CSCI 2270 – Data Structures

Recitation 3, Spring 2022

Pointers and Dynamic Memory

Objectives

1. Compile Time Memory Allocation
2. Run Time (Dynamic) Memory Allocation
3. Freeing Memory
4. Pass-by-value vs Pass-by-pointer vs Pass-by-reference
5. Exercise

1. Compile Time Memory Allocation

When we declare variables as shown below, the compiler allocates a space in memory for the variable so that the data can be stored in it at some point during program execution. This block of memory cannot be reused by any other variable, its size cannot be changed, its allocated at compile time by the compiler, and the allocated memory is not “freed” till the end of execution of the program. You can see the memory location allocated by using the ampersand (&) operator as a prefix to the variable.

```
#include<iostream>

using namespace std;

int main() {
    int a;
    cout << &a << endl;
    return 0;
}
```

The allocation of compile-time memory happens on the Stack. The size of the memory is known to the compiler. When a function is called, the memory for the variables gets allotted on the stack and when the function exits, the memory is deallocated.

CSCI 2270 – Data Structures

Recitation 3, Spring 2022

Pointers and Dynamic Memory

What if we do not know the memory required for the program? For example, as seen in previous recitation examples, you would like to pass an arbitrary number of arguments in the command line and have it printed by the program? For this, we can use dynamic memory allocation.

2. Run Time (Dynamic) Memory Allocation

The need for dynamic memory stems from the flexibility of the programs to allocate and deallocate memory as and when required, the uncertainty of not knowing the amount of memory required on program execution and many more. Here, we can tell the compiler to allocate memory to certain variables dynamically. Unlike static memory allocation, dynamic memory allocation has larger memory available to allocate, and there is no guarantee for two subsequent elements of an array to have contiguous memory locations. **Dynamic memory allocated variables are stored in an area of the memory called the heap.**

Consider the following example. We use **`new`** keyword for dynamic memory allocation. As you can see, dynamic memory allocation is always done using pointers. The first pointer **ptr1** holds a dynamically allocated integer, and the second holds a dynamically allocated integer array of size 10.

```
int main()
{
   // Dynamic memory allocation
   int *ptr1 = new int;
   int *ptr2 = new int[10];
   return 0;
}
```

3. Freeing Memory

Unlike compile-time memory allocation, where the program automatically deallocates memory by removing it from the stack on function exit, a dynamically allocated memory variable **MUST BE FREED** by the programmer. This is done by calling the **`delete`** command on the variable. If not, the memory continues to be occupied. Although, for smaller programs coded in this class, a code might not throw an error, repeated execution of the same code in production software can lead to memory leaks and software crashes. The syntax for `delete` is of two types, one for deallocating an array, the other for deallocating a variable:

To free the memory allocated to **ptr2** (array), we call

```
delete [] ptr2;
ptr2 = nullptr;
```

and to free the memory allocated to an integer **ptr1**, we call

```
delete ptr1;
ptr1 = nullptr;
```

Note: Always assign a **nullptr** to a deallocated dynamic memory pointer variable. After **`delete`** is called on a pointer, the address referenced by it is still available for manipulation to the user. To fix this scenario, one has to ensure that the deleted address is not available to anyone for manipulation. Thus, the deleted address is overwritten by assigning a **nullptr** to the dynamic memory variable.

4. Pass-by-value vs Pass-by-pointer vs Pass-by-reference

When we pass a variable into a function, what we are passing is called **actual parameters** and where it is received (i.e., the function), they are called **formal parameters**. They are also called ***actual*** and ***formal arguments***. When a function is called, a chunk of memory, called **activation record,** is allocated. It is the place, where the formal arguments and local variables defined in the memory are held.

In ***pass by value***, a **copy** of the **actual arguments** is stored in the **activation record** as formal arguments. Thus, any manipulation of formal arguments in the function will not be reflected after the function returns/exits. The following code shows this behaviour.

CSCI 2270 – Data Structures

Recitation 3, Spring 2022

Pointers and Dynamic Memory

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

void add2 (int num)
{
    num = num + 2;
    cout << num << endl;
}

int main ()
{
    int a = 10;
    add2(a);
    cout << a << endl;
}
```

Output

```
12
10
```

As you can see, the above function demonstrates pass-by-value. The manipulation of the variable **num** which is a *formal argument* is not reflected in the *actual argument* **a**. Note that this is the default behaviour.

What if we would like to modify the actual argument in a function or a client program?

Here, we use **pass-by-pointers** and **pass-by-reference**.

Pass-by-pointers

Consider the code below. Like **pass-by-value,** in **pass-by-reference** a copy of actual arguments is made in the **activation record** and stored as formal arguments. However, as the copy made is an address, and as we know that we can use pointers to access and manipulate the values in the datatype's address it is pointing to, the changes hold once the function exits/returns.

Notice the code, as we are passing the address of the variable **a**, the argument in the function **add2** must be a pointer.c

CSCI 2270 – Data Structures

Recitation 3, Spring 2022

Pointers and Dynamic Memory

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

void add2 (int * num)
{
    *num = *num + 2;
}

int main ()
{
    int a = 10;
    add2(&a);
    cout << a << endl;
}
```

Output

```
12
```

Pass-by-reference

In pass-by-reference, a copy of the address of the actual parameter is stored in the formal parameter. This is achieved by passing the variable as is into the function but prefixing the function argument with an ampersand. This lets the compiler know that the function call is pass-by-reference.

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

void add2 (int &num)
{
    num = num + 2;
}

int main ()
{
    int a = 10 ;
    add2(a);
    cout << a << endl;
}
```

Output

```
12
```

CSCI 2270 – Data Structures

Recitation 3, Spring 2022

Pointers and Dynamic Memory

Now examine the following code and try to find out why it is persisting the change?

```
#include<iostream>

using namespace std;

void add2 (int a[], int len)
{
    for (int i=0; i<len; i++)
    a[i]+= 2;
}

int main ()
{
    int a[] = { 1 , 2 , 3 };
    add2( a, 3 );
    for ( int i=0;i< 3;i++)
      cout << a[i] << endl ;
}
```

Is the previous one similar/different to the one given below?

```
#include<iostream>

using namespace std;

void add2(int *a, int len)
{
    for (int i=0; i<len; i++)
    a[i]+= 2;
}

int main ()
{
    int a[] = { 1 , 2 , 3 };
    add2(&a[0], 3);
    for (int i=0; i<3; i++)
      cout << a[i] << endl;
}
```

Can you tell, which one is pass-by-reference and which one is pass-by-pointer?

CSCI 2270 – Data Structures

Recitation 3, Spring 2022

Pointers and Dynamic Memory

5. Exercise

Open your exercise file and complete the TODOs in the C++ program which does the following:
1. It will read from a text file. The file contains numbers separated by white spaces. Provide the file name as a command line argument. Count of numbers in the file is not known.
2. Create an array dynamically of a capacity (say 10) and store each number as you read from the file.
3. If you exhaust the array and yet to reach the end of file, dynamically resize/double the array and keep on adding.

Submission: Once you have completed the exercises, zip all the files up and submit on the Canvas link. Name the submission in the following format: recitation3_firstname_lastname.zip