Programmazione e Calcolo Scientifico

Matteo Cicuttin

Politecnico di Torino

March 26, 2024

More on constructors

In the last class we discussed the **default constructor** for the stack class.

```
#define STACK_SIZE 8

class stack {
  int      data[STACK_SIZE];
  int      top;
public:
           stack();
  void     push(int value);
  int      pop();
  bool     empty();
  bool     full();
};
```

When we declare a new stack with

```
stack s;
```

the default constructor (`stack::stack()`) gets called and `s.top` gets initialized to zero.

Question: what if we want to copy a stack?

The copy constructor

The **copy constructor** is invoked when you copy an object.

```
#define STACK_SIZE 8

class stack {
  int   data[STACK_SIZE];
  int   top;
public:
        stack();
        stack(const stack& other);
  void  push(int value);
  int   pop();
  bool  empty();
  bool  full();
};
```

For example:

```
stack s1;
s1.pop(42);
s1.pop(70);
stack s2 = s1; // copy constructor called
stack s3(s1);  // copy constructor called
```

- The compiler generates a **default copy constuctors** which **copies** your objects **byte-by-byte**
- The copy constructor needs to be redefined only if that is not what you want (typically when you manage some dynamic memory)

User defined constructors

Let's imagine you want to build a class to model the elements of $\mathbb{Q}$.

```
class rational {
  int   numerator;
  int   denominator;
public:
        rational();
        rational(int num, int den);
};

rational::rational() {
  numerator = 0;
  denominator = 1;
}
```

```
rational::rational(int num, int den) {
  numerator = num;
  denominator = den;
}

rational r1; // default constructor called
             // and r1 initialized to 0

rational r2(7,3); // user defined constr.
                  // called, r2 = 7/3
```

Destructor

The destructor gets called just before an object goes out of scope.

```
struct object {
  object();
  ~object(); // destructor
};
```

- The destructor is a method with the same name of the class and prefixed with a tilde
- You usually need to define it when you manage dynamical resources and you need to release them

```
int f(double x) {
  object o1;
  while (x > 10) {
      object o2;
      /* stuff */
      // o2 destroyed here
  }
  /* stuff */
  // o1 destroyed here
}
```

Exercise to do at home

In order to understand when constructors and destructors get called, I suggest the following exercise:

```
struct object {
  object();
  object(const object&);
  object(int, int);
  ~object();
};

object::object() {
  cout << "default constructor\n";
}

object::object(const object &) {
  cout << "copy constructor\n";
}

object::object(int, int) {
  cout << "user defined constructor\n";
}
object::~object() {
  cout << "destructor\n";
}
```

- Create a program with different functions/blocks
- Create some objects of type `object`
- Observe what gets printed

The Standard Template Library

The C++ standard library is called STL (Standard Template Library). It provides you

- Data structures (`<vector>`s, `<list>`s, `<set>`s, associative arrays, ...)
- Algorithms (sort, search, ...) in `<algorithm>`
- Input/output (we already discussed streams)
- Thread and concurrency support (`<thread>`, `<atomic>`, `<mutex>`, ...)

Take a look to `https://en.cppreference.com/w/cpp`.

Dynamic vectors

The first data structure we will study is the dynamic vector.

- `std::vector<T>` with `T` the type of the contained elements
- Declared in `<vector>`, so remember to `#include <vector>`
- In C++ parlance, `std::vector` is a **container**
- Take a look to `https://en.cppreference.com/w/cpp/container/vector`

```
std::vector<int>          int_v;      // empty vector of ints
std::vector<double>       dbl_v;      // empty vector of doubles
std::vector<meteo_data>   md_v;       // empty vector of meteo_data
std::vector<float>        flt_v(50); // vector of 50 floats
```

Operations on vectors

Let `vec` be a `std::vector<T>` for some `T`.

- `resize()` change the size of the vector
- `operator[]` allows to access elements
- `at()` bound-checked element access

```
int size = 10;
std::vector<int> vec; // vec is empty
vec.resize(size);     // vec is resized to 10 elements

for (size_t i = 0; i < size; i++)
  vec[i] = i+10; // initialize elements

std::cout << vec[5] << "\n"; // OK, print 6th element
std::cout << vec[50] << "\n"; // error: program crashes unpredictably
std::cout << vec.at(50) << "\n"; // error: program crashes predictably
```

More operations on vectors

Let `vec` be a `std::vector<T>` for some `T`.

- `size()` get the current **size** of the vector
- `push_back()` add an element to the end of the vector
- `reserve()` reserve space for a certain number of elements (elements are not initialized), so the size of the vector does not change
- `capacity()` get the current **capacity** of the vector

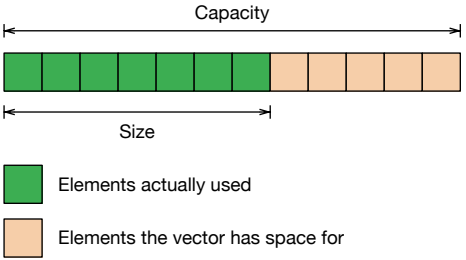
Size and capacity are two **very different** things:

- `size()`/`resize()` are related to the **actual** elements contained in the vector
- `capacity()`/`reserve()` are related to the space the vector has available to store elements
- `size()` $\leq$ `capacity()`
- `resize()`, `reserve()` and `push_back()` are potentially **expensive** operations

Here is where we must study and understand dynamic memory allocation.

Size vs. capacity

When you ask for a vector of `N` elements, `std::vector` could not ask to the system the exact quantity of memory needed, but a little more. `N` is the size, "`N` + little more" is the capacity.



- This **does not mean** that you are allowed to access elements outside $[0, \mathtt{N})$.

The `resize()` operation

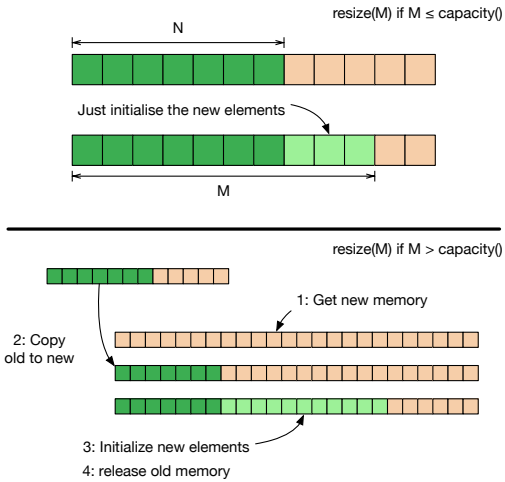
The `resize()` operates as follows:

- If there is sufficient capacity, the `resize()` operation just initializes the additional elements (needs to call default constructor)
- Otherwise, it must allocate new memory, move the existing elements, initialize the new ones and release old memory

Complexity:

$$\begin{cases} \max(0, \mathcal{O}(M-N)) & \text{if } M \leq \texttt{capacity()} \\ \mathcal{O}(M-N)) + \text{reallocation} & \text{otherwise} \end{cases}$$

With `push_back()` the reasoning is similar.



Dynamic memory

C++ provides us with facilities to request memory to the system and release it.

- Basic memory management in C++ is completely manual
- If you ask for memory, you have to remember to release it when you're done
- You must also be sure to not release the same memory twice...
- And you have to pay attention to not lose the pointers to your dynamic memory...

Memory management in languages like C and C++ is one of the most dangerous things. That's the reason why many people is moving to Rust.

In modern C++, if you end up doing manual memory management most probably you are doing the wrong thing. Use the data strures provided by the STL.

So why study it? Because otherwise impossible to fully understand what is going on under the hood.

Memory allocation

Dynamic memory is allocated using `new` and `new[]`.

```
int* is = new int[50]; // allocate 50 integers
meteo_data* md = new meteo_data; // allocate a single meteo_data
meteo_data* mds = new meteo_data[20]; // allocate 20 meteo_data
```

Each pointer points to the beginning of the newly allocated memory.

- Until now we used pointers only to point to single elements of a certain type
- How to access to the 8th `meteo_data` or the 36th `int` above?

Memory allocation - II

On pointers you can use `[]`. Therefore, `mds[7]` is the 8th `meteo_data` and `is[35]` 36th `int`.

The job of `x[n]` with `x` pointer of type `T`, is to take the address in `x`, to compute `x + n*sizeof(T)` and dereference the result.

```
is[5] = 42;        // write to the 6th element
int y = is[8];     // get the 9th element
```

Note that pointers remain just pointers: they do not carry any information about allocation size. It is up to you to remember it.

The block of memory you get from `new` is just a block of memory with no additional structure. No `resize()`, no `push_back()`, nothing. Much better to use `std::vector`.

Note: some people tells you that you shouldn't use `std::vector` because it is slow, and you have to allocate/deallocate arrays exclusively using `new`/`delete`. Those people do not have a clue about C++, so don't listen to them. Think with your own brain.

Memory release

Dynamic memory is released using `delete` and `delete[]`.

```
int* is = new int[50]; // allocate 50 integers
meteo_data* md = new meteo_data; // allocate a single meteo_data
meteo_data* mds = new meteo_data[20]; // allocate 20 meteo_data

/* your code... */

delete [] is;
delete md;
delete [] mds;
```

Notice the use of `[]`. The version `delete[]` is used to release array-like stuff, whereas `delete` is used to release element-like stuff. Using `delete[]` to release element-like stuff and `delete` to release array-like stuff is undefined behaviour.

- `https://en.cppreference.com/w/cpp/memory/new`
- `https://en.cppreference.com/w/cpp/language/delete`

Dangers of manual memory management

```
class int_vector {
  int* data;
  int  size;
public:
  int_vector();
  int_vector(int sz);
  void resize(int);
  ~int_vector();
}

int_vector::int_vector() {
  data = nullptr; size = 0;
}

int_vector::int_vector(int sz) {
  data = new int[sz]; size = sz;
}
```

```
int_vector::resize(int sz) {
  if (data) {
    delete [] data;
    data = nullptr; size = 0;
  }
  data = new int[sz]; size = sz;
}

int_vector::~int_vector(int sz) {
  delete [] data;
  data = nullptr; size = 0;
}

int_vector v1;
v1.resize(10);
int_vector v2 = v1; // boom
// v1 and v2 point to the same memory
```

Suggestions

For the next class, try to digest this stuff. In particular:

- Understand how constructors/destructors work
- Take some time to understand `new`/`delete` and their relation with pointers.

In the next class we will need these ingredients to discuss a fully dynamic data structure.