are a fundamental concept in C++ programming, essential for understanding memory management, dynamic data structures, and efficient computation. This lecture will delve into the intricacies of pointers, covering their declaration, usage, and the associated memory management strategies. This supplementary material will provide a comprehensive overview of pointers, preparing you for lecture 4.

1 Memory Management in C++

C++ provides several mechanisms for memory management, including automatic, dynamic, and static memory allocation. Before going through this, let's see a brief definition of the memory allocation areas.

1.1 Stack and Heap

Memory allocation can occur in two main areas: stack and heap. The stack is used for automatic memory allocation and operates following a Last-In-First-Out (LIFO) structure. It is managed by the program's runtime environment and is of fixed size. The heap, on the other hand, allows for dynamic memory allocation and deallocation. It is slower than the stack but provides flexibility in resizing variables. Memory fragmentation and memory leaks can occur in the heap if not managed properly.

1.2 Static Memory Allocation

Static memory is allocated at compile time and deallocated automatically when the program terminates. This memory is stored in RAM next to program data, not in the stack.

```
1 static int x = 10; // x is statically allocated
```

1.3 Automatic Memory Allocation

Automatic memory allocation occurs when a variable is declared, and memory is allocated on the **stack**. Local variables are stored in the stack and are automatically deallocated when the function returns.

```
1 void function() {
2     int x = 10; // x is allocated on the stack
3 }
```

1.4 Dynamic Memory Allocation

Dynamic memory allocation happens during runtime using the `new` operator and must be manually deallocated using the delete operator. This memory is stored in the **heap**.

```
1 int* p = new int; // dynamically allocate memory for an int
2 *p = 5;
3 delete p; // deallocate the memory
```

You might have noticed the star (*) before the variable name in this example. That's a pointer declaration. In this case `int *p` declares a pointer, whilst `new int` allocates memory location for an integer. Therefore, pointers allow us to directly access and manipulate the value stored at a specific memory location.

2 Understanding Pointers

A pointer is a variable that stores a memory address. The key advantage of pointers is the ability for dynamic memory management. This allows for efficient array and data structure manipulation.

2.1 Pointer Declaration and Initialisation

Pointers are declared by placing a star (*) before the pointer name. Recalling the previous example:

```
int* P = new int;
*P = 5;
```

In this case, the pointer `P` is initialised with an integer value. However, a pointer can also be initialised using the memory address of another variable. For this, we make use of the symbol &:

```
int A = 10;
int* P = &A; // P is a pointer to an int,
             // initialised with the address of A
```

This also works in the opposite way, i.e. the value stored in a memory address can be accessed using the star (*) symbol.

```
int A = 10; // sets variable A with the integer value 10
int* P = &A; // sets pointer P with the memory address of variable A
int B = *P + 1; // sets variable B with the integer value stored
                // at memory address of P
```

Try yourself the following example and check the output on the screen.

```
#include <iostream>
using namespace std;
int A = 10; // sets variable A with the integer value 10
int* P = &A; // sets pointer P with the memory address of variable A
int B = *P + 1; // sets variable B with the integer value stored
                // at memory address of P
int main() {
    cout << "value stored in variable A: " << A << endl;
    cout << "memory address of variable A: " << &A << endl;
    cout << "value stored in pointer P (memory address): " << P << endl;
    cout << "value stored in the memory address P (int): " << *P << endl;
    cout << "value stored in variable B: " << B << endl;
    cout << "memory address of variable B: " << &B << endl;
    return 0;
}
```

You should have noticed that memory addresses look like a random string of characters. This is actually an hexadecimal number that identifies a memory location. Don't worry about how it looks, just think about it as the post address of a storage room and the pointer name is the key to open the room. Consequently, you can store and remove different things from that room. This means that pointers allows for allocating memory space for variables with a size not known yet, or variables with a size determined by future calculations in the code.

3 Types of Pointers

3.1 Null Pointers

A null pointer is a pointer that does not point to any valid memory location. It is initialised with `nullptr`:

```
int *p = nullptr;
```

3.2 Void Pointers

A void pointer is a pointer that can point to any data type. It is declared using the void keyword.

```
void *vp;
int a = 5;
vp = &a;  // vp now points to an integer
```

3.3 Constant Pointers

A constant pointer is a pointer that cannot change the address it is pointing to after initialisation.

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

3.4 Pointer to Constant

A pointer to a constant is a pointer that cannot change the value of the variable it is pointing to.

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

4 Pointers and Arrays

Arrays and pointers are closely related. The name of an array variable is also a pointer to the memory location of the first item in the array. This allows us to manipulate arrays using pointer arithmetic. For example:

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

Pointers can be incremented and decremented using arithmetic operations, allowing traversal through arrays.

```
int arr[5] = {1, 2, 3, 4, 5};
int* p = arr; // p points to the first element of arr
for (int i = 0; i < 5; ++i) {
    cout << p[i] << endl; // prints the elements of arr
}
```

Additionally, adding values to a pointer allows to move to a different position in the array

```
int arr[5] = {1, 2, 3, 4, 5};
int* p = arr; // p points to the first element of arr
cout << *p << endl; // prints the first element of arr
p += 1; // move one position in the array
cout << *p << endl; // prints the second element of arr
for (int i = 0; i < 5; ++i) {
    cout << *(p + i) << endl; // prints the elements of arr
}
```

5 Pointers to Pointers

A pointer to a pointer is a variable that stores the address of another pointer. This is useful for multidimensional arrays and dynamic memory allocation.

```
int a = 10;
int* p1 = &a;
int** p2 = &p1;
std::cout << **p2 << std::endl; // prints 10
```

This is useful for defining 2D arrays, given the relation between pointers and arrays. Consider a 2D array as an array of arrays: `A = [a1, a2, a3]`, with `a1=[1, 2, 3]`, `a2=[4, 5, 6]`, `a3=[7, 8, 9]`. Likewise, a 2D array can be defined using pointers:

```
int rows = 3;
int cols = 3;
int** A = new int*[rows];
for(int i = 0; i < rows; ++i)
    A[i] = new int[cols];
```

At this point we have created an array of size `rows`, with each element being an array of size `cols`. Our array is not populated with values yet. This can be achieved in different ways:

```
int val = 1; // variable for the values to be stored
for (int i = 0; i<rows; ++i) {
        int *a = new int[cols]; // initialise a temporary array for each row
        for (int j = 0; j<cols; ++j) {
                a[j] = val; // store the values in the temporary arra
        ++val; // update the value
    }
    A[i] = a; // store the temporary array in our 2D array
    delete[] a; // deallocate memory
}
```

```
int val = 1; // variable for the values to be stored
for (int i = 0; i<rows; ++i) {
        for (int j = 0; j<cols; ++j) {
                A[i][j] = val; // store the value in the array
        ++val;
    }
}
```

You can try yourself and identify the advantages or disadvantages of different implementations.

6 Passing Pointers to Functions

Functions can accept pointers as parameters, allowing them to modify the value of the pointed-to variable.

Functions in C++ can have pointers as parameters. By passing a pointer to a variable, the function can modify the value stored at that memory location. This is particularly useful for functions that need to return multiple values or work with arrays. For example:

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

void traverse_arr(int* p, int l) {
    for (int i = 0; i<l; ++i)
            cout << p[i] << endl;
}

int main() {
    int a[5] = {1,2,3,4,5};
    int len_a = sizeof(a)/sizeof(a[0]);
    traverse_arr(a, len_a);
    return 0;
}
```

Using a pointer allows passing an array as a parameter to `traverse_arr(int* p, int l)`. Passing the array without using a pointer results in compilation errors.

Additionally, you can return a pointer from a function:

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

int* createArray(int size) { // return type of the function is an int pointer
    int *arr = new int[size];  // dynamic memory allocation for the array
    for (int i = 0; i < size; ++i) {
        arr[i] = i * 2;  // assign values to array elements
    }
    return arr;
}

int main() {
    int size = 5;
    int *array = createArray(size);
    for (int i = 0; i < size; ++i) {
        cout << array[i] << " ";
    }
    delete[] array; // Free the allocated memory
    return 0;
}
```

In this case `int* createArray(int size)` declares a function that takes an integer `size` and returns a pointer to an array of integers. Inside the function, memory is dynamically allocated for the array (`int *arr = new int[size]`) and its elements are initialised. Finally, in the main function, the returned pointer is assigned to `array`.

7 Smart Pointers

Smart pointers in C++ provide automatic memory management by using reference counting or other mechanisms to ensure that objects are properly deleted when they are no longer needed.

7.0.1 `std::unique_ptr`

A `std::unique_ptr` is a pointer that owns a memory resource exclusively and cannot be copied or shared. When the `std::unique_ptr` goes out of scope, the memory resource is automatically deleted.

Let's see an example using a pointer:

```cpp
#include <iostream>
void myFunction() {
    int* p = new int(10);  // Allocate an integer with value 10
    std::cout << *p << std::endl;
    delete p; // remember to deallocate
    }
int main() {
    myFunction();
    return 0;
}
```

Now using a `std::unique_ptr` smart pointer:

```cpp
#include <iostream>
#include <memory>  // header file required for smart pointers
void myFunction() {
    std::unique_ptr<int> p(new int(10));  // Allocate an integer with value 10
    std::cout << *p << std::endl;
    } // The pointer is automatically deallocated here
int main() {
    myFunction();
    return 0;
}
```

7.0.2 `std::shared_ptr`

A `std::shared_ptr` is a smart pointer that allows to manage the memory resource by different instances, while maintaining reference counting. This allows to use pointed object in different scopes since the memory keeps allocated until the last `std::shared_ptr` pointing to an object is destroyed.

```
#include <memory>
std::shared_ptr<int> p(new int(20));
std::shared_ptr<int> q = p;  // Both p and q now own the memory resource
```

7.0.3 `std::weak_ptr`

A `std::weak_ptr` is a smart pointer that does not own the memory resource but can access it if it exists. It is used to keep track of `std::shared_ptr` instances, allowing for preventing the use of regions of memory managed by a pointer that was already deleted.

```
#include <memory>
std::shared_ptr<int> p(new int(30));
std::weak_ptr<int> wp = p;  // wp does not own the memory resource
```

8 Common Pitfalls and Best Practices

- Memory Leaks: Always deallocate memory using delete or delete[] to avoid memory leaks. The use of smart pointers prevents this issue.
- Avoid Dangling Pointers: Ensure that pointers are not used after the memory they point to has been deallocated.
- Pointer Arithmetic Errors: Incorrect pointer arithmetic can lead to accessing invalid memory locations.
- Double Deletion: Deleting the same memory twice can lead to undefined behavior.

References

- Stroustrup, B. (2013). The C++ Programming Language. Addison-Wesley.
- Meyers, S. (2005). Effective C++: 55 Specific Ways to Improve Your Programs and Designs. Addison-Wesley.
- Sutter, H., & Alexandrescu, A. (2004). C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Addison-Wesley.
- Josuttis, N. M. (2012). The C++ Standard Library: A Tutorial and Reference. Addison-Wesley.
- Lippman, S. B., Lajoie, J., & Moo, B. E. (2012). C++ Primer. Addison-Wesley.
- Eckel, B. (2003). Thinking in C++. MindView LLC. Available at: http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html.