NTNU – Trondheim
Norwegian University of Science and Technology
Department of Computer and Information Science

Examination paper for TDT4102 - Procedural- and Objectoriented Programming

Academic contact during the exam: Trond Aalberg

Phone: 97631088

Examination date: 03. June 2015

Examination time: 09-13

Permitted examination support material: C: Specified printed and hand-written support material is allowed.

Allowed printed book: Walter Savitch, Absolute C++ or Lyle Loudon, C++ Pocket Reference.

Language: English

Number of pages: 11 including front page and appendix

Checked by by

Date	Signature

General introduction

Read the assignments carefully. Some descriptions are elaborated, but this is to give sufficient context, introduction and examples for the assignments.

When the phrases "*implement*" or "*write*" is used, we request a functioning implementation. E.g. if the assignment is to implement a function then the answer is to write the complete function including the declaration with appropriate parameters and return type, and the function body.

If the phrase "*declare*" is used, we are only interested in a function or class declaration. This is typically the code you will find in a header file.

If we ask you to "*explain*" you are free to decide how to answer, but use simple lines of code or short textual descriptions and be precise and avoid lengthy text.

If you find that information is missing from an assignment, explain any assumptions and prerequisites you find it necessary to make.

All code has to be in C++. The exam does not require knowledge of other classes and functions than the ones you have become familiar with in the exercises. Include statements and the organization of code in files is not relevant for the exam.

The exam is work demanding and we do not expect all to complete all questions. Try to be strategic and select the questions that are suitable for your level and your ambitions. The more time you spend browsing the book, the less time you have to find solutions. The various parts of an assignment is organized in a logical sequence, but the sequence does **not** imply an increase in difficulty.

The main parts count with the percentages indicated, but we reserve the right to change the weighting depending on how the results. The separate subparts may also be weighted differently.

Assignment 1: Implementing a class (45%)

Rational numbers are numbers that can be written as a fraction with integer numerator and denominator. The class below is the start of a class for rational numbers. The tasks in this assignment are to implement member functions and operators and answer some theoretical question.

```
class Rational{
private:
    int n;                   //numerator (norsk: teller)
    int d;                   //denominator (norsk: nevner)
    void reduce();           //reduce the fraction (norsk: forkorte brøken)
    int gcd(int a, int b);  //finds the greatest common divisor
public:
    Rational();              //creates a value of 0/1
    Rational(int n, int d); //creates a value of n/d
    int numerator();         //returns numerator
    int denominator();       //returns denominator
    double to_double();      //returns fraction as double value
};

int Rational::gcd(int a, int b){
    //Euclids Algorithm - finds the greatest common divisor for a and b
    if ( b == 0 ){
        return a;
    }
    return gcd(b,a%b);
}
```

a) Implement the member function **`reduce()`**. It will be used to reduce/simplify the **`Rational`**-instance (fraction) it is called in context of. It is a helper function that e.g. is to be used in the constructor or operators that modifies the state of an object. Calling **`reduce()`** should result in the following:

1) The fraction is reduce with the assistance of the **`gcd()`**-function.
   gcd is an implementation of Euclid's algorithm that will give you the greatest common divisor for to numbers. Given an object having **`n`** *= 2 and* **`d`** *= 4 we can use gcd to find the greatest common divisor that then can be used to reduce both members giving the result* **`n`** *= 1 and* **`d`** *= 2.*
2) The function will make sure we always express the sign of the fraction using the nominator. We want the nominator to be positive or negative, but the denominator should always be a positive number - following ordinary rules for fractions.
   *Given (***`n`** *= 2,* **`d`** *= -3), reduce() should change this to (***`n`** *= -2,* **`d`** *= 3).*
   *Given (***`n`** *= -2,* **`d`** *= -2), reduce() should change this to (***`n`** *= 2,* **`d`** *= 3).*

b) Implement the constructor **`Rational(int n, int d)`**. **`reduce()`** is of course intended to be used. *It is the responsibility of the programmer that uses this class to prevent 0 as denominator.*

c) Implement the member function **`double to_double()`**. The funtion returns the fraction as a decimal number of type double. For **`Rational a(1,2)`** a call to **`a.to_double()`** will return **`0.5`**.

d) Implement the "*less than*"-operator as <u>member of the class</u>.

e) Show how to use the "*less than*"-opertor in the implementation of the "*is equal*"-operator.

f) Implement the <u>postfix</u> increment operator as member of the class.
Incrementing increases the fraction with the value of `1`*. Given an object* **`Rational a`** *with the value of* `1/3` *(which means* `n = 1` *and* `d = 3`*), then the statement* **`a++;`** *should yield* `4/3` *(*`n = 4` *and* `d = 3`*) because we are adding* `3/3` *which is the same as* `1`*. Remember that the return value of postfix increment is the value of the object before it is incremented.*

g) Implement as members the arithmetical opertors that are needed to support the following statement: `a *= b * c;` Where `a`, `b` and `c` all are `Rational` objects.

h) Is it necessary to implement the assignment operator and/or the copy constructor for this class? Explain your answer.

i) In C++ there are different integer types such as `short`, `int`, `long` and they can be `signed` or `unsigned`. There are also different decimal number types such as `float` and `double`.

1) Declare `Rational` as a template-class where the type used for numerator and denominator is declared by the user of the class.
2) Do you see any pitfalls in the declaration/use of such a class? Explain your answer

j) How do you have to declare `gdc()` if you still want it as a part of the class, but want to make use of the function without an instance of Rational?
We want to be able to write: `int dvs = Rational::gcd(8, 10);`

k) The constructor `Rational(int n, int d)` could have been implemented to throw and exception of type `invalid_argument` if the denominator was initialized with `0`. Would this have been a sufficient and practical solution to prevent integer division by zero? The context for your answer should be a more complete implementation of `Rational` including all arithmetical operators. Explain your answer.

l) Given that you want to support the following assignment:

```
Rational r;
r.denominator() = 5;
```

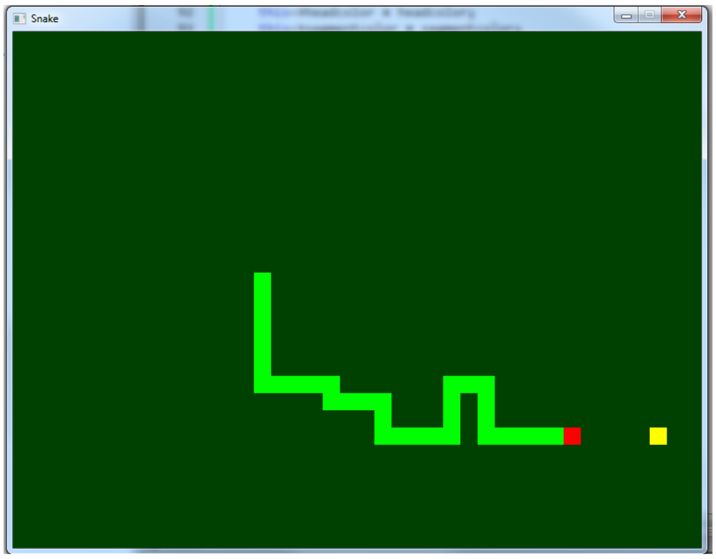
How can this be achieved simply by changing the declaration of this function?

Note! This is meant as a theoretical question and is not very realistic.

Assignment 2: The Snake game (45%)

In this assignment you will implement parts of a game that commonly is called **Snake**. In this game, the user controls with the arrow keys a snake that moves around on the screen and eats pieces of food that pops up at random places. Whenever a piece is eaten the length of the snake increases. The game is over if the snake hits the borders of the window or if the snake collides with itself.

The image shows how the game may look like. The yellow rectangle in the window is the piece the snake is hunting. The head of the snake is shown in red and the rest of the snake body is light green. When the snake changes direction the rest of the body will follow the same path leading to many turns in the body. As the length of the snake increases it will be harder to avoid hitting the border or preventing that the snake collides with itself.



The challenge in this assignment is to understand a somewhat complex problem (given the time limits) and to design and write code that is essential for the program. The code is partly based on the SFML-library for games and multimedia, but you do not need to know this library in advance to understand and do the assignment - and we have simplified the code.

Examples and explanations can be found in the appendix. Start by figuring out how `main()` works.

The full declaration of the functions mentioned in the assignment can be found in the appendix.

a) Implement the function **`initSnake()`**.
This function initializes the snake with 4 rectangles placed side by side approximately in the middle of the window.
A snake is actually only a collection of rectangles. In the example code we implement the snake using vector<Rectangle>. The class Rectangle has a function setPosition() that is used to place rectangles in the window and get-functions to read the position. Usage examples are found in the appendix.

b) Learn from examples in the appendix how the snake movement is to be implemented.

1) The library type **`vector<>`** is actually not very suitable for our snake implementation. Why?
2) What other container type in the library should you rather use?

There is no need to change 2a) even if you suggest a different type.

c) Implement the function **`moveSnake()`**. Calling this function once should move the snake one step in the direction given by the parameter. Assume that direction always is valid.
You do not have to consider the case where the snake collides with the border of the window and you are free to exchange vector with whatever type you suggested in 2b.

d) Implement the function **`collisionTest()`** that is used to detect "collision" between rectangles. Collision means in practise that two rectangles overlap.
All rectangles in this game are x- and y-aligned and we can use the straight-forward and simple testing suggested in the appendix.

e) Implement the function **`placePiece()`**. The **`Rectangle`**-variable **`piece`** is the piece the snake is hunting. This function will update **`piece`** with a random position within the window, but should also make sure that the piece is not placed on top of the snake.

f) Show and explain how and where in **`main`** you will use **`collisionTest()`** to check if the snake has "eaten" the piece. If the piece is eaten it should be updated with a new position.
In practise this means that you check if the head of the snake overlaps with the piece-rectangle. What we refer to as the head of the snake will be the rectangle in the front, but what the front is will depend on the order you are storing rectangles. In the game it will look like a piece is eaten and a new one appears in a different place. Internally we are using a single variable for the piece.

g) Whenever a piece is eaten the length of the snake should increaset with one rectangle. Show and explain how to implement this..
Tip: This can be solved by changing one of the previous functions.The length is to be increased but is do you really need to add a new rectangle at the end?

h) Create a new function **`selfCollide()`** to test if the snake collides with itself. This happens if the head collides with other rectangles in the snake body. Take care not to test if the head overlaps with itself. It is up to you to decide return value and list of parameters.

i) **A more open assignment**
The code in the appendix contains enough information to figure out the public functions of the **`Rectangle`** class. I this part the task is to create a subclass of **`Rectangle`** that we have called **`MovingRectangle`**. This is a more specialized class and objects of this type have a direction and a speed and will e.g. change direction if they hit the border. We will use this type for the pieces the snake is hunting and it gives pieces that move around on screen which means a more exciting hunt.

Requirements:

1) The class inherits from **`Rectangle`**.
2) Create a complete declaration of **`MovingRectangle`** in your answer and write an implementation of the **`constructor`** as well as implementiations of the two functions **`move()`** og **`place()`**. Return type and list of parameters are decided by you.
3) The class needs a constructor that initializes **`MovingRectangle`**-objects with a random position within a window, as well as sets an initial speed and direction.
4) The member function **`place()`** is used to move the rectangle to a different random position and give a new speed and direction. This function will be used by the constructor and whenever the piece is eaten by a snake. The function will be comparable to **`placePiece()`** from assignment 2e), but you do not have to take care of potential collisions with the snake.
5) The member function **`move()`** is used to move the rectangle one step in the direction that is set for the object. Given the instance **`MovingRectangle piece,`** the main idea is to make a call to **`piece.move()`** in every iteration of the game-loop in main and the visual impression will be a constant movement in this direction. If an object hits the border of the window it should change direction and move away from the border.
6) Write code that demonstrates the techniques and principles of your solution, but it should be easy to generalize and extend the code to a more complete solution.
7) Examples on how to control a movement can be found in the appendix.

Note! The assignment can be solved with a reasonable number of code lines.

Assignment 3: If there is still time left (10%)

The constructor in **TwoDArrayImpl** allocates memory for the rows in a two diemnsional array and the destructor takes care of deleting this memory. **TwoDArrayImpl** is a subclass of **TwoDArrayBase**.

```
class TwoDArrayBase {
public:
    TwoDArrayBase();
    ~TwoDArrayBase();
};

class TwoDArrayImpl : public TwoDArrayBase{
//Implemenentation of run-time allocated two-dimensional array
//using pointer to pointer technique
private:
    double **arr;
    int rows, columns;

public:
    TwoDArrayImpl( int rows, int columns) : arr(nullptr), rows(rows), columns(columns){
        //all pointers are initialized as nullptr during allocation
        arr = new double*[rows]{};
        for ( int i = 0; i < rows; i++ ) {
            arr[i] = new double[columns]{};
        }
    }
    ~TwoDArrayImpl() {
        for (int i = 0; i < rows; i++) {
            delete [] arr[i];
        }
        delete [] arr;
    }
};

int main() {
    try{
        TwoDArrayImpl *testarr = new TwoDArrayImpl(100, 50000);
        delete testarr;
    }catch(bad_alloc &exc){
        cout << "Error allocating memory" << endl;
    }
}
```

a) There is a problem in the constructor of **TwoDArrayImpl**. In case **new** fails to allocate memory it will cast a **bad_alloc** exception. We are catching this exception in main. The problem occurs if you run out of memory during the allocation. The program will then jump directly to the catch-statement in **main** and memory will not be release because **delete testarr** will never be executed. **Show how** the constructor can be programmed to catch **bad_alloc** when allocation fails and release allocated memory before resending the exception (and catching it again in main).

b) **TwoDArrayImpl** is a subclass of **TwoDArrayBase**. When declaring the pointer **testarr** as a super-type, as shown in the code blow, you notice that the subtypes destructor never is called (independent of exception or not). What is the typical cause of this error?

```
int main() {
    try{
        TwoDArrayBase *testarr = new TwoDArrayImpl(100, 50000);
        delete testarr;
    }catch(std::bad_alloc &exc){
        cout << "Error allocating memory" << endl;
    }
}
```

Appendix

Snake main-function with game-loop

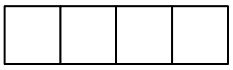
A Window-object represents the on screen window of the game and a while loop is used to create a “living” game. In every iteration the displayed image is wiped out and a new image is drawn. By drawing the same objects at slightly different positions you create the impression of movement.

```
const int RECT_WIDTH = 10;
const int RECT_HEIGHT = 10;
const int WIN_WIDTH = 1200;
const int WIN_HEIGHT = 800;
//We are using constants for size of rectangles and size of window
//We do not set any specific colors and rely on default
//which is black background and white shapes

enum Direction {UP, DOWN, LEFT, RIGHT};

//Functions explained in the assignments
void initSnake(vector<Rectangle> &snake);
void placePiece(Rectangle &piece, const vector<Rectangle> &snake);
void moveSnake(vector<Rectangle> &snake, Direction direction);
bool detectCollision(const Rectangle &a, const Rectangle &b);

//getDirection retrieves information from the keyboard and updates the direction
//variable according to pressed arrow key and valid change of direction
void getDirection(Window &window, Direction &direction);

int main(){
    srand(time(0));                 //Seeding rand() which is used by the functions
    Clock timer;                    //A timer is comparable to a stop watch
    Direction direction = RIGHT;    //Set an initial movement direction of the snake

    Window window(WIN_WIDTH, WIN_HEIGHT, "snake"); // GUI window

    vector<Rectangle> snake;                //Our snake is a collection of rectangles
    initSnake(snake);                       //initialized with 4 adjacent rectangles

    Rectangle piece;                        //The objekt that the snake is hunting
    piece.setSize(RECT_WIDTH, RECT_HEIGHT); //Setting size and position of piece
    placePiece(snake, piece);

    while (window.isOpen()){  //game loop that runs until window is closed

        getDirection(window, direction);     //Updates the direction

        window.clear();                      //Clears the screen

        if (timer.getElapsedMilliseconds() >= 100){ //Check if 0.1 ms has passed
            moveSnake(snake, direction);
            timer.restart();                        //Reset/restart the timer
        }

        for (Rectangle &s: snake){
            window.draw(s);         //Draw each rectangle of the snake on screen
        }
        window.draw(piece);         //Draw the piece rectangle

        window.display();           //Display the screen
    }
}
```

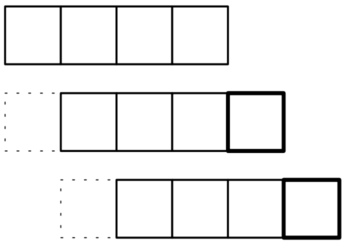
Placing rectangles next to each other

```
//Example code for drawing rectangles side by side along x-axis
//This code shows all Rectangle-functions you need.
//Coordinate (0, 0) is upper left corner of window.
//The position of a rectangle is its upper left corner.

const int size = 10;          // example rectangle size
Rectangle first;
Rectangle second;

first.setSize(size, size);    // set width and height
first.setPosition(250, 250); // set position of first rectangle on screen

second.setSize(size, size);
second.setPosition(first.getx() + size, first.gety());

window.draw(second);
window.draw(first);
```

Moving the snake

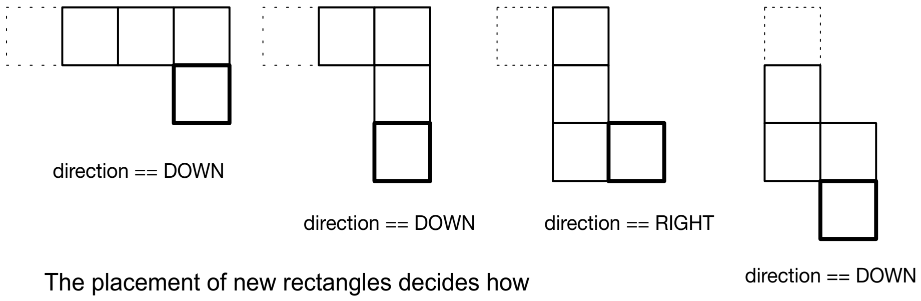


Start with 4 equivalently sized rectangles placed side by side



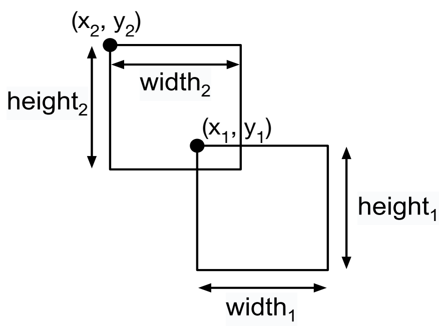
Create movement by adding one new rectangle (thick line) at one end and remove one rectangle (dotted line) from the other end.

In this example we create a movement towards the right. Note that the snake-movement is not smooth but stepwise. Every operation moves the snake in a step equal to the size of the rectangle.



The placement of new rectangles decides how the snake moves and turns on the screen.

Collision detection



The following statements are all true if the rectangles overlap:

$x_1 < x_2 + width_2$
$x_1 + width_1 > x_2$
$y_1 < y_2 + height_2$
$height_1 + y_1 > y_2$

Controlling the movement of an object

```
//Example code to incrementally change the position of an object
//Speed and direction can also be managed using a single variable for velocity
void movementexample(){
    int dx = -1, dy = 1;                //moving down and towards left
    double speedx = 0.2, speedy = 0.2;  //same speed along both axis
    double x = 500, y = 50;             //example starting coordinate
    int width = 1000, height = 1000;    //window size
    while(x > 0 && x < width && y > 0 && y < height){
        //game-loop to move the coordinate until it hits the edges
        x = x + (speedx * dx);
        y = y + (speedy * dy);
        cout << x << ", " << y << endl;
    }
}
```

Sequence containers in C++ standard library

Headers		`<array>`	`<vector>`	`<deque>`	`<forward_list>`	`<list>`
Members		**array**	**vector**	**deque**	**forward_list**	**list**
	constructor	*implicit*	vector	deque	forward_list	list
	destructor	*implicit*	~vector	~deque	~forward_list	~list
	`operator=`	*implicit*	operator=	operator=	operator=	operator=
iterators	`begin`	begin	begin	begin	begin before_begin	begin
	`end`	end	end	end	end	end
	`rbegin`	rbegin	rbegin	rbegin		rbegin
	`rend`	rend	rend	rend		rend
const iterators	`begin`	cbegin	cbegin	cbegin	cbegin cbefore_begin	cbegin
	`cend`	cend	cend	cend	cend	cend
	`crbegin`	crbegin	crbegin	crbegin		crbegin
	`crend`	crend	crend	crend		crend
capacity	`size`	size	size	size		size
	`max_size`	max_size	max_size	max_size	max_size	max_size
	`empty`	empty	empty	empty	empty	empty
	`resize`		resize	resize	resize	resize
	`shrink_to_fit`		shrink_to_fit	shrink_to_fit		
	`capacity`		capacity			
	`reserve`		reserve			
element access	`front`	front	front	front	front	front
	`back`	back	back	back		back
	`operator[]`	operator[]	operator[]	operator[]		
	`at`	at	at	at		
modifiers	`assign`		assign	assign	assign	assign
	`emplace`		emplace	emplace	emplace_after	emplace
	`insert`		insert	insert	insert_after	insert
	`erase`		erase	erase	erase_after	erase
	`emplace_back`		emplace_back	emplace_back		emplace_back
	`push_back`		push_back	push_back		push_back
	`pop_back`		pop_back	pop_back		pop_back
	`emplace_front`			emplace_front	emplace_front	emplace_front
	`push_front`			push_front	push_front	push_front
	`pop_front`			pop_front	pop_front	pop_front
	`clear`		clear	clear	clear	clear
	`swap`	swap	swap	swap	swap	swap
list operations	`splice`				splice_after	splice
	`remove`				remove	remove
	`remove_if`				remove_if	remove_if
	`unique`				unique	unique
	`merge`				merge	merge
	`sort`				sort	sort
	`reverse`				reverse	reverse