Solutions to this sheet are due on 24.11.2017 til 14:00. Please hand in a digital version of your answers via e-mail. The e-mails subject has to contain ccpp. Do zip-compress your solutions. For questions please send mail or speak to me during the exercises.

**Note:** If you copy text elements / code elements from other sources, clearly mark those elements and state the source. Copying solutions from other students is prohibited. All of your files that belong to your solution have to be contained in a single .zip file that is named according to the following naming scheme: `<name>_<surname>_solution<XX>.zip`. Replace `<name>` and `<surname>` with your actual name and replace `<XX>` with the sheet number the solutions belong to. You can look up your results using this link [https://docs.google.com/spreadsheets/d/1V8rKtimsQS6thKGkTh6CChlv-LwuIBIA3RvKA7ZSH2M/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1V8rKtimsQS6thKGkTh6CChlv-LwuIBIA3RvKA7ZSH2M/edit?usp=sharing)

During this exercise sheet you will deepen your knowledge about exceptions and operator overloading. Use and extend the code snipped from the lectures website. You can achieve 16 points in total.

**Exercise 1.**

Because of the finite representation of integer values in memory, an integer data type can only hold a limited range of integer values. If the range of that data type is exceeded using arithmetic operations, an integer over- or underflow is caused. A C++ program containing a signed integer over- or underflow has undefined behavior. Therefore, integer over- and underflows often lead to dangerous bugs. But these things can be detected. An integer overflow for a summation of two `int` values can be detected using the following code:

```cpp
#include <iostream>
#include <limits>
using namespace std;

int main() {
    int a = 100;
    int b = 200;
    // checking over− / underflow for addition
    if ((b > 0) && (a > numeric_limits<int>::max() − b)) {
        cout << "addition of a and b would overflow\n";
    } else if ((b < 0) && (a < numeric_limits<int>::min() − b)) {
        cout << "addition of a and b would underflow\n";
    } else {
        cout << "normal\n";
    }
}
```
Image you have to implement a software that is used to run a nuclear power plant. You cannot risk that computations using signed integer arithmetic produce wrong results due to over-/underflows or division by zero. For that reason, write a wrapper type for the build-in `int` type by performing the following tasks. Use the code provided by the lectures website and extend the code as necessary.

a) The above code shows how to detect an integer overflow for summation of two signed integer variables. Find out how to detect an integer underflow for `-`, `*`, `/` yourself. (You are of course allowed to use google as well. It is sufficient to use the knowledge gained here to correctly perform task c) you do not have to prove that you know how to detect these separately for task a) )

(3 P.)

b) Define your own new signed integer type called `sint` (safe int) using the keyword `class` which is robust against over- and underflows. Provide implementations for the following special member functions for our `sint` type. (Hint: Think about what special member functions can be set to default.)

- `sint();` // default ctor, that initializes with 0
- `sint(int i);` // ctor that initializes with value of i
- `~sint();` // dtor
- `sint(const sint & s);` // copy
- `sint& operator=(const sint & s);` // copy assign
- `sint(sint && s);` // move
- `sint& operator=(sint && s);` // move assign

(2 P.)

c) Overload the following operators such that the `sint` can be used pretty much like a build-in data type. All of the arithmetic operators (`+`, `++`, `--`, `*`, `/`) must check if an integer over- or underflow or a division by zero occurs during a calculation and throw a suitable exception (`overflow_error`, `underflow_error`, `logic_error`) in case an error occurs to notify the user of the `sint` type. Test your code using the commented code inside `main` (‘de-comment’ as necessary and catch exceptions as they occur).

- `friend sint operator+ (sint lhs, sint rhs);`
- `friend sint operator- (sint lhs, sint rhs);`
- `friend sint operator* (sint lhs, sint rhs);`
- `friend sint operator/ (sint lhs, sint rhs);`
- `friend sint operator/ (sint lhs, sint rhs);`
- `sint &operator++();` // prefix `++` no parameter, returns a reference
- `sint operator++(int);` // postfix `++` dummy parameter, returns a value
- `sint &operator--();` // prefix `--` no parameter, returns a reference
- `sint operator--(int);` // postfix `--` dummy parameter, returns a value
- `friend ostream& operator<< (ostream& os, const sint& s);`

(6 P.)
d) It is probably a good idea to provide an additional member function that returns the integer stored internally in that class:

- `int getInt() noexcept;`

(1 P.)

e) What is the size (in bytes) of a variable of type `sint` on your machine? In general, what is the size of a user defined type? How and where does the compiler store member functions? (2 P.)

f) You know what is coming next. Do as you would do as a professional software developer and split your `sint` code into header and implementation file! (2 P.)