

1 Week 9

As you begin to do more complex things in C++, managing memory is a crucial aspect of effective programming. The language provides several ways to interact with memory, including manual management through pointers, as well as using containers like vectors and strings. Understanding how memory is used and managed in C++ can help you write more efficient and safe code.

When working with arrays or dynamic memory allocation in C++, you have direct control over the memory location where your data is stored. This means that you can manipulate the memory directly, which can be useful for when you have to do many small operations. However, it also means that you must be careful not to access invalid memory locations or write beyond the allocated bounds, as this can lead to undefined behavior, crashes, or memory leaks.

Pointers are a key component of C++'s memory management system. They allow you to store the memory address of another variable, providing direct access to the data stored at that location. This can be useful for creating complex data structures and algorithms, as well as for manipulating large amounts of data in efficient ways.

1.1 Pointers

Pointers are variables that store the memory address of another variable. A memory address will be a sequence of numbers and letters encoding the hexadecimal address. Think of it like a street address for a house. Just as every house has a unique street address that identifies its location on Earth's surface, every piece of data in a computer program has a unique memory address that identifies its location in the computer's memory. This allows the computer to quickly locate and access the data when it is needed.

To declare a pointer, we need to specify the type of the variable it's pointing to. So if it's meant to be the memory address for an `int` we should declare the variable name to be type of a pointer to an `int` (`int*`). To make an assignment to the pointer we declared, we need a way of capturing the memory address of a variable. Use the `&` symbol before the variable name. For example,

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

(Recall, what was another way we used `&`? Perhaps setting a reference to a variable like `&ref_to_x = x;`). Furthermore, you could have a pointer that points to a memory address (a pointer to a pointer) and so on. Dereferencing a pointer using the dereference operator (`*`) allows you to access the data stored at the memory location pointed to. In the example above, `x` and `*p` can both be used to obtain the value 10. In practice, it's common we have pointers to a container rather than a fundamental type. Suppose we have a pointer `ptr` that points to a container `y`. We can access an element in `y` using the square brackets like `y[i]` or we can use `ptr[i]` to also access the `i`th element. What happens is that `ptr` points to the memory address of the 0th element of `y`. Then it moves by an offset of `i` (times the variable size) in memory to the new location of the `i`th element. Finally, it accesses the value stored at that new memory location which corresponds to the `i`th elements of the container `y`. This highlights a few important things: (i) containers like vector and Carray have contiguous memory addresses and because of this, arithmetic works with pointers since their memory addresses are neighbors.

Pointer arithmetic operations can be performed just like any other variable. The following performs the same arithmetic:

```
cout << 2 * x << endl;
cout << 2 * *p << endl;
```

1.2 CArrays

In C++, the term “Carrays” typically refers to C-style arrays, which are native arrays provided by the C programming language and also available in C++. These arrays have some differences compared to C++’s standard library containers you’ve used like `std::vector`.

For example, Carrays have a static size. They cannot change the number of elements they hold like vectors can. Carrays don’t care if you try to access something outside the maximum size. This means accessing elements outside the array’s bounds can lead to undefined behavior or memory corruption. They’re similar to vectors in that they hold elements of the same types (i.e. only `ints` or only `chars`). Also, like vectors the elements are held in contiguous order and can be indexed using square brackets (`[]`) starting at 0.

Carrays are often used to represent data in nature so that mathematical calculations can be done. For example, a CArray might be able to capture temperatures on a map. So then each location in the CArray can be thought of as corresponding to a location. We can then do mathematical operations like finite derivatives if we have such a sample of the temperatures in the area. Temperatures at one location are going to affect temperatures at other locations which are closest to it. When we use a CArray to hold temperatures, we are also storing in such an organized way. A data point of the temperature at one location will be next to a temperature data point that’s also closest to it. Then a program can be written such that it uses the current point and the neighboring points to compute the change in temperature over the space.

1.3 Exercises

1.3.1 Predict the output exercises

Write the console outputs (not including the `Program ended with exit code: 0` line). Assume the code runs error-free, includes the appropriate headers, and is using `std` namespace.

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

   int main() {
       int x = 1;

       cout << "Value of x: " << x << endl;
       cout << "Address of x: " << &x << endl;

       int& y = x;

       cout << "Value of y: " << y << endl;
       cout << "Address of y: " << &y << endl;

       int* p = &x;
```

```

cout << "Value of p: " << p << endl;
cout << "Address of p: " << &p << endl;
cout << "Dereferencing p: " << *p << endl;

int** pp = &p;

cout << "Value of pp: " << pp << endl;
cout << "Address of pp: " << &pp << endl;
cout << "Dereferencing pp: " << *pp << endl;
cout << "Double dereferencing pp: " << **pp << endl;

return 0;
}

```

2. Write out the contents of the following initialized arrays.

```

#include <iostream>
using namespace std;

int main() {
//    char str1[3] = "abc"; // array of chars
char str2[] = "abc";
cout << sizeof(str2) << endl;

int x[] = {1,2,3};
int y[5] = {1,2,3};
int z[4] = {1};
int w[3] = {0};

int arr2D[4][3] = {
    { 1 },
    { 0, 1 },
    { [2]=1 },
};

int another_arr2D[4][3] = {1, 3, 5, 2, 4, 6, 3, 5, 7};

int designate_arr2D[4][3] = {[0][0]=1, [1][1]=1, [2][0]=1};

return 0;
}

```

```

3. #include <iostream>
   using namespace std;

   int main() {
       int a = 32, *ptr = &a;
       char ch = 'A', &cho = ch;
       cho += a;
       *ptr += ch;
       cout << a << ", " << ch << endl;

       return 0;
   }

```

```

4. #include <iostream>
   using namespace std;

   int main() {
       int num[5] = {-1};
       int* p;

       p = num;
       *p = 10;

       p++;
       *p = 20;

       p = &num[2];
       *p = 30;
       p = num + 3;
       *p = 40;

       p = num;
       *(p + 4) = 50;

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

       return 0;
   }

```

```

5. #include <iostream>
   using namespace std;

   int main() {
       int x[5] = {1, 2, 3, 4, 5};
       int* p = x;

       for (int i = 0; i < 2; i++) {
           int temp = *(p + i);
           *(p + i) = *(p + 4 - i);
           *(p + 4 - i) = temp;
       }

       for (int i = 0; i < 5; i++)
           cout << x[i] << " ";
       cout << endl;

       return 0;
   }

```

```

6. #include <iostream>
   using namespace std;

   void update(int* p) {
       int q = 10;
       p = &q;
   }

   int main() {
       int r = 20;
       int *p = &r;
       update(p);
       cout << *p << endl;

       return 0;
   }

```

1.3.2 Error spotting exercises

Error spotting will be on the exam. Look at the follow code below and state:

- What errors occurred (build, run time, or logic) or if there are none.

- In the case of a logic error, what does the code currently accomplish?
- How the code can be fixed (use the line numbers to help with the description).

A description of how the code should perform will be put in a block comment. After you make your initial guess, type the code in your IDE and run it. Copying very carefully may help you find some of the errors. Play around with the code to fix it.

```
7.  1  #include <iostream>
    2  using namespace std;
    3
    4  int main() {
    5      int arr1[5] = {1, 2, 3};
    6      cout << arr1[4] << endl;
    7
    8      int arr2[3] = {1, 2, 3};
    9      cout << arr2[3] << endl;
   10
   11      int arr3[4];
   12      arr3[0] = 1;
   13      arr3[3] = 4;
   14      cout << arr3[1] << " " << arr3[2] << endl;
   15
   16      return 0;
   17  }
```

```
8.  1  #include <iostream>
    2  using namespace std;
    3
    4  // This code is meant to print out the
    5  // elements 1, 2, 3 of the array arr.
    6  int main() {
    7      int arr[3];
    8      arr = {1, 2, 3};
    9      for (int i = 0; i < 3; ++i) {
   10          cout << arr[i] << " ";
   11      }
   12
   13      return 0;
   14  }
```

```
9.  1  #include <iostream>
    2  using namespace std;
```

```

3
4  int main() {
5      int arr[];
6      arr[0] = 1;
7      arr[1] = 2;
8      cout << arr[1] << endl;
9
10     return 0;
11 }

```

10.

```

1  #include <iostream>
2  using namespace std;
3
4  /* We expect the output of this code to be
5  1
6  1
7  */
8  void inc(int* p){
9      ++(p);
10 }
11
12 int main() {
13     int r = 0;
14     int *p = &r;
15     inc(p);
16     cout << r << endl;
17     cout << *p << endl;
18
19     return 0;
20 }

```

1.3.3 Practice coding exercises

11. Write a function that prints out an array. Here some code to start:

```

void printer(int* p, int size) {
    return;
}

```

Create a few arrays to test this function.

12. Write a function that takes in an array `arr` and prints out the index `i`, `arr[i]` times on one line. Example:

```
{3, 2, 1} ->
```

```
000
```

```
11
```

```
2
```

```
{2, 2, 3, 1} ->
```

```
00
```

```
11
```

```
222
```

```
3
```

You can start with this:

```
void myFun(int* p, int size) {  
    return;  
}
```

13. **Game of Life (1D array)** This activity has you simulate the Game of Life. This is a simple cellular automata model that simulates the emergence of complex patterns from simple rules. It starts with a grid of cells, each of which can be either alive or dead. These cells interact with their neighbors according to a set of rules. This first version we will have the grid represented by a 1D array. The grid will be periodic, so imagine the population is living in a circle (the first element is neighbors with the last).

The rules for the 1D model are as follows:

- Overcrowding: If a cell has both neighbors alive, it dies.
- Loneliness: If a live cell has no neighbors alive, it dies.
- Reproduction: If a dead cell has exactly one neighbor alive, it becomes alive. Survival: Any other live cell remains alive, and any other dead cell remains dead.

So you will start with a grid representing the current population. Then apply the rules to each element on that grid to figure out what a new grid looks like for the next generation.

- Initialization** Begin by creating your first population. Define a variable for the grid size and assign it the value 10. Initialize your first grid to be a boolean array, the same size as the specified grid size. Assign the variables to be all `false`. Then replace the value at the indices 1 and 2 with `true`. Write a function that will print your population. You can use a symbol like `*` for true (alive) and a symbol like `-` for the false (dead). Test it on the array you created.
- Simulate generations** Initialize another new grid. It doesn't matter what it has, all the elements will be reassigned. Write a `for` loop that goes through 5 generations of cell life. Each loop will create a new grid. The new grid will be updated cell-by-cell

by the function `updateCell` using a nested for loop. Afterwards, the old grid should be updated cell-by-cell with the new one. Print the grid and some statement for each new generation. For example:

Generation 1:

```
-***-----
```

Generation 2:

```
**-----
```

Generation 3:

```
*****--*
```

- (c) **Add Functions** There will be two functions to write with the following function signatures:

```
int countLiveNeighbors(int index, const bool grid[], int size);
void updateCell(int index, bool grid[], bool newGrid[], int size);
```

Use the rules above to predict the population in the next generation. The rules above depend two things: the current state of the cell (dead or alive), and the amount of surrounding neighbors which are alive. To better organize are code, write some functions to perform the checks and to update the individual cell's state.

- (d) Create the function `countLiveNeighbors`. This will return an integer of the live neighbor count. In the function, initialize a variable to count the live neighbors with the `int` 0. Next, we will count the live neighbors. Check if the neighbor to the right (`index+1`) is alive (`true`) and the neighbor to the left (`index-1`) is alive (`true`). Add the results to the live neighbor count. One tricky part will be counting the neighbors if you are at the first or last index. You can use `% size` to correct for the right periodic neighbors and `% size` after adding the `size` for the left periodic neighbors. Have the function return the number of a live neighbors.
- (e) Create the function `updateCell`. Begin by updating the new cell with the current cell state. This will take care of the cell update if no changes need to be made based on the rules. Next, implement some control flow for the specific rules above. Create and assign a variable `isAlive` to store the boolean for the current cell state. Count the number of live neighbors for that current cell using the function `countLiveNeighbors`. Based on the number of live neighbors of a current cell and whether or not the cell is alive, write some control flow to change the new cell state if needed.
- *Aside: Notice you haven't had to pass the grids by reference in this exercise. This is because C arrays are decayed to pointers when pass through functions. The pointer will point to a memory address to be checked and changed.
- (f) Test your code! Try various initial cell populations and see how they grow with each generation.

14. **Game of Life (2D array)** Try the same thing. This time the cell population is in a period grid. You have some flexibility in choosing who you want to be the neighbors: the four adjacent and above, the four diagonal to the cell, or the eight of them in combination.

The rules for the 2D model are as follows:

- Overcrowding: If a live cell has more than three live neighbors, it dies.
- Loneliness: If a live cell has less than two live neighbors, it dies.
- Reproduction: If a dead cell has exactly three live neighbors, it becomes alive.
- Survival: Any other live cell remains alive, and any other dead cell remains dead.