DD2387 Programsystemkonstruktion med C++

Lab 1: The Essentials

1th of September 2015

Introduction

The purpose of this lab is to teach you how to use 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 implementations.

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

Please read through the entire 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 is easy to read, maintain, and understand. This includes the usage of correct indentation, as well as having suitable names for your variables.
- Your implementations must not leak memory, or any other acquired resource.

  Pay attention to the purpose of constructors, and their relationship with the corresponding destructor: *Resource Acquisition Is Initialization*[1].
- Your implementations should demonstrate that you fully understand the semantics associated with the keyword `const`.
- You are not allowed to use the containers available in the Standard Library to solve the assignments listed in this document[2], unless this has been explicitly stated to be allowed for some assignment.
- Questions that require a written answer should be answered in an elaborate manner; single word answers will not be accepted. Each answer must consist 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 results are missing in rapp you must contact the course leader and present your signed report receipt.

Submitting and presenting your 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 cprog15, 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 implementations you would like to present during the oral presentation must be submitted to, and be approved by, *Kattis*.

How do I answer questions associated with a particular assignment?

A file that contains your answers should be attached to at least one (1) of your submissions to *Kattis*. Please make note of the submission id for the submission which you attached the file to, 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 should be used as a template for submitting your answers.

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

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 `4.8.4`.

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 thereby produce an executable we may invoke `g++` in the manner stated below:

```
> cp /info/DD2387/labs/lab1/0.1_make_it_happen/hello_world.cpp .
> g++ -o say_hello.out hello_world.cpp
> ./say_hello.out
```

where we first copy the source code, compile it and then run it.

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. This file contains 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/lab1/0.1_make_it_happen/makefile .
> more makefile
%.out: %.cpp
 g++ -std=c++11 -g -Wall $*.cpp -o $*.out
```

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.out
> ./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++`?
- Is it possible to store the previous mentioned flags in a variable in the makefile?
- What is the difference between an object file, and an executable?

0.2 Hello World (iteration statements, pointers, make)

There is a `makefile` in the *assignment directory* with contents that differ from the one viewed in the previous subsection. Make a copy of it in the current working directory.

You should also copy `main.cpp`, `hello.h`, and `hello.cpp`, since these are mandatory parts of the upcoming assignment.

```
> cp /info/DD2387/labs/lab1/0.2_hello_world/makefile .
> cp /info/DD2387/labs/lab1/0.2_hello_world/*.cpp .
> cp /info/DD2387/labs/lab1/0.2_hello_world/hello.h .
```

Look at the contents of `makefile`, and invoke `make` to see the resulting behavior. Try to build the executable two times without making any changes to the dependent files.

```
> make
g++ -std=c++11 -g -Wall -pedantic -c hello.cpp
g++ -std=c++11 -g -Wall -pedantic main.cpp hello.o -o main.out
> make
make: 'main.out' is up to date.
```

Upon invoking `make`, the program will look for *dependencies* that are more recently modified than the *target file*. If there are no such file, and a target file is already present, `make` will decide that there is nothing to do; why work for something that has already been done?

`touch` can be used to change the modification timestamp of a file to the current time (effectively making it look *as if* the file has been modified).

Change the modification timestamp of `main.cpp` by invoking `touch`.

```
> touch main.cpp
```

Now invoke `make`, and notice that `hello.cpp` will not be recompiled.

```
> make
g++ -std=c++11 -g -Wall -pedantic main.cpp hello.o -o main.out
```

0.2.1 Questions

- If you invoke `touch hello.cpp` prior to invoking `make`;
  - How many files were rebuilt?
  - Why?
- Why do you think `make` checks the modification timestamp of the relevant files as part of deciding what to do?

0.2.2 Change compiler

There is an alternative compiler (`clang`) installed on the ubuntu systems. To switch from using `gcc` to `clang`, modify the `makefile` and uncomment the following line (by removing `#`).

```
#CC   = clang++-3.6 ...
```

Note: *The above line has been shortened due to lack of space.*

Using a different compiler can often allow you to view potential errors from a different angle (mostly because different compilers present errors in different ways), making it easier to realize what went wrong, and more importantly; why.

Build and run the program.

0.2.3 Assignment

`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 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 "KTH" 0
> ./hello "KTH" 2
Hello, 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++.*

0.2.4 Requirements

- `main.cpp` contains the definition of the applications `main`-function. There is no need for you to modify this file, but please make sure that you understand what is written in it.
- `hello.cpp` has a corresponding `.h`-header that contains *forward-declarations* for the functions you are to implement.

  This header, `hello.h`, is `#include`d by `main.cpp`, and should be included by `hello.cpp`.

- `hello.cpp` shall be compiled seperately, resulting in an *object file* that is to be linked into the resulting executable.
- You shall implement a function named `parse_args` in a separate *translation unit* named `hello.cpp`.

  This function is responsible for parsing/interpreting the command line arguments passed to the application, and shall return a `std::pair` with what to print, and how many times to print it.

  `parse_args` shall handle all input errors by returning `-1` as the second value of the returned `std::pair`, as well as printing a suitable error message on `stderr`.
- You shall implement a function named `hello` in a separete *translation unit* named `hello.cpp` (i.e. the same file that shall contain the implementation of `parse_args`).

  `hello` is responsible for printing the *hello world*-string on *stdout* with the following semantics;

  - If the value *zero* is passed as the second parameter, nothing shall be printed.
  - If a value greater than *zero* is passed as the second argument (`count`), `"Hello, "` shall be printed followed by `count` space-delimited occurrences of the first argument.

    The output shall end with an exclamation mark, followed by a new-line.

    Note: `hello ("KTH", 3)` *shall print* Hello, KTH KTH KTH! *(see the previous example invocations of the program for more information).*
- Correct output from your program shall be printed through `std::cout`, whereas potential error diagnostics (detected by `parse_args`) shall be printed through `std::cerr`.
- Your implementation, `hello.cpp`, shall be uploaded to, and approved by, *Kattis*.

0.2.5 Hints

- The argument named `argc` to `main` will contain the number of arguments passed to your application. Remember that the name of the executable counts to this number.
- The second argument, `argv`, will provide access to the individual arguments 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`.

0.2.6 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 accompanying debugger for gcc is called `gdb`. It uses a command line interface. There is a graphical user interface to gdb called `ddd` on the lab computers. If you use your own computer there may be other debuggers available (lldb, visual studio, eclipse etc). Feel free to google and install an alternative C++ debugger.

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 add print-statements in the source code, to make it easier to reason about its runtime behavior.

0.5.1.1 Questions (`birth.cpp`)

- What 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.

0.5.3.1 Questions

- `valgrind` indicates that the program suffers from a few problems, which and why?
- If you comment out the entire if-block in `foo`, is there any difference in how much memory that is leaked?
- Revert `bad_plumming.cpp` to its original state, then only comment out the line that contains the if-condition.

```
for (int i = 0; i < x; i++)
    // if (v[i] != 0)
        v[i] = new Data;
```

If you now change the last line of `main` to the below; why is it that 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 should 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 should be uploaded to, and approved by, *Kattis*.

0.6.2 Hints

- Modifying the contents of a copied-to `UIntVector`, should 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 should satisfy the requirements of `UIntVector`, as well as the following:

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

  You should use `static_assert`, with an appropriate error message, to make sure that this is the case.
- Additional initialization functionality:
  - It should be possible to *default-construct* the container, which should be semantically equivalent to `Vector<T> (0)`.
  - It should 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*[5] 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.

[5] 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 states 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;
> };
> ```