DD2387 Programsystemkonstruktion med C++
Lab 1: The Progressive

21st of September 2015

Introduction

By finishing the assignments present in this document, students will effectively gather extra credit that will affect the student's grade upon finishing the course.

General Requirements

- The *General Requirements* listed in *Lab 1: The Essentials*.
- During the oral presentation of each assignment, the questions associated with such shall be answered verbally.

Submitting and presenting your work

Every assignment requires an oral presentation where you, and your potential partner, shall be able to explain and answer questions regarding your solutions.

Some assignments require you to submit code for automatic testing, which will verify that your implementation is correct in relation to the requirements set forth by the assignment in question.

To submit an implementation for automatic testing, open up a web browser and point it to `https://kth.kattis.com`, then;

- authenticate using your KTH-id, if this is the first time you are using *Kattis* you must register for the service after signing in, also;
- make sure that you register as a student taking cprog15, before you try to submit any of your solutions.

Contents

1 The Progressive (assignments for extra credit) 3

1.1 The Matrix (10p, krav för betyg C) . . . 3

1.1.1 Requirements . . . 3

1.1.2 Questions . . . 3

1.1.3 Hints . . . 4

1.2 *Reserved* . . . 4

1.3 Concurrency is The New Black (5p) . . . 5

1.3.1 Requirements . . . 5

1.3.2 Questions . . . 5

1.3.3 Hints . . . 5

1.4 The Hypercube (4p) . . . 6

1.5 Space Is Not Infinite (10p, krav för betyg A) . . . 6

1.6 *Reserved* . . . 7

1.7 The Master of Life & Death (4p) . . . 8

1.7.1 Hints . . . 8

1.7.2 Requirements . . . 9

1.8 Better Safe Than Sorry (3p krav för betyg A) . . . 10

1.8.1 Requirements . . . 11

1.8.2 Questions . . . 11

1.8.3 Hints . . . 11

1 The Progressive (assignments for extra credit)

1.1 The Matrix (10p, krav för betyg C)

4-GET Fuel is a company that focus on fuel efficiency for cutting edge robots similar to those in *The Matrix (1999)* — if the robots can run forever without using humans as power, humanity shall forever be safe.

Given the companies name it might not come as a surprise that they have misplaced the source code for a bunch of different `Matrix` implementations. All they know is that there are `8` implementations that have bugs, and `1` (one) that works as intended (see wikipedia on matrices[1]).

Your task is to find the correct implementation among the object files available in the assignment directory[2].

Note: *The* object files *are compiled using* `g++` *on* `u-shell.csc.kth.se`*, and as such they are bound to the architecture used on the host in question.*[3]

Note: *Implement as much of* `class Matrix` *as you deem necessary in order to locate the implementation faults in the previously mentioned* object files.

1.1.1 Requirements

- Identify the individual problems associated with the faulty *object files*, as well as proving that the working *object file* is correct.
- Notes regarding the different *object files*, as well as the *unit tests* used during this assignment, shall be available during the oral presentation.
- You shall use a unit testing framework—such as *cxxtest*—in order to implement *unit tests* that checks for correct—and with that errornous—behavior.

  See the assignment directory for a skeleton implemention.

1.1.2 Questions

- Are there any design decisions that could be improved in regards of `matrix.hpp`?
- Which object file contains a valid implementation?
- What are the problems associated with the faulty object files?
- Some of the matrix methods are similar. One way to avoid code duplication would be to implement one operator in terms of another. What are the *pros* and *cons* doing so for the matrix class? Is there a better functional way to avoid code duplication?

[1] `https://en.wikipedia.org/wiki/Matrix_(mathematics)`
[2] `http://www.csc.kth.se/utbildning/kth/kurser/DD2387/cprog15/labdir/lab1/1.1_extra/`
[3] This means that they, as an example, cannot be used to build an application targetting `Windows`.

1.1.3 Hints

- One of the implementations is logically correct but leaks memory; `valgrind` is a great resource for these kinds of problems.
- Each faulty *object file* contains at most `1` (one) problem.

1.2 *Reserved*

Intentionally left blank.

1.3 Concurrency is The New Black (5p)

Your task is to implement a new container template named `template<typename T> class SafeVector` that, in addition to the functionality of `Vector<T>`, provides a member-function, `SafeVector<T>::safeswap`, that swaps the elements at the corresponding indices.

```
template<class T>
void
SafeVector<T>::safeswap (std::size_t index1, std::size_t index2)
{
  // TODO: implement
}
```

The added member-function shall be thread-safe, which means that it shall be possible to invoke the member-function on the same object from many separate threads, while still obtaining a result *as if* the member-function was called sequentially (in some undefined order).

1.3.1 Requirements

- Implement two different solutions, one that locks the entire container, and another that simply locks the two elements being swapped.
- Test the two implementations, and make sure both of them work as they shall.
- Make sure that your implementation avoids potential deadlocks.
- No other member-functions needs to be thread-safe.
- The implementation should be submitted to, and approved, by *Kattis*.

1.3.2 Questions

- What are the pros and cons associated with the two different implementations?
- What is a "*deadlock*"?
- `SafeVector<T>` certainly adds some functionality compared to `Vector<T>`, but are there any side-effects associated with this added functionality?

  Make sure you can answer questions such as:

  - Can a developer safely replace the usage of `Vector<T>` with `SafeVector<T>`, without running into issues where the latter is not usable in a context that allowed the former?
  - Is there any performance penalty associated with `SafeVector<T>` compared to `Vector<T>`, even if `SafeVector<T>::safeswap` is never called?

1.3.3 Hints

- Look at `http://cppreference.com`, and read about the utilities available in `<thread>`, and `<mutex>`.

- In order to compile multi-threaded implementations with `gcc` and/or `clang`, make sure that you use `-pthread` as an argument to the compiler.
- Make sure you understand the implications of using something such as `std::mutex`; if such is a data-member, how does it affect the class?

1.4 The Hypercube (4p)

Använd mallar för att implementera en klass `Hypercube` som hanterar liksidiga matriser med godtycklig dimension. Ta hjälp av `Matrix` eller `Vector` för implementationen. Exempel:

```
Hypercube<3> n(7);    // kub med 7*7*7 element
Hypercube<6> m(5);    // sex dimensioner, 5*5*...*5 element
m[1][3][2][1][4][0] = 7;

Hypercube<3> t(5);
t = m[1][3][2];       // tilldela med del av m
t[1][4][0] = 2;       // ändra t, ändra inte m
std::cout << m[1][3][2][1][4][0] << std::endl;  // 7
std::cout << t[1][4][0] << std::endl;           // 2
```

När du har löst uppgiften, tänk efter hur du kan göra en elegant lösning på 10-15 rader. Redovisa helst en elegant lösning men en elegant tankegång kan också godkännas.

1.5 Space Is Not Infinite (10p, krav för betyg A)

Implementera en specialisering `Vector<bool>` som använder så lite minne som möjligt, dvs representerar en `bool` med en bit. Använd någon stor heltalstyp (såsom `unsigned int`) för att spara bitarna. Observera att du ska kunna spara ett godtyckligt antal bitar. Skapa funktionalitet så att vektorn kan konverteras till och ifrån ett heltal (i mån av plats). Implementera all funktionalitet från `Vector` som t.ex. `size`. Du behöver inte implementera `insert` och `erase` om du inte vill.

Skapa en iteratorklass till `Vector<bool>` som uppfyller kraven för en random-access-iterator (dvs har pekarliknande beteende). Ärv från `std::iterator<...>` (genom `#include <iterator>`) för att lättare få rätt typdefinitioner. Låt din `iterator` ärva från din `const_iterator` eftersom den förra ska kunna konverteras till den senare. Läs på om iterator_traits<T> och definera de typdefinitioner du behöver (kanske `typedef bool value_type`). Du behöver inte implementera `reverse_iterator` eller `const_reverse_iterator` om du inte vill. Exempel som ska fungera:

```
Vector<bool> v(31);    // Skapa en 31 stor vektor
v[3] = true;
Vector<bool> w;        // tom vektor
std::copy(v.begin(), v.end(), std::back_inserter(w));
std::cout << std::distance(v.begin(), v.end());
// konstant iterator och konvertering
Vector<bool>::const_iterator it = v.begin();
std::advance(it, 2);
```

Det kan vara mycket svårt att få till så att `sort(v.begin(), v.end())` fungerar.

1.6 *Reserved*

Intentionally left blank.

1.7 The Master of Life & Death (4p)

A container can be described as an entity which is responsible for life time management of objects stored inside the container; $[begin(), end())$.

This means that every element in the user accessible range should be properly constructed, and that there should be no constructed object (owned by the container) that is not part of this range.

If an element is erased from a container, the corresponding object's destructor should be called, likewise; if a reallocation of the underlying storage is needed, a container should not construct any more objects than there are elements present in the range. The elements shall still be in consecutive order.

Your task is to modify your previous implementation of `Vector<T>`, effectively making it follow the semantics described in the previous paragraphs.

1.7.1 Hints

```
struct T1 {
   T1 ()          { ++object_count; }
   T1 (T1 const&) { ++object_count; }
  ~T1 ()          { --object_count; }

  static unsigned int object_count;
};

unsigned int T1::object_count = 0;

int main () {
  {
    Vector<T1> v1 (3);   assert (T1::object_count == 3 && v1.capacity () >= 3);
    Vector<T1> v2;       assert (T1::object_count == 3);

    v1.push_back (T1{}); assert (T1::object_count == 4 && v1.capacity () >= 4);
    v2 = v1;             assert (T1::object_count == 8);

    v2.erase (1);        assert (T1::object_count == 7);
    v2.erase (1);        assert (T1::object_count == 6);
  }

  assert (T1::object_count == 0);
}
```

1.7.2 Requirements

- Every requirement specified in *The Template Vector* still applies, more specifically; you are not allowed to reallocate the underlying storage every time an insertion/erase takes place[4].
- Your implementation should be submitted to, and approved by, *Kattis*.

[4]The time complexity of insertion through `push_back`, and the equivalent `insert`, should still be *amortized constant.*

1.8 Better Safe Than Sorry (3p krav för betyg A)

No requirement of *exception safety* was stated in the problem description of `Vector<T>`, which inherently makes the container inappropriate to use in a production environment.

In this assignment you should modify your previous implementation of `Vector<T>` to make it meet the requirements of having *Strong Exception Safety*.

The Fundamental Problem

Imagine an implementation where a function does some calculations, where the calculations include a callback supplied by the callee:

```
void f (std::function<int()> g) {
  int * p1 = new int[1024];
  int * p2 = new int[2048]; // (A)
  int    x = g ();          // (B)

  // ...

  delete [] p1;
  delete [] p2;
}
```

The implementation looks innocent enough. Any acquired resources are released before leaving the function.. but, what if `(A)` or `(B)` throws an exception?

The function is, per definition, not *exception safe*.

The Levels of Safety

There are several different levels of exception safety, ranging from unsafe to "*super safe*", where the higher levels includes the guarantees made by any of its previous entities.

- **No Exception Safety**

  There are no guarantees if an exception is thrown in the implementation, implementation acquired resources might leak, there might be side-effects such as writes to callee accessible data, or the application might crash.

  The behavior is undefined.

- **Basic Exception Safety**

  Even though there might be callee observable side-effects, any resources acquired by the implementation shall not leak in case of an exception.

  This level is also known as the "*no-leak guarantee*".

- **Strong Exception Safety**

  The callee observable state is guaranteed to remain the same as prior to invoking the implementation, even in cases where an exception occurs.

- **The No-throw Guarantee**

  The implementation should never throw an exception.

1.8.1 Requirements

- The implementation should not leak any acquired resources, even if an exception is thrown.
- The callee observable effects in case of an exception should be as if the operation was never invoked.
- The implementation should be submitted to, and approved, by *Kattis*.

1.8.2 Questions

- Is it always desirable to aim for *Strong Exception Safety*?
- What are the implications of using a *Strong Exception Safe* version of `Vector<T>`?
  - Does it affect performance?
  - Can you think of any other implications compared to your previous implemetation?

1.8.3 Hints

- Remember that exceptions may occur during operations on the objects within the container, such as when copying/moving elements; you are to handle such cases appropriately.
- Remember that *move-constructors*, by design, modify the moved-from object. If such a constructor throws an exception, there is no way to "*rollback*" to the previous state.