Universität Zürich[UZH]
Physik-Institut

Lecture 5: Memory management in C/C++

1 Introduction to memory management

Today we will discuss how to manage computer memory in your programs. So far, we have been using only the *stack* portion of memory.

1.1 Heap and stack

Let's begin by examining the limitations of the *stack*. Consider the following example:

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

If you compile and run this code, the output will be as follows:

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

If you now uncomment line 13 and 14, you will instead get the following output.

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

You ran out of *stack* with your `array` declaration. Let's estimate how large the `array c` is:

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

Even though modern machines typically have several gigabytes of RAM (for example, mine has 32 GB), the available *stack* is limited to a much smaller fraction. All of you should be using machines with at least 4 GB of RAM. For example, in my virtual machine with 4 GB of RAM, an `array` of 10000000 integers corresponds to

$$\frac{38.15\,\mathrm{MB}}{4\,\mathrm{GB}} = 0.93\,\%$$

of my total RAM, yet the `array` allocation already exceeds the *stack* limit (by far, actually). This illustrates why large data structures should not be placed on the *stack*. The *stack* is well-suited for small, short-lived variables, but as soon as you anticipate *large enough* data, you should switch to *dynamic allocation* on the *heap*.

1.2 Declaring pointers

So far, we have always declared variables directly *by value*. In C/C++, however, there is also the option to declare *pointer* variables — i.e., variables that store the *address* of another object in memory instead of holding the value directly.
Because different data types occupy different amounts of memory, a pointer must be declared with the type of object it points to. The syntax is similar to that of ordinary variables:

```
int* one;
char* two;
float* three;
```

Unlike ordinary (direct) variables, you cannot assign values to a pointer at declaration in the same way. Instead, a pointer can be initialized with an address (not a raw value), as in the following examples:

```
int x = 5;
int* one = &x; // points to an existing variable
```

Here, `one` is initialized with the address of `x`, a previously declared variable.

```
char* two = "hello"; // points to a string literal
```

In this special case of a `char*` pointer, the program creates a constant string literal *hello* in *static (read-only) memory* and makes `two` point to it.

```
float* three = nullptr; // explicitly initialized to nullptr
```

Here, the pointer is explicitly initialized with `nullptr`, which makes sure it starts out in a well-defined state instead of containing an uninitialized, unpredictable address.
More on pointers can be found on http://www.cplusplus.com/doc/tutorial/pointers/ .

Some important operators related to pointers are:

```
&       // Address-of operator: Produces the memory address of an object.
*       // Dereference (indirection) operator: Accesses the object a pointer points to.
->      // Member access through pointer: Equivalent to writing (*ptr).member.
```

In the context of raw pointers, the main advantage of the `->` operator is primarily syntactic convenience. The following example illustrates how pointers and their corresponding values are connected through the *address-of* (`&`) and *dereference* (`*`) operators:

```
// test_pointer_value.cxx
#include <iostream>
using namespace std;
int main(){
  int x = 42;
  int* p = &x; // p stores the address of x
  cout << "Address of x: " << &x << "   " << "Value of x:           " << x << endl;
  cout << "Value of p:   " << p << "   " << "Value via pointer p : " << *p << endl;
  *p = 99; // change x through the pointer
  cout << "Address of x: " << &x << "   " << "New Value of x:       " << x << endl;
  cout << "Value of p:   " << p << "   " << "Value via pointer p : " << *p << endl;
  return 0;
}
```

And the output is as follows:

```
Address of x: 0x7ffe82827bbc   Value of x:           42
Value of p:   0x7ffe82827bbc   Value via pointer p : 42
Address of x: 0x7ffe82827bbc   New Value of x:       99
Value of p:   0x7ffe82827bbc   Value via pointer p : 99
```

In C++, char* pointers behave a bit differently from pointers to other types when printed with cout. This is because cout treats a char* not simply as an address, but as the beginning of a *null-terminated string*. The following example demonstrates this behaviour, along with some basic pointer arithmetic:

```cpp
// test_char_pointer.cxx
#include <iostream>
using namespace std;
int main(){
  const char* word = "hello";
  cout << "word =            " << word << endl;         // prints 'hello' (whole string)
  cout << "*word =           " << *word << endl;        // prints 'h' (first character)
  cout << "*(word + 1) =     " << *(word + 1) << endl;  // prints 'e' (second character)
  cout << "word + 1 =        " << word + 1 << endl;     // prints 'ello'  (from 2nd to '\0')
  cout << "address of word = " << static_cast<const void*> (word) << endl; // shows address
  return 0;
}
```

The output of the compiled program is:

```
word =            hello
*word =           h
*(word + 1) =     e
word + 1 =        ello
address of word = 0x60924b004004
```

1.3 Declaring variables on the heap

Declaring variables on the *heap* is conceptually similar to declaring them on the *stack*, but there is one key difference: you must explicitly request and free memory from the *heap* using special functions or operators. In C, this is done with malloc and free, while in C++ it is done with new and delete. Because *heap* memory is accessed indirectly, pointers are essential. A pointer variable stores the address of a memory block on the *heap*, while the pointer itself typically resides on the *stack*.

When we use arrays in C, managing memory becomes a little more intricate:

```cpp
int* one = (int*) malloc(sizeof(int));        // space for one int
char* two = (char*) malloc(6 * sizeof(char));  // space for "hello\0"
float* three = (float*) malloc(sizeof(float));  // space for one float
```

When we start using arrays in C, this gets a bit more involved:

```cpp
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 a one-dimensional array, we must allocate enough memory for the number of elements we want, and the pointer holds the address of the first element. For a two-dimensional array, we first declare a *pointer-to-pointer* (*double pointer*) to represent the rows, then allocate each row separately and assign its address to the corresponding element of the double pointer.
Note that in C++ implicit conversion from void* (the type returned by malloc) to another pointer type is not allowed. Therefore, an explicit cast such as (int*) or (int**) is required. In addition, we must explicitly specify the amount of memory to allocate.

When working with `array`s allocated on the *heap*, it is crucial to free every allocated memory block once it is no longer needed.

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

If you forget to release dynamically allocated memory, it remains allocated and cannot be reused by your program for as long as the program is running. This situation is called a *memory leak*. Memory leaks can gradually consume available memory, leading to increased memory usage, performance degradation, or even program failure if the system runs out of available memory. You must therefore be especially careful to release dynamically allocated memory when it is no longer needed and to handle all exit and error conditions appropriately. When a program terminates, modern operating systems normally reclaim the memory and other resources allocated to the process.
Below is a simple skeleton for experimenting with pointers and *heap* allocation:

```
// pointertest.cxx
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
  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++){
    cout << "Address " << &array1d[i] << ", " << "Value: " << array1d[i] << endl;
  }
  free(array1d);
  for (int i = 0; i<100;i++){
    free(array2d[i]);
  }
  free(array2d);
  return 0;
}
```

The output is the following:

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

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

We will now start a 60-minute practical programming session.

2 Practical Session – Part I

Handling memory in C

1. Allocate a 1-dim. `array` of 100 integers on the *heap* using `malloc`.
2. Allocate a 2-dim. `array` of 10×10 integers on the *heap* using `malloc`.
3. Assign values from 0 to 99 to the elements of both the 1-dim. and the 2-dim. `array`s.
4. Write general functions to print the memory addresses of the elements of both the 1-dim. and the 2-dim. `array`s along with the contained values.
5. Implement in both cases an optional escape condition to stop printing when the value of the element is 11.
6. Write a general function that takes a 1-dim. `array` and a target integer value, and prints the memory address(es) of where this value is found, along with its index in the `array`.
7. Implement a similar function that takes a 2-dim. `array` and a target integer value, and prints the memory addresss(es) of where this value is found, along with its row and column indices in the 2-dim. `array`.
8. Identify all potential memory leaks in your code and fix them.

We will now continue with a 15-minute theory session.

3 Memory management in C++

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 a much larger portion of the RAM.

In `C++`, the approach for dealing with primitive datatypes and `array`s is quite similiar, but the functions `malloc` and `free` have been superseeded by `new` and `delete`, which come with a significantly simplified syntax:

```
// pointertest_newdelete.cxx
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
  int* array1d;
  int** array2d;
  array1d = new int[100];
  array2d = new int*[100];
  for (int i = 0; i < 100; i++) {
    array2d[i] = new int[100];
  }
  for (int i = 0; i < 100; i++) {
    array1d[i] = i;
  }
  for (int i = 0; i < 100; i++) {
    cout << "Address " << &array1d[i] << ", " << "Value: " << array1d[i] << endl;
  }
  delete[] array1d;
  for (int i = 0; i < 100; i++) {
    delete[] array2d[i];
  }
  delete[] array2d;
  return 0;
}
```

With `new` and `delete`, type casting is handled automatically and you no longer need to manually calculate the size of the memory block as with `malloc` and `free`. Moreover, the `new`/`delete` mechanism works uniformly for built-in types, derived types, and even user-defined classes, automatically calling *constructors* and *destructors* where appropriate.

However, in `C++` the most straightforward solution is to use `vector`s. They handle dynamic memory allocation internally, so the data they manage is automatically stored on the *heap*, while the `vector` object itself typically resides on the *stack* and is usually much smaller in size. If desired, you can also allocate the `vector` object itself on the *heap* — though this is rarely necessary. In that case, only a pointer to the `vector` resides on the *stack*, as illustrated in the following example:

```
// pointertest_heap_vector.cxx
#include <iostream>
#include <vector>
using namespace std;
int main(int argc, char *argv[]) {
  vector<int>* vector1d = new vector<int>;
  for (int i = 0; i < 10; i++){
    vector1d->push_back(i);
  }
  for (size_t i = 0; i < vector1d->size(); i++){
    cout << "Address " << &(*vector1d)[i] << ", Value: " << (*vector1d)[i] << endl;
  }
  delete vector1d;
  return 0;
}
```

The following is the resulting output:

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

There are several ways to use a `vector` with dynamic memory, since it can be constructed and managed in different ways. For example, you can declare `vector`s like this:

```
vector<int> vector1;
vector<int>* vector1d = new vector<int>;
vector<int*> vector1dd;
vector<int*> *vector1ddd = new vector<int*>;
```

In `C++`, whether it makes sense to create a `vector` as a local object or allocate it dynamically depends primarily on its required lifetime and ownership semantics, not on the number of elements it contains. A `vector` manages its elements in dynamically allocated storage, while the `vector` object itself is typically very small.

For most practical purposes, the simplest approach — letting the `vector` manage its own memory — is sufficient. Nevertheless, pointers can have practical applications, particularly when working with *polymorphic classes and objects* rather than simple types such as integers. n the following exercises, we will therefore also consider `vector`s containing pointers and dynamically allocated `vector`s.

It is also possible to create more complex structures, such as *heap*-allocated `vector`s of `array`s, or other combinations of `vector`s and dynamically allocated objects:

```
int dim1 = 10;
int dim2 = 100;
vector<int*>* vectorcd = new vector<int*> (dim1);
for (int i = 0; i < dim1; i++){
  int* array1d = new int[dim2];
  for (int j = 0; j < dim2; j++){
    array1d[j] = i + j;
  }
  vectorcd->push_back(array1d);
}
for (int i = 0; i < dim1; i++){
  delete[] vectorcd->at(i);
}
delete vectorcd;
vectorcd = nullptr;
```

For the sake of simplicity, it is strongly recommended to avoid such complex constructions unless absolutely necessary. If nested containers are needed, using nested `vector`s is generally the preferable approach.

Note that in modern `C++`, *smart pointers* can significantly simplify memory management by allowing dynamically allocated data to be managed automatically. The allocated memory is automatically freed when the smart pointer goes out of scope:

```
unique_ptr<int[]> smartarray1d = make_unique<int[]>(100);
```

If the memory needs to be freed explicitly beforehand, this can be done using `reset()`:

```
smartarray1d.reset();
```

We will now start a 60-minute practical programming session.

4 Practical Session – Part II

Handling memory in C++ — Extend Your Functions Package

1. Repeat the exercises of today's first Practical Session using the `C++` functions `new`/`delete` instead of `malloc`/`free`, import the functions into your package and test if the behaviour of the dynamically allocated `array`s is unchanged.

2. Repeat the exercises of today's first Practical Session for 1-dim. and 2-dim. dynamically allocated `vector`s, add the functions to your package and test if the behaviour is unchanged wrt. the `array`s.

3. Create a general function that takes an `array` (annd its size) and prints its average, median, sample standard deviation, skew, maximum, and minimum.

4. CCreate a similar function that takes a pointer to a `vector` and prints the same statistical measures. Compare its behaviour with the function operating on a dynamically allocated `array`.

5. Feed a random number generator into the above functions, ensure all outputs are correct, and check for memory leaks, fixing any that appear.

We will now conclude with a 15-minute question session.

5 Conclusion

You have now learned how to use dynamically allocated memory. While working with dynamically allocated memory is more complicated than using local variables, its greatest advantage is that it allows you to work with large amounts of data without placing large objects directly in the *stack*. The way dynamically allocated memory is used can also matter in practice. For example, GPUs are often well suited to processing simple, contiguous `array`s and performing operations on their elements, whereas host-side `C++` containers such as `vector` cannot necessarily be used directly in GPU code.

By allocating memory dynamically, you can work with datasets that are much larger than those that could reasonably be stored on the *stack*. In the next class, we will begin working with very large `array`s and `vector`s.