DD2387 Programsystemkonstruktion med C++

Lab 1: The Essentials

25th of August 2014

Introduction

The purpose of this lab is to acknowledge that you know about the elementary concepts of C++. This includes, but is not limited to; the usage of classes, iteration statements (loops), variables, memory management, and templates, as well as; building, debugging, and running your program.

You are allowed to solve the assignments either by yourself, or with at most one (1) partner, where the latter is strongly advised. Do note that the oral presentation associated with this lab mandates that every member of your team is able to answer questions regarding every aspect of your implementation.

This means that even though you are allowed to divide the work load, knowledge of the entire implementation must be shared equally between every member of your team.

Please read through the entire contents of this document, prior to solving any of the problems listed, since most assignments are connected in one way or another.

General Requirements

- Your code should be modularized in classes, and files.
- Make sure that your implementations are easy to read, maintain, and understand. This includes the usage of correct indentation, as well as having suitable names for your variables.
- Your implementations shall not leak memory, or any other acquired resource.

  Do mind the purpose of constructors, and their relationship with the corresponding destructor: *Resource Acquisition Is Initialization*[1].
- Your implementations shall demonstrate that you fully understand the semantics associated with the keyword `const`.
- Throughout this document, you are not allowed to use the containers available in the Standard Library to solve the assignments listed[2], unless this has been explicitly stated to be allowed for some assignment.
- Those questions that require a written answer shall be answered in an elaborate manner; single worded answers will not be accepted. It is mandatory that each answer consists of *at least* one (1) full sentence.

[1] `http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization`

[2] As an example, you are not allowed to use `std:vector` when writing your own vector implementation.

Groundwork

***Note**: As long as you hold on to your lab report receipt, there's no need to go through these steps more than once.*

- Download and print the lab report receipt found at the course overview for DD2387 at KTH Social[3].
- Fill out the lab report receipt.
- Remember to ask for a signature after each oral presentation associated with a particular lab.

  All results are reported to `http://rapp.csc.kth.se`, and you are advised to check so that your results have been reported correctly. If there is missing reported results in rapp you must contact the course leader and present your signed report receipt.

How do I get credit for my work?

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 cprog14, before you try to submit any of your solutions.

You are free to make as many submissions as you wish, but please note that the implementation you would like to present during the oral presentation must be submitted to, and approved by, *Kattis*.

How do I answer questions associated with a particular assignment?

A file that contains your answers shall be attached to at least one (1) of your submissions to *Kattis*. Please note on submission you attached the file, so that it is easily accessed during your oral presentation.

If an assignment requires one or several written answers, a file named `inquery.txt`[4] will be available in the corresponding *lab directory*. `inquery.txt` includes every question associated with a particular assignment, and shall be used as a template for submitting your answers.

[3] `https://www.kth.se/social/course/DD2387/`
[4] **TODO:** add link

Additional Information

- When this document refers to files in the "*lab directory*", it is referring to the contents of `/info/DD2387/labs/lab1`, which can be accessed through `u-shell.csc.kth.se`.
- When this document refers to files in the "*assignment directory*", it is referring to a directory within the *lab directory*, that corresponds to the current assignment.

  As an example, `/info/DD2387/labs/lab1/0.1_make_it_happen` is the *assignment directory* associated with assignment "0.1 Make It Happen".

You may also find links to the relevant data by browsing the contents of:

- `http://www.csc.kth.se/utbildning/kth/kurser/DD2387/kurskatalog/`

Additional information, and the latest version of this document, is available at the course web:

- `https://www.kth.se/social/course/DD2387`.

Contents

0 The Essential (mandatory assignments)

0.1 Make It Happen (compilers, build systems)

This section will focus on the basics of compiling source code. Please use the literature associated with this course, as well as resources online to acquire further information about the topics discussed.

0.1.1 How do I compile my implementation?

This course will use `g++` to compile source code into object files, as well as linking them to produce a final executable. The current recommended, and used, version is `4.8.1`[5].

There is a file named `hello_world.cpp` in the directory associated with this assignment.

```
> cat /info/DD2387/labs/lab1/0.1_make_it_happen/hello_world.cpp
#include <iostream>

int main () {
  std::cout << "Hello, world!\n";
}
```

To compile the source code, and effectively produce an executable, we may invoke `g++` in the manner stated below:

```
> cp /info/DD2387/labs/0.1/hello_world.cpp .
> g++ -o say_hello.out hello_world.cpp
> ./say_hello.out
```

If there are any problems during compilation, such as trying to compile an ill-formed program, `g++` will print diagnostics related to such, and no executable will be created.

0.1.2 *GNU make*: a build system

For larger projects it is recommended to use a build system. One such system is called *GNU* ***make***; which is what we will use throughout this course.

`make` will read the contents of a file named `makefile` in the current working directory. The contents of this file shall be rules specifying how to build parts of your project, or simply put; how to produce an executable.

There is a file named `makefile` in the *assignment directory*, make a copy of it in the current working directory, and look at its contents by invoking `more makefile`.

```
> cp /info/DD2387/labs/lab1/1.1/makefile .
> more makefile
%.out: %.cpp
 g++ -std=c++0x -g -Wall $*.cpp -o $*.out
```

[5] You may have to change the default gcc compiler by invoking the following in a terminal: `module add gcc/4.8.1`.

Instead of messing around with a rather complex invocation of `g++`, you can now produce an executable named `hello_world.out`, compiled from a source file named `hello_world.cpp`, by simply invoking `make hello_world`.

```
> make hello_world
> ./hello_world.out
Hello, world!
```

0.1.3 Questions

- What does `$*` mean inside the `makefile`?
- What is the purpose of `-Wall` and `-g`, when passed as arguments to `g++`?
- What is the difference between an object file, and an executable?

0.2 Hello World (iteration statements, pointers, make)

`hello, world` is a classic program that dates back to 1974, first published in a paper titled *Programming in C: A tutorial*. The program has one simple purpose; to print "*hello, world*".

Since the typical implementation is very trivial, your task is to write a more versatile alternative, having the following semantics:

```
> ./hello
Hello, world!
> ./hello "DD2387"
Hello, DD2387!
> ./hello "KTH" 3
Hello, KTH KTH KTH!
> ./hello "Malcom X" NaN
error: 2nd argument must be an integral greater than zero!
> ./hello kth dd2387 3
error: Too many arguments!
```

Note: *This assignment is not an exercise in object oriented programming (OOP), but a mere introduction to the fundamental parts of C++.*

0.2.1 Requirements

- The `main`-function shall be written in a file called `main.cpp`.
- You shall implement a function which is responsible for printing the *hello world*-string, which shall be defined in a separate `.cpp-file` (*Translation Unit*).
- The separate `.cpp` shall have a corresponding `.h`-header that contains a *forward-declaration* for the function therein. This header is to be `#include`d by `main.cpp`.
- The definition of your print-function shall be compiled seperately, and later linked with `main.cpp`.
- Correct output from your program should be printed through `std::cout`, whereas error diagnostics should be printed through `std::cerr`.
- The implementation shall pass the tests done by invoking the following command:

  ```
  > /info/DD2387/labs/lab1/0.2_hello_world/hw_verifier <your-program>
  ```

0.2.2 Hints

- Use an argument named `argc` to `main` to get the number of arguments passed to your application. Remember that the name of the executable counts to this number.
- Use an argument named `argv` to `main` to get access to the individual parameters passed to your application.
- `std::atoi` from `<cstdlib>` can be used to convert a `char const *` to an integer. If the function is unable to interpret the data as an integer, it will return `0`.

0.2.3 Questions

- What is the purpose of `std::cout`, `std::cerr`, and `std::clog`, respectively?

0.3 Train Spotting (debugging)

There is a source file named `weird.cpp` in the *assignment directory*. Read through its source code and try to reason about the runtime behavior of the program without running it.

```
int powerof (int x, int y) {
  int res = 1;

  for (int i = 0; i < y; ++i);
    res *= x;

  return res;
}

int main () {
  int const a = 2;
  int const b = 4;

  int   x = powerof(a, b);
  float y = 3.1415;

  std::cout << a << "^" << b << " = " << x << ";\n";

  if (y == 3.1415)
    std::cout << y << " is equal to 3.1415!\n";
  else
    std::cout << y << " is not equal to 3.1415!\n";
}
```

Compile and execute the program. Hopefully you notice how the behavior differs from what one might expect. Your task is to figure out why that is with the help of a debugger.

0.3.1 Using a debugger

The *course directory* includes three documents, each serving as a light introduction to the most commonly used debuggers available under *Windows*, *Mac OS*, and *Unix/Linux*, respectively.

***Note**: The relevant documents are not ready to be published at the current time of writing; if you would like to go ahead with this assignment, you are advised to google for a suitable debugger for your platform.*

Use the debugger of your choice to identify why the program does not behave as one might have anticipated.

0.3.2 Questions

- Why does not `powerof` return the expected value (`16`), when invoked with `2` and `4`?
- Why does not `y` compare equal to `3.1415`?

- Is there any difference in behavior if we compare `y` to `3.1415f`, if so; why?
- What is the recommended method to use when trying to determine if two floating-point values are equal, and why?

0.4 Does It Fit? (unit testing)

There is a file named `count_if_followed_by.cpp` in the *assignment directory*:

```
// .-------------------------------------------------------
// |        count_if_followed_by (data, len, a, b);
// |
// | About: Returns the number of occurances of
// |        a followed by b in the range [data, data+len).
// '-------------------------------------------------------

int count_if_followed_by (char const * p, int len, char a, char b) {
  int        count = 0;
  char const * end = p + len;

  while (p != end) {
    if (*p == a && *(p+1) == b)
      count += 1;

    ++p;
  }

  return count;
}
```

0.4.1 `cxxtest`; a unit test framework

A unit test framework, such as *cxxtest*, allows a developer to specify constraints, and the expected behavior, of an implementation that he/she would like to test.

These rules are later used to generate *unit tests*. These *unit tests* will test to see that an implementation behaves as it shall (according to the previously stated specification).

The steps associated with using a unit test framework for C++ typically includes the following:

1. Specify the constraints and requirements that you would like to test.
2. Ask the unit test framework to generate a *test runner* having semantics associated with your specifications.
3. Compile the *test runner* into an executable.
4. Invoke the executable to commence testing.

0.4.1.1 Generating a *test runner*

There is a file named `simple.cxxtest.cpp` in the *assignment directory*.

Asking *cxxtest* to generate a *test runner* from the contents of this file can be accom-

plished through the following:

```
> cp /info/DD2387/labs/lab1/0.4_does_it_fit/simple.cxxtest.cpp .
> /info/DD2387/labs/cxxtest/cxxtestgen.py --error-printer \
  -o simple_testrunner.cpp simple.cxxtest.cpp
```

0.4.1.2 Compiling the *test runner*

Before we can execute our *test runner*, the *test runner* itself must be compiled it into an executable. This includes linking it together with an object file that contains our implementation.

Create an object file of our implementation:

```
> cp /info/DD2387/labs/lab1/0.4_does_it_fit/count_if_followed_by.cpp .
> g++ -c -o count_if_followed_by.o count_if_followed_by.cpp
```

Compile our *test runner*, and link it with the object file:

```
> g++ -o simple_test.out -I /info/DD2387/labs/cxxtest/ \
  simple_testrunner.cpp count_if_followed_by.o
```

The test can be run by invoking `./simple_test.out`.

0.4.1.3 Writing a test using *cxxtest*

***Note**: You may simplify the task of generating, and compiling, test runners by writing a new rule inside your `makefile`.*

There is an intentional bug in the definition of `count_if_followed_by`; it will potentially access one element outside the range specified. Collectively, bugs of this sort is most often referred to as "*off-by-one errors*".

```
// expected: result == 0
//  outcome: result == 1 (!!!)

char const data[4] = {'G','G','X','G'};
int  const result  = count_if_followed_by (data, 3, 'X', 'G');
```

0.4.2 Requirements

- Submit three (3) different tests to *Kattis* that test the correct, and incorrect, behavior of `count_if_followed_by`. The tests shall be presented during your oral presentation.

0.4.3 Questions

- Why is it important to test the boundary conditions of an implementation, especially in the case of `count_if_followed_by`?

0.5 Will It Float? (temporaries, resource management, valgrind)

0.5.1 `birth.cpp`

There is a source file named `birth.cpp` in the *assignment directory*, you are to copy this file and analyze the behavior of the compiled program.

It is recommended to use a debugger, or to adding print-statements in the source code, to make it easier to reason about its runtime behavior.

0.5.1.1 Questions (`birth.cpp`)

- What different constructors are invoked, and when?
- Will there be any temporaries created, if so; when?
- When are the objects destructed, and why?
- What will happen if we try to free a dynamically allocated array through `delete p`, instead of `delete [] p`?

0.5.2 `valgrind`; a memory management analyzer

`valgrind` is a popular tool for analyzing the memory management within applications written in C/C++, use it on the previously compiled executable.

```
> valgrind --tool=memcheck --leak-check=yes ./birth.out
```

0.5.2.1 Questions (`valgrind`)

- `valgrind` indicates that there is something wrong with `birth.cpp`; what, and why?

0.5.3 `bad_plumming.cpp`

There is source file named `bad_plumming.cpp` in the *assignment directory*, copy and compile this program, then run `valgrind` to analyze the correctness and behavior present.

0.5.3.1 Questions

- `valgrind` indicates that the program suffers from a few problems, which and why?
- If you uncomment the entire if-block in `foo`, is there any difference in how much memory that is leaked?
- If you change the last line of `main` to the following; why does valgrind still issue diagnostics related to memory management?

```
Data ** p = foo(v, size);
delete [] p;
```

0.6 The Simple Container (operator overloading, memory management)

Your task is to write an implementation of `class UIntVector`, a container that can store any arbitrary number of positive integers (`unsigned int`).

0.6.1 Requirements

- It shall be possible to create an empty `UIntVector`, having zero (0) elements.
- Appropriate constructors shall be `explicit`.
- Your implementation of `UIntVector` shall include;
  - a constructor taking a single argument of type `std::size_t` that specifies the number of *zero-initialized* elements to be stored in the container, and;
  - a copy/move-constructor, and;
  - a constructor taking a `std::initializer_list`, and;
  - a copy/move-assignment operator taking another `UIntVector` (potentially of a different size), and;
  - overloads of `operator[]` that makes it possible to access/modify elements at a desired index.
    - The first element of the container shall be at index `0`.
    - An exception of type `std::out_of_range` shall be thrown if a user tries to access an index out-of-bounds.
  - The following *member-functions* shall be implemeted:

`void reset()`	Assigns `unsigned int{}` to each element in the container.
`std::size_t size()`	Returns the number of elements in the container.

- Your implementation shall be uploaded to, and approved by, *Kattis*.

0.6.2 Hints

- Modifying the contents of a copied-to `UIntVector`, shall not change the contents of the copied-from `UIntVector`.
- Assigning a vector to itself might seem silly, but you are to make sure that it is handled correctly.
- Make sure that *member-functions* that does not change the internal state of the `UIntVector` is marked as `const`.
- There is a simple *cxxtest*, named `test_vec.cpp`, in the *lab directory* that is intended to tests the most fundamental parts of your implementation.

  `test_vec.cpp` has intentionally been left incomplete; recommended practice is for you to further develop it.

0.6.3 Questions

- `operator[]` must in some cases be marked as `const`, but not always; when, and why?
- The semantics of copying a `UIntVector` might not be trivial; why must we manually implement the relevant code, instead of having the compiler generate it for us?

0.7 The Template Container (templates, iterators, ...)

The previously implemented `class UIntVector` serves its purpose, but what if we would like to store an arbitrary type `T` in a similar container, without the need to write a new implementation from scratch?

Your task is to write a class template that, when instantiated as `Vector<T>`, yields a type that can store any arbitrary number of elements of type `T`.

0.7.1 Requirements

Your class template implementation shall satisfy the requirements of `UIntVector`, as well as the following:

- The class template shall be able to be instantiated as `Vector<T>`, where `T` denotes the type of the contained elements.
- It shall not be possible to instantiate the class template unless the specified element type is both `MoveConstructible` and `MoveAssignable`.

  You shall use `static_assert`, with an appropriate error message, to make sure that this is the case.
- Additional initialization functionality:
  - It shall be possible to *default-construct* the container, which shall be semantically equivalent to `Vector<T> (0)`.
  - It shall be possible to construct an instance of the container by passing the initial number of elements, as well as the initial value for those elements, as in; `Vector<float> (10, 3.14f)`.

- The following *member-functions* shall be implemented.

	Requirements
`void push_back(T)`	Appends the given element value to the end of the container with *amortized constant* time complexity, meaning that insertions are constant in most cases.
`void insert(std::size_t, T)`	Inserts the element value immediately before the index specified. If `index == size()`, see `push_back`. If the index is out-of-bounds; throw a `std::out_of_range`.
`void clear()`	Removes every element, effectively making `size() == 0`.
`void erase(std::size_t)`	Removes the element at the index specified.
`std::size_t size()`	Returns the number of elements in the container.
`std::size_t capacity()`	Returns the number of elements that can potentially be stored in the container without having to reallocate the underlying storage.
`unspecified` `begin()`	Returns a *RandomAccessIterator* to the first element of the range.
`unspecified` `end()`	Returns a *RandomAccessIterator* referring to the element right after the last element in the container.
`unspecified` `find(T const&)`	Returns a *RandomAccessIterator* referring to the first element that compares equal to the argument, or `end()` if no such element is found.

0.7.2 Hints

- Make sure that you mark the appropriate member-functions as `const`.
- `<type_traits>` contains *traits*[6] that can be used to check whether a certain type has some characteristic, such as to see if a type `T` is *MoveConstructible*.
- There is a simple *cxxtest*, `test_template_vec.cpp`, available in the *lab directory*, which intentionally is somewhat incomplete.

  You should further increase its functionality to make sure that your implementation satisfies the requirements of this assignment.

[6] a *type trait* is an entity which can be used to query characteristic of any arbitrary type `U` during compile-time.

0.7.3 Questions

- Iterating over a range of elements can be done with a *range-based for-loop*, but the type of *source* must meet certain requirements; what are they?

```
for (auto const& elem : source) {
   ...
}
```

- The C++ Standard sometimes state that a type in the Standard Library is *unspecified*; why do you think that is?

> 20.11.7.3 Class `high_resolution_clock` [time.clock.hires]
>
> Objects of class `high_resolution_clock` represent clocks with the shortest tick period. `high_resolution_clock` may be a synonym for `system_clock` or `steady_clock`.
>
> ```
> class high_resolution_clock {
> public:
>   typedef unspecified                                  rep;
>   typedef ratio<unspecified, unspecified>              period;
>   typedef chrono::duration<rep, period>                duration;
>   typedef chrono::time_point<unspecified, duration> time_point;
>   static const bool is_ready =                         unspecified;
>
>   static time_point now() noexcept;
> };
> ```

1 The Progressive (assignments for extra credit)

1.1 The Matrix (10p, C)

Extrauppgift 1.1 (10p, krav för betyg C) Skriv en dynamiskt allokerad matrisklass `Matrix`. På kursbiblioteket finns en header-fil Matrix.h som du ska utgå ifrån. Observera att det finns ett eller fler medvetna designfel i headerfilen. Matrix.h ska använda din vektorimplementation i lab1. För att man inte ska kunna lägga till extra element på en rad eller kolumn används internt en klass `matrix_row`.

```
matris[7].push_back(1);  // Ej tillåtet
```

Matrisklassen ska ha en del funktionalitet som förklaras nedan.

Åtkomst till elementen ska ges enligt följande exempel:

```
int x = matris[7][2];
matris[3][1] = x;
```

Låt matrisklassen definiera

```
std::ostream &operator<<(std::ostream &, const Matrix &)
```

så att man kan skriva ut matrisen med `cout` med rader och kolumner.

Definiera även en inmatningsoperator (operator») så att man kan mata in värden i en matris på matlab format [1 2 0; 2 5 -1; 4 10 -1] ingen felhantering behöver implementeras för felaktig inmatning. Första tecknet är alltid `[`.

```
Matrix m;
std::cin >>  m;
```

Användaren matar in [1 2 -3 ; 5 6 7]

```
std::cout << m << std::endl;
```

Utskriften visas nedan till vänster. Till höger visas samma utskrift men med understrykningstecken istället för mellanslag (notera mellanslag sist på raden). Testa utskriften med cxx_test genom att skriva till en strängström.

```
[ 1 2 -3                    [_1_2_-3_
; 5 6 7 ]                   ;_5_6_7_]
```

Implementera aritmetik för matriser. Implementera tilldelningsoperator och kopieringskonstruktor samt identitet (sätter kvadratisk matris till identitetsmatrisen), negation och transponering. Överlagra operatorer för matrisaritmetik.

`+`, `-` och `*` samt `*` för skalärmultiplikation.

Tips: Tänk över retur- och argumenttyper: vilka är `const` och vilka kan inte vara referenser? Vilka funktioner är `const`? En del av operatorerna delar funktionalitet. Utnyttja detta för att underlätta implementationen.

Skriv testfall för dina metoder. Skriv även testfall som inte ska fungera (matrisernas dimension är fel). Exempel på testfall, skalär- och matrismultiplikation av 0-stora, 1-stora matriser, kvadratiska, rektangulära matriser. Kedjeaddition och multiplikation av matriser. Vad finns det för designfel i Matrix.h? Vad borde man kunna göra som man inte kan göra?

På kursbiblioteket finns nio felaktiga matrisimplementationer. De ligger i underbibliotek kompilerade för ubuntu. Matriserna är kompilerade med std::vector och Matrix.h är ändrad därefter.

Skriv testfall som fångar felen i varje matris. Testfall 5 och 9 är tillståndsfel efter utskrift och tilldelning och kan vara svåra att träffa. Samla flera testfallsanrop i en testfunktion och anropa den efter utskrift/tilldelning. Testa även dina testfall på din matrisimplementation (som bygger på din egen vektor). Din testkod är oberoende av vektor men testkoden måste kompileras om med de felaktiga matriserna.

För att bygga med en buggig matrisimplementation finns en färdig Makefile på kurskatalogen. Kopiera hela katalogen. Ändra eventuellt i sökvägen till cxxtest i Makefile. Bygg första testet med `make runtest01`

Vid redovisningen ska du kunna redogöra för alla delar i din kod, även den kod du inte skrivit (t.ex. `matrix_row`). Du ska kunna svara på hur Matrix.h kan förbättras och ha gissningar på vad som är galet i de olika matrisimplementationerna du testat.

1.2 Running Through The Matrix (4p)

Extrauppgift 1.2 (4p) Använd `Matrix` för att implementera en labyrintlösare. Låt elementen i matrisen motsvara en ruta i labyrinten. Skriv ut den slutgiltiga lösningen (utan återvändsgränder). Prova din labyrintlösare med filen `maze.cpp`. Skapa en funktion `read(const char **data)` som initierar matrisen med en labyrint. För den som vill ha större/andra labyrinter finns en labyrintgenerator `maze_generator.cpp` som genererar C++-syntax.

1.3 *Reserved*

Intentionally left blank.

1.4 The Hypercube (5p)

Extrauppgift 1.4 (5p) 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 (6p)

Extrauppgift 1.5 (6p) 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.

1.6 Walking Across The Universe (5p, A)

Extrauppgift 1.6 (5p, krav för betyg A) 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. Det krävs inte men du bör kunna resonera om vad som saknas i din lösning.

1.7 Bit Me (3p)

Extrauppgift 1.7 (3p) Låt `Vector<bool>` överlagra operatorer för booleska operationer: `~`, `&`, `|` och `^`. Använd datorns hårdvara i största möjliga utsträckning genom att använda booleska operationer på `unsigned int`.

Skapa även minst tre funktioner `weight` som räknar antalet satta bitar (ettor). Använd de booleska operationerna för detta, d.v.s. gör beräkningen utan att titta på varje bit separat. Ett sätt att göra det är att mappa 256 bitmönster i en array och helt enkelt slå upp antalet bitar i varje byte. Ett annat sätt är att räkna ettor och ta bort den högraste ettan i varje iteration. Ytterliggare en variant är att invertera alla ettor och nollor innan man räknar högraste ettan. En mer matematisk variant är följande tvåradare som utan slinga räknar ettor i ett 32-bitars tal.

```
xCount = x - ((x >> 1) & 033333333333) - ((x >> 2) & 011111111111);
return ((xCount + (xCount >> 3)) & 030707070707) % 63;
```

Redovisa några tester där respektive variant fungerar bäst. Tänk också på vad som händer om vektorerna har olika storlek.

1.8 The Master of Life & Death (Xp)

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 shall be properly constructed, and that there shall 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 shall be called, likewise; if a reallocation of the underlying storage is needed, a container shall not construct any more objects than there are elements present in the range.

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

1.8.1 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);
}
```

1.8.2 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[7].
- Your implementation shall be submitted to, and approved by, *Kattis*.

[7] The time complexity of insertion shall still be *amortized constant*.

1.9 Concurrency is The New Black (Xp)

***Note**: unfinished...*

1.10 Better Safe Than Sorry (Xp)

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 shall 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 written to be 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 shall never throw an exception.

1.10.1 Requirements

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

1.10.2 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 constructor might throw an exception, there is no way to "*rollback*" to the previous state.
- There are many tools available in the Standard Library, start by looking at the entities available in `<memory>`.