C++ Programming

Exercise Sheet 5

Secure Software Engineering Group
Philipp Schubert
philipp.schubert@upb.de

May 21, 2021

Solutions to this sheet are due on 28.05.2021 at 16:00. Please hand-in a digital version of your answers via PANDA at https://panda.uni-paderborn.de/course/view.php?id=22691.
Note: If you copy text or code elements from other sources, clearly mark those elements and state the source. Copying solutions from other students is prohibited.

In the first exercise, you will deepen your knowledge on exceptions and operator overloading. The second exercise is a typical coding interview question (like many of the exercises of this course). Use and extend the code snippets provided at https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Softwaretechnik/Lehre/CPP_Programming/SS2021/code_05.zip. You can achieve 16 points in total.

Exercise 1.
Because of the finite representation of integer values in memory, an integer type can only hold a limited range of integer values. If the range of the given type is exceeded an integer over- or underflow is caused. *Signed* integer over- or underflows lead to undefined behavior. Therefore, such over- or underflows often lead to dangerous bugs. An integer overflow for an addition of two `int` variables can be detected using the following code:

```
#include <iostream>
#include <limits>

int main() {
  int a = 100;
  int b = 200;
  // checking over- / underflow for addition
  if ((b > 0) && (a > std::numeric_limits<int>::max() - b)) {
    std::cout << "addition of a and b would overflow\n";
  } else if ((b < 0) && (a < std::numeric_limits<int>::min() - b)) {
    std::cout << "addition of a and b would underflow\n";
  } else {
    std::cout << "everything is fine\n";
  }
  return 0;
}
```

Rather than using hand-crafted checks, you should use the respective compiler built-ins. Depending on the C++ compiler you are using, please check out:

- `http://clang.llvm.org/docs/LanguageExtensions.html#checked-arithmetic-builtins`
- `https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html`

Imagine you have to implement some critical software. You cannot risk that computations using signed integer arithmetic produce wrong results due to over- or underflows, or division by zero. For that reason you need to write a wrapper type for the built-in **int** data type by performing the following tasks. Use the code provided on the lecture's website and extend it as necessary.

a) Check out how to detect an integer over- and underflow for **+**, **-**, *****, **/** using the compiler built-ins. (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: (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 getUnderlyingValue() const;**

(2 P.)

c) Overload the following operators such that the **sint** type can be used 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**) to alert 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.

Consider the following code:

```
#include <iostream>

struct Node {
  int data;
  Node *next;
  Node(int i) : data(i), next(nullptr) {}
  friend std::ostream &operator<<(std::ostream &os, const Node &n) {
    os << "Node\n"
       << "\tdata: " << n.data << "\n\tthis: " << &n
       << "\n\tnext: " << 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** must 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 may wish to also consult the following video: `https://www.youtube.com/watch?v=t5NszbIerYc`.)

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

```
$ clang++ -std=c++17 -Wall -Wextra -g list.cpp -o list
$ valgrind --leak-check=full --track-origins=yes ./list
```

The valgrind tool can be installed using `$ sudo apt install valgrind`

Alternatively, you can also use Clang's address and undefined behavior sanitizer to ensure that your program does not contain memory issues, e.g.:

```
$ clang++ -std=c++17 -Wall -Wextra -g -fsanitize=address,undefined
-fno-omit-frame-pointer list.cpp -o list
$ ./list
```