6. Templates and complexity

Ján Dugáček

September 12, 2017

Table of Contents

Why to use templates?

```
struct pointerGuard {
        int* pointer_;
        pointerGuard(int* pointer) : pointer_(pointer) { }
        ~pointerGuard() {
                delete pointer_;
        }
};
```

- The code above is not particularly useful, because it has to be defined for each type it might contain
- Any pointer can be converted to `void*`, but the `delete` needs to know the exact type so that it would call proper destructors (`free()` would work, but it would not call a destructor)

Main usage

```
template <typename T>
struct pointerGuard {
        T pointer_;
        pointerGuard(T pointer) : pointer_(pointer) { }
        ~pointerGuard() { delete pointer_; }
};
void doStuff {
        pointerGuard<int*> guard(new int(6));
        pointerGuard<double*> guard2(new double(3.141526));
}
```

- `pointerGuard` is a template class, each instance is a different structure with different functions as methods, created at compile time
- There is no performance impact
- If the template argument is not a pointer, the destructor will fail to compile

Other usage

- Although the template arguments can be of the `typename` type, there are more specific types of template arguments, like `int` that requires it to be an integer (arithmetic can be done on it at compile time) or `class` that requires it to be a class

```
template <typename T, int size>
struct array {
        T contents[size];
        T& operator[] (int at) {
                if (at>=0 && at<size) return contents[at];
                return contents[0];
        }
};
array<int, 6> arr;
```

- In this example, `size` is not a variable, but a constant used at compile time

Other usage #2

- Templates produce long and hard to comprehend error messages when used wrongly
- Templates can be used to do much more complex things, to the extent that any algorithm can be programmed with templates to be executed at compile time
- This leads to all kinds of weirdness, i.e. the following erroneous is a broken loop at compile time:

```
template<int n>
struct fibonacci
{
    static constexpr int value
        = fibonacci<n-1>::value + fibonacci<n-2>::value;
};
// Constants for 1st and 2nd values are not given
// ...
int result = fibonacci<40>::value;
```

Exercises

1. Write a `sharedPointer` class that holds a pointer of any type, keeps track of its copies and deletes the pointer when the last copy is destroyed
2. Write a `expandableArray` class that can dynamically reallocates an array of objects of given type if it is filled, needs `operator[]` and an `append()` method

```
struct something {
        something(const something& a) { /*copying*/}
        void operator=(const something& a) { /*assignment*/}
```

Advanced:

1. Write class `deletablePointer` that holds a pointer of any type, keeps track of its copies and all its copies' target is set to `nullptr` when the target is deleted

Linked list

```
struct section {
        section* previous;
        section* next;
        int value;
};
struct linkedList {
        section* first;
        section* last;
};
```

- If we insert an element into the middle of an array, a large part of the array is copied
- If we save the array in a *linked list*, where each element is stored elsewhere but contains pointers to next and previous elements, this operation's duration does not depend on its size

Big O notation

- Inserting into the middle of an array of size n may require up to $c \cdot n$ operations (where the whole array is copied), where c is some constant; we mark this $\mathcal{O}(n)$ and call it *linear complexity*
- Accessing an element in an array of size n requires c operations, where c is some other constant; we mark this $\mathcal{O}(1)$ and call it *constant complexity*
- Inserting into the middle of a *linked list* requires c operations, where c is some constant (different than the previous one); this is again $\mathcal{O}(1)$
- Accessing n-th element of a *linked list* requires $c \cdot n$ operations (it needs to run through all previous ones to find it), where c is some constant; this is again $\mathcal{O}(n)$
- The constants are clearly different, *linked list* is clearly slower, but if the amount of data is huge, the one with better complexity is better (so if we need a lot of insertions, we *have to* use the *linked list*)

Big O notation #2

- An algorithm that sorts an array of size n by running n times through it and swapping couples of elements in the wrong order (called *bubblesort*) needs $c \cdot n^2$ operations to sort the array, so its complexity is $\mathcal{O}(n^2)$, we call it *quadratic complexity*
- An algorithm that finds a number in a sorted array of n numbers can do it by checking if the middle element is greater and lesser than the number sought, focus on the half where the element is, compares it with the middle of this half, selects the quarter where it is and so on does $c \cdot \ln(n)$ operations to find it, so its complexity is $\mathcal{O}(\ln(n))$, we call it *logarithmic complexity*
- An algorithm that learns the prime divisors of number of size n bits by trying to divide it by all numbers up to $\sqrt{2^n}$ needs (if unlucky) $c_1 \cdot 2^{\sqrt{c_2 \cdot n}}$ operations to find it, so its complexity is $\mathcal{O}(e^n)$ and we call it *exponential complexity*

Big O notation #3

- When using the *big O notation*, we care only about how fast the value increases when n is sufficiently large, regardless of constants or slower increasing parts
- The time needed by any two algorithms' marked as $\mathcal{O}(n)$ to work through data of size n differs by a non-zero, non-infinity factor if n is large enough
- Therefore $\mathcal{O}(n) = \mathcal{O}(n + 10000)$ (the constant does not matter if n is large enough), $\mathcal{O}(n) = \mathcal{O}(10000n)$ ($\lim\limits_{n\to\infty} \dfrac{10000n}{n} = constant$, but we would usually prefer the first algorithm), $\mathcal{O}(n) = \mathcal{O}(n + \ln n)$ (if n is large enough, the linear part is too small because $\lim\limits_{n\to\infty} \dfrac{n + \ln n}{n} = 1$)
- If the time depends on more values, for example n and m, both are in the notation, for example $\mathcal{O}(m + \ln n)$ and its writing is not reduced (in this case, n may be way larger than m)
- In reality, sizes can be small, so time $2n + 5$ may be better than time 200, although first is $\mathcal{O}(n)$ and the other is $\mathcal{O}(1)$

Exercises

1. Find a way to sort an array with complexity only $\mathcal{O}(n \cdot \log n)$ (just find a way, writing the code would take long)
2. Find a way to store data in a way that accessing a given element would have complexity $\mathcal{O}(\log n)$ and adding an element would also have $\mathcal{O}(\log n)$ complexity (just think of a way)

Advanced:

1. Find a way to sort an array of elements numbered from 0 to n (not necessarily all are present) with complexity $\mathcal{O}(n)$
2. Prove that the complexity of an operations that appends at the end of an array, reallocating it to a larger one (copying all previous elements) when it's full, is $\mathcal{O}(1)$ in an average case (it's called *ammortised complexity*, one addition may need $c \cdot n$ operations, but if we average enough of them, it's different)

std::vector

- std::vector is a class that maintains an expanding array
- Its elements are accessed with operator [] and placed at the end of the array with its push_back() method
- **Do not get pointers to its elements**, the array may be reallocated, rendering all pointers invalid

```
std::vector<int> vec;
for (int i = 1; i < 10; i++)
        vec.push_back(42 / i);
vec[3] = 0;
unsigned int size = vec.size();
```

- You need to include vector to use this
- This data structure is called *vector* in computer science

std::list

- Data is stored in a *linked list*, allows adding or removing elements anywhere in constant time, but accessing an element at given index needs linear time
- Elements are accessed using an `iterator`, which holds information about the element
- You can access iterator's elemens using the dereference operator

```
std::list<int> list;
for (int i = 1; i < 10; i++)
    list.push_back(42 / i);
for (std::list<int>::iterator it = list.begin();
        it != list.end(); )
    if (*it == 3) it = list.erase(it);
    else ++it;
list.insert(list.begin(), 3);
```

- You need to include `list` to use this

std::map

- Data is stored in a *red-black tree*, allows accessing elements in logarithmic time, adding or removing elements anywhere also in logarithmic time
- Elements and indexes can be read using an `iterator` (they are sorted by their indexes), but the main usage is accessing via `operator []` (can be used to construct new elements as well)
- The index does not have to be an integer, it can be anything that can be compared

```
std::map<std::string, int> array;
array["hi"] = 3;
array["zaphod"] = 4;
array["ford"] = array["zaphod"] - 1;
for (std::map<std::string, int>::iterator it = array.begin();
        it != array.end(); ++it)
    std::cout << it->first << "="<< it->second << std::endl;
```

- You need to include `map` to use this
- `std::multimap` can contain more elements with the same index, but access is not so simple

std::unordered_map

- Data is stored in a *hash table*, allows accessing, adding and removing elements in constant time
- Elements are accessed and created via `operator []`, elements and indexes can be read using an `iterator` (they are not stored in any specific order)
- The indexes must be convertible to integers with *as little relation* to them as possible using a `std::hash<theType>` function, so it's a bit trickier to get working with custom types
- This is called *dictionary* in many other programming languages

```
std::unordered_map<std::string, int> hashtable;
hashtable["Breivik"] = 70;
hashtable["Osama"] = 3000 + hashtable["Breivik"];
hashtable["Stalin"] = hashtable["Osama"] * 6666;
if (hashtable.find("Dugi") == hashtable.end())
        std::cout << "Dugi is not a murderer!" << std::endl;
```

- You need to include `unordered_map` to use this
- `std::unordered_multimap` can contain more elements with the same index, but access is not so simple

Exercises

1. Rewrite an existing exercise that uses an expanding array to use `std::vector`
2. Write a program that sorts a certain number of randomly generated elements using bubblesort (array is passed *n* times, always swapping adjacent out-of-order elements) and using std::map to sort them, compare the time it takes

```
#include <ctime>
// ...
time_t before = time(0);
for (int i = 0; i < 10000; i++) {}
std::cout << "Took " << (time(0) - before) << std::endl;
```

Advanced:

1. Write a `filter()` function that reads through a `std::list`, erasing all elements lesser than a number given as another argument
2. Study the order in which are numbers stored in a `std::unordered_map` indexed by numbers, is it a good way to order them?

A bit of physics

- The computer's memory is large enough for many purposes (4 GiB means a milliard of integers), but its distance from the processor needs a bit of thought
- The distance between CPU and RAM is roughly 10 cm, so the time needed to travel there is roughly at least 3.10^{-8}s (assuming the speed of light, which is *not* the case)
- If the processor is running at 2 GHz, its cycle takes 2.10^{-9}s (and it does in average 7 operations per cycle!)
- In practice, it can be assumed that it takes as much as time as 150 cycles

Cache

- To deal with this issue, the processor has its own small memory called *cache* that contains all recently accessed data along with their surroundings
- Cache has usually several levels, each larger and with greater access time, the smallest is about 64 KiB, the largest 128 MiB
- Because this is still imperfect, most processors have a feature called *hyper-threading*, a CPU-like structure that does other stuff while it is waiting for data to arrive into cache

Avoiding cache misses

- Even with hyper-threading, the processor would mostly wait for data rather than do work if the program had a cache miss every dozen operations
- There is no way to tell the CPU to cache some area and in most cases, it would not be known ahead anyway
- The program should therefore have the data it needs at similar time stored at similar location
- Automatic local variables are stored close to each other and some of them are always accessed (current function's arguments), so they are fast
- Accessing a dynamically stored array may cause a cache miss, but once it's accessed, all the elements around the accessed element are cached and can be quickly accessed
- Dynamically allocated pieces of data spread on many locations (depending on what was free at allocation time) often incur a cache miss per access
- Locations accessed through many consecutive pointer jumps can cause several cache misses per access
- The main benefit of using `char` and `short int` instead of `int` is that they take less space and more of them can be cached simultaneously

Data structures and cache misses

- A regular array is the fastest
- `std::vector` is the best solution for most cases, because its contents are coherent (and thus can be the best even if you need to insert into its middle using an iterator)
- If `std::vector` is not good, usually the best solution is `std::unordered_map` because it can access similarly to vector after calculating its address
- `std::list` often causes a cache miss per iteration, so it's rather slow and should be used only in cases where the lower complexities are absolutely necessary due to the sizes of arrays ($\mathcal{O}(n)$ vs. $\mathcal{O}(200)$)
- `std::map` often causes several cache misses per each element access and iteration, so it should be used only if absolutely necessary
- Note that objects dynamically allocated at similar time (for example one after another) will usually be given addresses one after another, so cache misses will not always happen as much as expected
- More info here: `http://ithare.com/c-performance-common-wisdoms-and-common-wisdoms/`

Homework

- Write a class that works like an XML tag, contains several named attributes, other tags or text, with methods to add new attributes and tags inside, access their contents and remove them and also has a method to write the tag along with the tags and attributes it contains into a document; use efficient STL containers
- Challenge: Give it also a constructor that can be used to parse an XML document
- About XML: `https://en.wikipedia.org/wiki/XML`

Advanced homework:

- Same as the regular one, but give it also a constructor that can be used to parse an XML document and deal with the few characters that need to be escaped