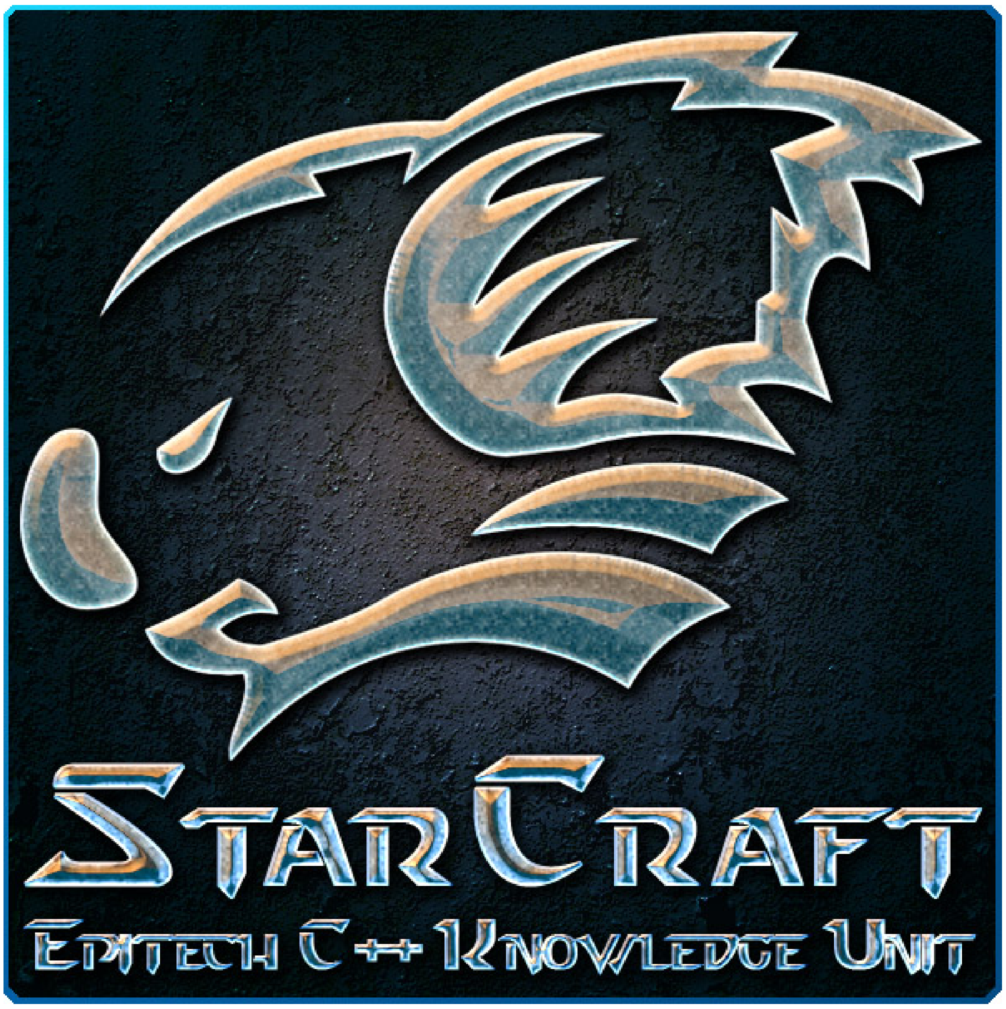
STARCRAFT 2

Koalab koala@epitech.eu

Abstract: *Starcraft is a real-time strategy video game developed and edited by Blizzard Entertainment. During this exam, you will reproduce some of its basic principles.*



Contents

Chapter I

Instructions

* Incoming transmission *

During this test, you will help the Queen of Blades to create her units, to make them evolve and control them efficiently, in order to absorb and control every trace of life from the Earth...

You must put an `auteur` file, at the root of your `rendu` folder, that will contain your login followed by a carriage return.

```
>cat auteur
koala
>
```

This test is composed of 4 exercises, each counting for 5 points, simulating missions that you could be required to perform under real conditions. So stay focused because no error can be tolerated, and correction will stop at the first mistake encountered.

Under real conditions, time will be your biggest enemy. You only have 4 hours to complete this tests serie.

In your `rendu` folder, each exercise will be put in an independant folder. The folder name will match with the exercise number.
Ex: ex0/, ex1/, ...

Some exercises use the code written in previous exercise(s), so you will have to copy these files if necessary.

Moreover, the code contained in each folder must not contain a `main` function, as we will use our own `main` to compile and test your exercises. That's also why file names and class names must be scrupulously respected, if you want to get points.

Identifiers you will have to write must be written as following:

```
aFunctionName();
anotherFunction();
int     anInteger;
getSomething();
```

Every asked outputs must be done on the standard output, and will end with a carriage return.

The compiler to use is `g++` , with the compiling options `-W, -Wall, and -Werror.`

Remember to protect your header files.

Please note that all C functions (`*alloc, free, *printf*` , etc.), as well as the `using` keyword in C++, are illegal, and their use will be punished by the Order of the -42, consisting in making you benefit from several genetic mutations that our organical nature can offer.

You have been warned, the Queen of Blades won't accept defeat!

`If an exercise header mentions "no forbidden function", that means "no more forbidden function than previously".`

The `switch` , successive `if` (and/or `else if`) are FORBIDDEN in this entire exam. Those points will be checked in the correction and every attempted fraud will be penalised by a -42.
Only bifurcations less or equal to `IF THEN (ELSE IF THEN ELSE)` are allowed.
Be sure to have read this paragraph CAREFULLY!

`Completely read the exercise before starting it, you will save time!`

* End of transmission *

Chapter II

Prologue

I am the Queen of Blades, and you will have 4 hours to help us. If you are not done by this time, the price to pay will be the infection, or maybe my little Zerglings will take care of you.

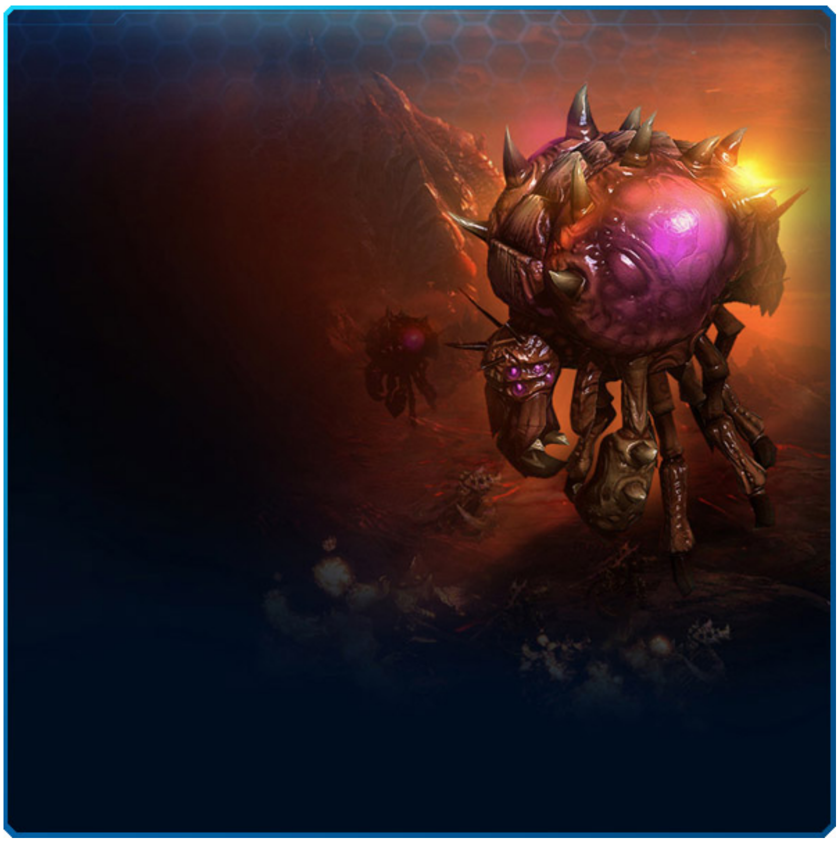
The threat is approaching fast, we have to form and train more fighting units to defend ourselves. Jim Raynor should be about to land on Char with his MMM (Marin, Marauder, Medivac). We must repel them, the hive survival depends on it.

In the next hours, you will first learn to form basic units, then you will make them able to evolve.

Chapter III

Exercise 0

	Exercise : 0	points : 5
Basic unit creation		
Turn-in directory: `/rendu/ex0`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `AUnit.hpp, AUnit.cpp, Overlord.hpp, Overlord.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		


You will create an Overlord, which will supply our troups

You will first learn to use the unlimited hive's potential. Since ten years we are on standby and waiting. Today we must relearn how to multiply ourselves and how to evolve as quickly as possible.

As a start, I want you to be able to create any unit type in order for me to control them. For that, you must meet the following expectations.

- `AUnit` abstract class creation
  - Create the `AUnit` abstract class, which will contain the following private properties:

```
- the x position               int           _x
- the y position               int           _y
- the number of damages made   int           _damages
```

  - This class will also contain the following public pure methods:

```
- void         attack(int x, int y);
- void         move(int x, int y);
```

  - Moreover, the `AUnit` class will have the following public member functions:

```
- int          getX(void) const;
- void         setX(int x);
- int          getY(void) const;
- void         setY(int y);
- int          getDam(void) const;
- void         setDam(int damages);
- void         pos(void) const;
```

  - You must implement these three getters, these three setters and the `void pos(void) const` method, which will print on the standard output the following message (the variable factors are prefixed with `@`), followed by a carriage return:

```
        Position x : @_x y : @_y <(8<) <(8)> (>3)>
```

  - Lastly, add a virtual destructor to the class.

- `Overlord` class creation

- Now create the `Overlord` class, inheriting from the `AUnit` class. This class will be of canonical form (of Coplien).
- Every `Overlord` constructors must display the following output, followed by a carriage return:

```
        Creation d'un Overlord
```

- At his creation, an `Overlord` object will have as position {0, 0}. He can't attack, so he can't make any damage.
- Add an explicit constructor that will take as a parameter a `const AUnit &` object and will copy its properties.
- At its destruction, an `Overlord` object must display the following output, followed by a carriage return:

```
        Destruction d'un Overlord
```

- This class implements the following methods:
  * `void attack(int x, int y)` , which will move the unit to the position defined by the `x` and `y` parameters, and then will display the following output, followed by a carriage return:

```
        Un Overlord ne peut attaquer
```

  * `void move(int x, int y)` , which redefines the `_x` and `_y` properties to the values given as paremeters, and displays the following output (the variable factors are prefixed with `@`), followed by a carriage return:

```
        Deplacement en x : @x y : @y
```

Here is an usage example:

```
{
  AUnit *overlord1 = new Overlord();

  overlord1->attack(6, 7);
  overlord1->move(1, 2);
  overlord1->pos();

  Overlord overlord2(*overlord1);
  Overlord overlord3;

  overlord3.pos();
  overlord2.pos();
  overlord2.move(42, 30);

  overlord3 = *overlord1;
  delete(overlord1);
}
```

The associated output would be:

```
$>./a.out | cat -e
Creation d'un Overlord$
Deplacement en x : 6 y : 7$
Un Overlord ne peut attaquer$
Deplacement en x : 1 y : 2$
Position x : 1 y : 2 <(8<) <(8)> (>3)>$
Creation d'un Overlord$
Creation d'un Overlord$
Position x : 0 y : 0 <(8<) <(8)> (>3)>$
Position x : 1 y : 2 <(8<) <(8)> (>3)>$
Deplacement en x : 42 y : 30$
Destruction d'un Overlord$
Destruction d'un Overlord$
Destruction d'un Overlord$
```

Chapter IV

Exercise 1

	Exercise : 1	points : 5
An attacking unit		
Turn-in directory: `/rendu/ex1`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `AUnit.hpp, AUnit.cpp, Overlord.hpp, Overlord.cpp, Zergling.hpp, Zergling.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

Zerglings are very fast creatures for the short-range combat

Jim Raynor and his Protoss allies are starting to discover our presence. Zeratul have reported our message, Terrans know that we are not defeated. I need a simple but efficient unit that will allow me to take the marines and their miserable weapons away, while waiting for us to be able to absorb them.

- `Zergling` class creation
  - Now create the `Zergling` class, inheriting from the `AUnit` class. This class will be of canonical form (of Coplien). Unlike the `Overlord` , a `Zergling` will be able to attack.
  - Every `Zergling` constructors must display the following output, followed by a carriage return:

```
        Creation d'un Zergling
```

  - At his creation, a `Zergling` object will have as position {0, 0}. He deals 5 damages at each attack.
  - Add an explicit constructor that will take as a parameter a `const AUnit &` object and will copy its properties.
  - At its destruction, a `Zergling` object must display the following output, followed by a carriage return:

```
        Zergling est mort
```

  - This class implements the following methods:
    - `void attack(int x, int y)` , which will move the unit to the position defined by the `x` and `y` parameters, and then will display the following output (the variable factors are prefixed with `@`), followed by a carriage return:

```
        Dommage Zergling : @_damages
```

    - `void move(int x, int y)` , which redefines the `_x` and `_y` properties to the values given as paremeters, and displays the following output (the variable factors are prefixed with `@`), followed by a carriage return:

```
        Deplacement en x : @x y : @y
```

- AUnit class improvement
  - Create a « operator overload, that will take as a parameter a `std::ostream &` object and a `const AUnit &` object. This overlad will be located in the `AUnit.cpp` file.

Here is an example of usage of modifications you just made:

```
{
  AUnit *overlord = new Overlord();
  AUnit *t = new Zergling();

  t->attack(6,7);
  overlord->attack(6,7);
  overlord->move(1, 2);
  overlord->pos();
  Overlord overlord2(*overlord);
  std::cout << *overlord << std::endl;
  overlord2.pos();
  overlord2.move(42, 30);
  std::cout << *overlord << std::endl;
  delete (overlord);
  delete (t);
}
```

The associated output should be:

```
$> ./a.out | cat -e
Creation d'un Overlord$
Creation d'un Zergling$
Deplacement en x : 6 y : 7$
Dommage Zergling : 5$
Deplacement en x : 6 y : 7$
Un Overlord ne peut attaquer$
Deplacement en x : 1 y : 2$
Position x : 1 y : 2 <(8<) <(8)> (>3)>$
Creation d'un Overlord$
Position x : 1 y : 2 Dommage : 0$
Position x : 1 y : 2 <(8<) <(8)> (>3)>$
Deplacement en x : 42 y : 30$
Position x : 1 y : 2 Dommage : 0$
Destruction d'un Overlord$
Zergling est mort$
Destruction d'un Overlord$
```

Chapter V

Exercise 2

	Exercise : 2	points : 5
And now the mutations		
Turn-in directory: `/rendu/ex2`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `AUnit.hpp, AUnit.cpp, Overlord.hpp, Overlord.cpp, Zergling.hpp, Zergling.cpp, Baneling.hpp, Baneling.cpp, Overseer.hpp, Overseer.cpp, Hatchery.hpp, Mutation.hpp, Mutation.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

Overlords can mutate in Overseer (on the left), which detect ennemies
Zerglings can mutate in Baneling (on the right), which are suicide units

Jim Raynor is no longer a problem. Protoss are a much bigger threat: their Immortals and their Dark Templars cause great upset to us. I want my `Zergling` to be able to "take care" of the Templars. In order to do so, they have to see them! Provide an evolution to my `Overlord` to make them able to detect the Dark Templars. At the same time, provide an evolution to my `Zergling` to take care of the Immortals.

- `Baneling` class creation
  - Now create a `Baneling` class, inheriting from `AUnit` . This class will be of canonical form (of Coplien).
  - Every `Baneling` constructors must display the following output, followed by a carriage return:

```
    Mutation d'un Zergling en Baneling
```

  - At his creation, a `Baneling` object will have as position {0, 0}. He deals 15 damages at each attack.
  - Add an explicit constructor that will take as a parameter a `const AUnit &` object and will copy its properties.
  - At its destruction, a `Baneling` object must display the following output, followed by a carriage return:

```
    Baneling a explose
```

  - This class implements the following methods:
    * `void attack(int x, int y)` , which will move the unit to the position defined by the `x` and `y` parameters, and then will display the following output (the variable factors are prefixed with `@`), followed by a carriage return:

```
    Dommage Baneling : @_damages
```

    * `void move(int x, int y)` , which redefines the `_x` and `_y` properties to the values given as paremeters, and displays the following output (the variable factors are prefixed with `@`), followed by a carriage return:

```
    Deplacement en x : @x y : @y
```

- `Overseer` class creation

    - Now create an `Overseer` class, inheriting from `AUnit`. This class will be of canonical form (of Coplien).
    - Every `Overseer` constructor must display the following output, followed by a carriage return:

```
Mutation d'un Overlord en Overseer
```

    - At his creation, an `Overseer` object will have as position {0, 0}. He can't make any damage.
    - Add an explicit constructor that will take as a parameter a `const AUnit &` object and will copy its properties.
    - At its destruction, an `Overseer` object must display the following output, followed by a carriage return:

```
Destruction d'un Overseer
```

    - This class implements the following methods:

        * `void attack(int x, int y)`, which will move the unit to the position defined by the `x` and `y` parameters, and then will display the following output, followed by a carriage return:

```
Un Overseer ne peut attaquer mais detecte
```

        * `void move(int x, int y)`, which redefines the `_x` and `_y` properties to the values given as paremeters, and displays the following output (the variable factors are prefixed with `@`), followed by a carriage return:

```
Deplacement en x : @x y : @y
```

- `Hatchery` class creation

- Create a `Hatchery` class, an incubator that will allow us to produce units.
- In order to create units, this class must contain a `static template create` member function, that will return an `AUnit *` object. The final created unit type will be given as the template parameter.

- Create a `Mutation` class, that will have a member function `static AUnit *muter(AUnit *)` . That fonction must be able to transform a `Zergling` given as a parameter into a `Baneling` , and an `Overlord` given as a parameter into an `Overseer` (the final returned object type depends on the received object type). The position of the returned unit must be the one where was the unit given as parameter. The latter, having mutated, must be destroyed. The returned unit will cause as much damage as all of the other units of its type.

Here is how we could use our new classes:

```
{
  AUnit *overlord1 = Hatchery::create<Overlord>();
  AUnit *t = Hatchery::create<Zergling>();

  t->attack(6,7);
  t->pos();
  t->move(1, 2);
  t->pos();
  std::cout << *t << std::endl;
  t = Mutation::muter(t);
  std::cout << *t << std::endl;
  t->pos();
  t->attack(7,8);
  overlord1->attack(9,9);
  overlord1->move(1, 2);
  overlord1->pos();
  Overlord overlord2(*overlord1);
  overlord2.pos();
  overlord2.move(42,30);
  AUnit *tmp = Hatchery::create<Zergling>();
  delete(overlord1);
  delete (tmp);
}
```

The associated output would be:

```
$> ./a.out | cat -e
Creation d’un Overlord$
Creation d’un Zergling$
Deplacement en x : 6 y : 7$
Dommage Zergling : 5$
Position x : 6 y : 7 <(8<) <(8)> (>3)>$
Deplacement en x : 1 y : 2$
Position x : 1 y : 2 <(8<) <(8)> (>3)>$
Position x : 1 y : 2 Dommage : 5$
Mutation d’un Zergling en Baneling$
Zergling est mort$
Position x : 1 y : 2 Dommage : 15$
Position x : 1 y : 2 <(8<) <(8)> (>3)>$
Deplacement en x : 7 y : 8$
Dommage Baneling : 15$
Deplacement en x : 9 y : 9$
Un Overlord ne peut attaquer$
Deplacement en x : 1 y : 2$
Position x : 1 y : 2 <(8<) <(8)> (>3)>$
Creation d’un Overlord$
Position x : 1 y : 2 <(8<) <(8)> (>3)>$
Deplacement en x : 42 y : 30$
Creation d’un Zergling$
Destruction d’un Overlord$
Zergling est mort$
Destruction d’un Overlord$
```

Chapter VI

Exercise 3

Exercise : 3	points : 5
The cerebral supervisor	
Turn-in directory: `/rendu/ex3`	
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`
Makefile: `No`	Rules: `n/a`
Files to turn in : `AUnit.hpp, AUnit.cpp, Overlord.hpp, Overlord.cpp, Zergling.hpp, Zergling.cpp, Mutation.hpp, Mutation.cpp, Baneling.hpp, Baneling.cpp, Overseer.hpp, Overseer.cpp, Hatchery.hpp, ICommand.hpp, Command.hpp, Controller.hpp, Controller.cpp`	
Remarks : `n/a`	
Forbidden functions : `None`	

Protoss are now defeated, just in time! But Jim Raynor has teamed up with the Dominion to be able to beat me. There is no possibility of that occuring! Jim found a Xel'Naga artefact that interferes on my control over my troops. You have to get me a cerebral controller, allowing me to give orders to my Zergs.

- Create an `ICommand` interface, that includes a `void execute(void)` pure method (and a virtual destructor).
- `Command` class creation
  - Now create a `Command` template class, inheriting from the `ICommand` interface
  - In order to construct a `Command` object, we need 4 parameters: the first is an instance of the class used in the template, the second is a method pointer in the form of `void Class::fct(int, int)` , the two remaining will be the two integers given as parameter when using the method pointer. This constructor won't display anything on the standard output.

    ◦ This class implements the method `void execute(void)` :
      This method calls the method pointer with the two saved integers.

- In order to use `Command` , create a `Controller` class. This class contains a `ICommand *` stack, of `std::stack` type and named `_tasks` , as well as two member functions:

    ◦ `void addTask(ICommand *)` , that adds the `ICommand` object given as parameter in its stack, and displays on the standard output the following message, followed by a carriage return:

```
        Tache ajoutee
```

    ◦ `void execute(void)` , which waits for having three tasks before executing them. So in the case where less than three tasks have been added, the method displays on the standard output the following message, followed by a carriage return:

```
        Nombre de taches inferieur a 3
```

Otherwise, all tasks contained in the stack are executed (the stack is emptied as the methods are called).

Here is an example use of the classes we just made:

```
{
  AUnit *overlord = Hatchery::create<Overlord>();
  AUnit *t = Hatchery::create<Zergling>();
  Controller controller;

  controller.addTask(new Command<AUnit>(*overlord, &AUnit::attack, 0, 0));
  controller.execute();
  controller.addTask(new Command<AUnit>(*overlord, &AUnit::move, 1, 42));
  controller.execute();
  controller.addTask(new Command<AUnit>(*t, &AUnit::attack, 0, 5));
  controller.addTask(new Command<AUnit>(*t, &AUnit::attack, 0, 12));
  controller.execute();
  delete(overlord);
  delete(t);
}
```

The associated output should be:

```
$> ./a.out | cat -e
Creation d'un Overlord$
Creation d'un Zergling$
Tache ajoutee$
Nombre de taches inferieur a 3$
Tache ajoutee$
Nombre de taches inferieur a 3$
Tache ajoutee$
Tache ajoutee$
Deplacement en x : 0 y : 12$
Dommage Zergling : 5$
Deplacement en x : 0 y : 5$
Dommage Zergling : 5$
Deplacement en x : 1 y : 42$
Deplacement en x : 0 y : 0$
Un Overlord ne peut attaquer$
Destruction d'un Overlord$
Zergling est mort$
```

Chapter VII

Epilogue

Jim Raynor and his miserable allies are defeated! Protoss are a mere shadow of their former self and will be very soon at my mercy! Jim Raynor himself is dead, and his troops will be soon absorb. The Dominion no longer exists, and the way to the Earth is now open...

Chapter VIII

Legal Notices