C++ Programming

Exercise Sheet 5
Software Engineering Group EIM-I
Philipp Schubert
May 31, 2019

Solutions to this sheet are due on 07.06.2019 til 14:00. Please hand in a digital version of your answers via e-mail. The e-mail’s subject has to contain cppp19. Do zip-compress your solutions. 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_<X>.zip. Replace <name> and <surname> with your actual name and replace <X> with the number of the exercise sheet. You can look up your results using this link [https://docs.google.com/spreadsheets/d/1Ve7yD5uxTDjXbZJhW9e5QupvffLeC20j61gP2P4ouJla/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1Ve7yD5uxTDjXbZJhW9e5QupvffLeC20j61gP2P4ouJla/edit?usp=sharing).

This exercise sheet will help you to deepen your knowledge on exceptions and operator overloading. Use and extend the code snipped from [https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Softwaretechnik/Lehre/CPP_Programming/SS2019/code_05.zip](https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Softwaretechnik/Lehre/CPP_Programming/SS2019/code_05.zip).

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, an integer over- or underflow is caused. A C++ program has undefined behavior if a signed integer over- or underflow occurs at runtime. Therefore, integer over- and underflows often lead to dangerous bugs. However, they 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";
    }
    return 0;
}
```
Rather than performing hand-crafted checks, you can also use compiler built-ins. Depending on the C++ compiler that you are using, please check:

- [http://clang.llvm.org/docs/LanguageExtensions.html#checked-arithmetic-builtins](http://clang.llvm.org/docs/LanguageExtensions.html#checked-arithmetic-builtins)

Imagine 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 built-in `int` type by performing the following tasks. Use the code provided on the lecture’s website and extend it 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 may wish to use the compiler built-ins to detect over- / underflow rather than crafting your own checks. 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).

(0 P.)

b) Define your own signed integer type called `sint` (safe int) that is robust against over- and underflows using the keyword `class`. 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
- `int getValue() noexcept;`

(2 P.)
c) Overload the following operators such that the sint can be used pretty much like a built-in integer 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 in 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);
- 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);

(5 P.)

d) 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? (3 P.)

Exercise 2.
With your current knowledge on pointers you are now able to implement your own advanced data structures such as lists. In this exercise, you need to implement a rudimentary version of a singly linked list. (This question is likely to be asked during a job interview when applying for a job in software development.)

Consider the following code:

```c++
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node *next;
    Node(int i) : data(i), next(nullptr) {}  
    friend ostream &operator<<(ostream &os, const Node &n) {
        os << "Node n" << ":" << n.data << "\nthis: " << &n << "\nnext: " << n.next;
        return os;
    }
};

void addElement(Node **head, int data);
void printList(const Node *head);
void deleteList(Node *head);

int main() {
    Node *list = nullptr;
    addElement(&list, 1);
```
addElement(&list, 2);
addElement(&list, 3);
addElement(&list, 4);
printList(list);
deleteList(list);
return 0;
}

Provide the missing implementations for **addElement**, **printList** and **deleteList**! (6 P.)

- **addElement** must allocate and initialize a new node and add it to the end of the list pointed to by **head**. The end of the list is denoted with the **nullptr**.

- **printList** needs to iterate the list pointed to by **head** and print each node of the list. Use the **operator<<** that has already been overloaded to print a list node of type **Node**.

- **deleteList** must be implemented to deallocate all dynamically allocated nodes maintained in the list.

(Hint: Since you are dealing with pointers it might be helpful to draw the structure of the list on a piece of paper and think about what each function has to do. Use while loops to iterate a list; check for the **nullptr** to determine the end of a list.)

You can check your list implementation for memory issues by compiling the code using the flag `-g` and then, calling **valgrind**, e.g.:

```
$ g++ -std=c++14 -Wall -Wextra -g list.cpp -o list
$ valgrind --leak-check=full --track-origins=yes ./list
```

The **valgrind** tool can be installed using $ sudo apt-get install valgrind