Lecture 5: Memory management in C/C++

1 Introduction to memory management

Today we will talk about how to manage computer memory in your program. Up until now, we have been strictly using stack part of the memory.

1.1 Heap and stack

Let's start with the limits of stack. Inspect the following example

```
1  //declare_large.cxx
2  #include<iostream>
3  #include<stdio.h>
4  #include<vector>
5  #include<fstream>
6  #include<sstream>
7  using namespace std;
8  void check(char c=0){
9          cout<<"Crash point "<<c<<endl;
10 }
11 int main(int argc,char *argv[]){//Main begins
12         check();
13         int a[1000];
14         check('a');
15         int b[1000000];
16         check('b');
17         //int c[10000000];
18         //check('c');
19     return 0;
20 }//Main Ends
```

If you compile and run the code, you will get the following output:

```
Crash point
Crash point a
Crash point b
```

If you now uncomment line 17 and 18 you will instead get the following output.

```
Segmentation fault (core dumped)
```

You ran out of space in stack in your array declaration. Let's calculate how large c is, 10000000 integers =

$$10000000 \times 4\,\mathrm{B} = 40\,000\,000\,\mathrm{B} \sim 38.15\,\mathrm{MB}$$

All of you are using machines with at lest 4 GB and stack size will be smaller as a result. In my case, I have 32 GB of RAM and 10000000 integers mean.

$$\frac{38.15\,\mathrm{MB}}{32\,\mathrm{GB}} = 0.119\,\%$$

of my total RAM. As you can see that my stack is limited to far less than 1 percent of installed memory. You can probably guess why one would want to start using heap for larger data. Stack is perfectly fine to use on small variables but as soon as you anticipate "large enough" data, you need to start using heap.

1.2 Declaring pointers

Declaring variables using heap is almost the same as using stack. However we have to let computer know that we are referring to heap by using special operators when we declare and use variables in heap. More on pointers can be found on http://www.cplusplus.com/doc/tutorial/pointers/
Some of the important operators are:

```
1 - &
```

Address-of operator.

```
1 - *
```

Deference operator.

```
1 - ->
```

Overloadable functions operator (equiavlent to stack .).[1]

Similar to stack variables, you declare variables in the following manner. Do note that you cannot assign values during declaration of pointers.

```
1       int *one ;
2       char *two ;
3       float *three ;
```

When we start using arrays in C, this gets a little bit "funnier".

```
1       int *array1d;
2       int **array2d;
3       array1d=(int*) malloc(100 * sizeof(int));
4       array2d=(int**) malloc(100 * sizeof(int*));
5       for (int i=0; i<100;i++){
6               array2d[i]=(int*) malloc(100 * sizeof(int));
7       }
```

For 1D array, you must declare the size of the array to the pointer. For 2D array, you must declare a double pointer for the array in 1D, then assign singer pointers to the array from double pointer and assign their size.
Also, when you start using the arrays declared in heap, you must make sure that you free the memory segments then delete the pointer.

```
1       int *array1d;
2       int **array2d;
3       array1d=(int*) malloc(100 * sizeof(int));
```

[1] more information on this can be found on https://en.cppreference.com/w/cpp/language/operators

```
        array2d=(int**) malloc(100 * sizeof(int*));
        for (int i=0; i<100;i++){
                array2d[i]=(int*) malloc(100 * sizeof(int));
        }
        free(array1d);
        delete array1d;

        for (int i=0; i<100;i++){
                free(array2d[i]);
        }
        free(array2d);
        delete array2d;
```

If you fail to do so, when the program executes and exits the memory will not be free and still be locked and allocated to a pointer that no program is using anymore. This is one of the cases of memory leak. The memory leak can lead to systems failure as well as corrupt data and many undefined behaviour. You have to be particularly careful in the exit and escape conditions.
Below is a full skeleton for testing pointers.

```
//pointertest.cxx
#include<iostream>
#include<stdio.h>
#include<vector>
#include<fstream>
#include<sstream>
using namespace std;
int main(int argc,char *argv[]){//Main begins
        int *one;
        char *two;
        float *three;

        int *array1d;
        int **array2d;
        array1d=(int*) malloc(100 * sizeof(int));
        array2d=(int**) malloc(100 * sizeof(int*));
        for (int i=0; i<100;i++){
                array2d[i]=(int*) malloc(100 * sizeof(int));
        }

        for (int i=0; i<100; i++){
                array1d[i]=i;
        }

        for (int i=0; i<100; i++){
                printf("Address %p, Value: %i \n", &array1d[i] , array1d[i]
↪  );
                //cout<<&array1d[i]<<endl;
        }

        free(array1d);
        delete array1d;
        for (int i=0; i<100;i++){
                free(array2d[i]);
```

```
34          }
35          free(array2d);
36          delete array2d;
37      return 0;
38  }//Main Ends
```

The output is the following:

```
Address 0x14f5e70, Value: 0
Address 0x14f5e74, Value: 1
Address 0x14f5e78, Value: 2
Address 0x14f5e7c, Value: 3
...
```

Using the above skeleton, we will now practice memory management in C framework.

= The practical programming part of this course will now begin for 60 minutes. =

2 Handling memory in C

1. Declare a 100 element integer array using pointers and malloc in heap.
2. Declare a 10 x 10 integer array using pointers and malloc in heap.
3. Assign both of the above numbers from 0 to 100.
4. Print the memory addresses and their assigned values.
5. Create an escape condition to escape from printing addresses and their values when the value is 11 for the 1D array and for 10 x 10 array.
6. Create a general function that intakes a 1D array and an integer value and looks for the address of the given integer value and its location in array index.
7. Do the same as above fore 10 x 10 array.
8. Look for all of the possible memory leaks and fix them.

= The theoretical lecture part of this course will now continue for 15 minutes. =

3 The C++ way

You may have noticed by now that handling memory in stack and heap is rather different using the C way. You must declare your pointer, assign a chunk of memory, then assign its value. The advantage is obviously the access to much larger portion of the RAM.
As for C++, it's rather easy. This is thanks to the usage of vectors. The vectors have many in-built features such as dynamic memory allocation. As long as you let the computer know that you want to use heap, the vectors will handle everything for you but you have to be careful with the pointers.

```
//pointertestcpp.cxx
#include<iostream>
#include<stdio.h>
#include<vector>
#include<fstream>
#include<sstream>
using namespace std;
int main(int argc,char *argv[]){//Main begins

        vector<int> *vector1d= new vector<int>;
        for (int i=0; i<10; i++){
                vector1d->push_back(i);
        }

        for (auto i = vector1d->begin(); i != vector1d->end(); i++){
                printf("Address %p, Value: %i \n", i , *i );

        }
        free(vector1d);
        delete vector1d;

        return 0;
}//Main Ends
```

The following is the resulting output:

```
Address 0x1037f00, Value: 0
Address 0x1037f04, Value: 1
Address 0x1037f08, Value: 2
Address 0x1037f0c, Value: 3
...
```

There are of course many different variations on how to use the dynamic memory of a vector since you have many options built in. For instance you can also call:

```
vector<int> *vector1d= new vector<int>;
vector<int*> vector1dd;
vector<int*> *vector1ddd;
vector<int*> *vector1dddd = new vector<int*>;
```

Obviously there is no "right" way. You can also do a mixed 1d vector x array.

```
        vector<int*> *vectorcd= new vector<int*>;
        int* array1d=(int*)malloc( sizeof(int) * 10 );
        for (int i=0; i<10; i++){
```

```
4                   for (int j=0; j<10; j++){
5                           array1d[j]=i+j;
6                   }
7                   vectorcd->push_back(array1d);
8           }
```

However you are strongly recommended to use nested vector or single dimensional vector for the purpose of simplicity. For this course, we will only use single dimensional heap vectors.

= The practical programming part of this course will now begin for 60 minutes. =

4 Handling memory in C++

1. Declare a 100 element integer vector in heap.
2. Assign the above, integers from 0 to 100.
3. Print the memory addresses and their assigned values.
4. Create an escape condition to escape from printing addresses and their values when the value is 11 for the above vector.
5. Create a general function that intakes a 1D vector and an integer value and looks for the address of the given integer value and its location in array index.
6. Look for all of the possible memory leaks and fix them.
7. Create a general function that takes in a heap array and prints its average, median, sample standard deviation, skew, maximum and minimum value.
8. Create a general function that takes in a heap vector and prints its average, median, sample standard deviation, skew, maximum and minimum value.
9. Feed a random number generator into the above functions and make sure everything works and look for memory leaks and fix them.

5 Conclusion

You have now learned how to use heap. As complicated as using heap is compared to stack memory, the greatest advantage is the size.
The exact style and method used to use the heap part can be important as for example many GPU can handle simple arrays and array operations but cannot handle vectors as vectors are more complicated.
As long as you are using heap. You can use most of the installed RAM in your computer. Next class we will start declaring very large arrays and vectors.