LABORATORY 9
Exceptions. Generic programming. Streams

Objectives

The purpose of the laboratory is to highlight some more advanced C++ features: exceptions, generic programming and streams. In the beginning, the laboratory work covers signaling and handling of errors using exceptions. Next, it deals with generic programming by showing some basic generic functions and classes. Finally, C++ streams are presented, with examples on how they can be used for input-output operations.

Theoretical concepts

Exceptions in C++

The code below gives the declaration of a class called *Array100* which represents a fixed-size array with 100 elements. The function *int& at(int pos);* returns a reference to the element stored at position *pos* in the array. However, there is an error in the implementation of this function: if the method is invoked with a negative number or a number greater or equal to 100 as the *pos* parameter, the behavior is undefined. Moreover, one can notice that the function *at* cannot itself solve the problem.

```
class Array100{
public:
	// … other methods
	int& at(int pos){return data[pos];}
private:
	int data[100];
};
```

There are three common ways of handling errors.
The first one involves using return codes/parameters. In this case, the function returns a variable that indicates whether the operation was successful. For example, the *scanf* function returns how many elements it actually managed to read from the standard input. The main drawbacks of this mechanism are: (1)

the programmer will often ignore the return value of the function and thus won't be aware of the status of the operation, (2) if the function actually needs to return a value (for example, a function that computes the sqrt of a number) the code will get more complicated (the error code must be passed as an input/output parameter).
Another approach to handling errors is by setting some error flags/variables that indicate the status of the operation. For example, C++ streams have several flags defined in the base class *ios_base* (*badbit*, *goodbit*, *eofbit*, *failbit*). As for the previous mechanism, it is common to forget to check these variables after doing an operation on the stream.
Handling errors by either return code or error flags also makes the code more convoluted and harder to read, because after each operation one needs to use a conditional statement to check the result of an operation.
Exceptions are a more elegant way of handling errors. An exception is a program response to an anomalous/exceptional condition that might occur during the program's execution (in the previous example, if we try to get the element stored on the position -1 in the array). Exceptions allow for an elegant way to transfer control from the part of code in which the exception occurred to the part of code that can properly handle the exception.
Three main components are used when working with exceptions in C++: *throw*, *try*, and *catch*.

Signaling an exception

In C++, signaling an exceptional conditional is achieved by means of the *throw* keyword. This process is called raising an exception, and the normal flow of execution will be interrupted until a handler for this exception is found.
To raise an exception, the *throw* keyword is employed, followed by a value of any datatype: an error code (int or char), a description of the error (a string), or any custom object.
The following examples show how to throw an exception of different types:

- For example (a) an exception of type of *int* is thrown: *throw 1;*
- For example (b) a string exception is thrown: *throw "Oops! Invalid index!";*
- For example (c) a custom object is thrown (the class *MyException* is defined someplace else): *throw MyException("Invalid index").*

(a)

```
int& Array100::at(int pos){
if(pos < 0 || pos >=100)
    throw 1; // throwing an exception of type int
return data[pos];
}
```

(b)

```
int& Array100::at(int pos){
if(pos < 0 || pos >=100)
    throw “Oops! Invalid index!”; // throwing a string as an
exception
return data[pos];
}
```

(c)

```
int& Array100::at(int pos){
if(pos < 0 || pos >=100)
    throw MyException(“Invalid index”); // throwing a custom
object as an exception (the class MyException is defined
someplace else)
return data[pos];
}
```

Checking for exceptions

To check that an exception has occurred in any part of the code, the *try* block must be used. This acts like an observer of the code and checks whether there are any exceptions thrown in the code written within the try block. It works in conjunction with the *catch* block (presented below).

Handling exceptions

Finally, the handling of an exception is performed in the *catch* block. A *catch* block can handle an exception of a single data type (passed as a parameter to the catch block).

Multiple catch blocks can be specified for a *try* block to catch different types of exceptions. One thing to keep in mind, is that no implicit casting is performed for primitive data types!

The code below modified the *Array100* class to throw an integer exception when the user attempts to access an element outside the valid range. The *at* method throws an exception of -1 if the index is negative and -2 if the index is greater than or equal to 100.

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

class Array100{
public:
    // … other methods
    int& at(int pos) {
        if (pos < 0) {
            throw -1; // Throw an integer exception for
negative index
        }
        if (pos >= 100) {
            throw -2; // Throw a different integer exception
for out-of-bounds index
        }
        return data[pos];
    }
private:
    int data[100];
    unsigned int capacity;
};

int main() {
    Array100 myArray;
    try {
        // Attempt to access an element with an invalid
index
        int value = myArray.at(150);
    }
    catch (float e) {
        cout << "Float exception caught." << endl;
    }
    catch (double e) {
        cout << "Double exception caught." << endl;
    }
    catch (int e) {
        if (e == -1) {
```

```
            cout << "Error: Negative index provided." <<
endl;
        } else if (e == -2) {
            cout << "Error: Index out of bounds." << endl;
        }
    }
    return 0;
}
```

```
Error: Index out of bounds.
```

As mentioned above, in the case of *catch* clauses, no implicit casting is performed. The *try-catch* block in the main function attempts to catch exceptions of types float, double, and int. However, because the *at* method only throws int exceptions and no casting is performed , only the *catch (int e)* block will actually catch and handle the exceptions thrown by the function *at*. The code above will display: "Error: Index out of bounds.".

Exceptions and derived classes

Although for *catch* blocks implicit casting between primitive types is not performed, upcasting (casting between a derived class object to a base class object) is performed implicitly in catch blocks. In other words, when performing exception handling for hierarchy of classes, because of inheritance and polymorphism, if an exception of a derived class exception is thrown, it can be caught by a catch block written for one of its base classes. This is a form of implicit upcasting, where the derived class object is treated as an object of its base class.
Because of this, catching exceptions by reference, especially when working with exception class hierarchies, is the recommended practice in C++, because:

- It avoids object slicing. Object slicing happens when a derived class object is assigned to a base class object, leading to the loss of the derived part of the object. If a derived exception object is caught by value, object slicing will occur, stripping away any derived class information and behaviors;
- It avoids unnecessary copying of exception objects. By catching exceptions by reference a more efficient exception handling is achieved, as the overhead associated with making a copy is avoided;

- It enables polymorphic behavior. In C++ polymorphism only works with pointers or references to the base class, and therefore catching exceptions by reference ensures polymorphic behavior.

The example below shows an example of catching a derived class exception object with a *catch* clause that takes as parameter a constant reference to a base class. This allows the *what()* method of the *DerivedException* to be called correctly, demonstrating polymorphic behavior. If the exception were caught by value, only the *BaseException* part of the object would be caught, it would have called the *BaseException*'s what() method instead.

```
#include <iostream>

class BaseException {
public:
    virtual const char* what() const {
        return "Base exception occurred";
    }
};

class DerivedException : public BaseException {
public:
    const char* what() const noexcept {
        return "Derived exception occurred";
    }
};

void oops() {
    throw DerivedException();
}

int main() {
    try {
        oops();
    } catch (const BaseException& e) {
        std::cout << e.what() << std::endl;  // This will
correctly call DerivedException's what() method
    }

    return 0;
}
```

```
Derived exception occurred
```

Rethrowing exceptions

In a catch block, an exception can be rethrown by using the *throw* keyword without any specified parameter: *"throw;"*.

```
Array100 arr;
try{
   int &i = arr.at(-1);
}catch(int e){
    cout<<"Exception of type int caught!"<<endl;
    throw; // the exception is rethrown
}
```

In the example above, the invalid access operation *arr(-1)* throws an exception of type *int*, the *catch(int e)* block catches it. Inside the catch block, the message *"Exception of type int caught!\n"* is displayed to the standard output. Next, the code uses the *throw* statement to rethrow the exception. Rethrowing exceptions is useful when the current scope cannot fully handle the exception, and it wants to pass the exception up to a higher scope for further handling.

Catch-all handler

C++ also defines a catch-all handler, which is a catch block that catches all exceptions, regardless of their type. The catch-all handler is denoted by an ellipsis "..." as the exception type, as illustrated below:

```
#include <iostream>
int main(){
  try {
        throw 1.0;
        std::cout << "This never prints\n";
    }
    catch (int& e) {
       // this is not a suitable handler for 1.0 (double
type)
        std::cout << "Caught integer exception: " << e <<
'\n';
    } catch(...){
     // the catch-all handler will be executed
       std::cout<<"Catch any type of exception.\n";
```

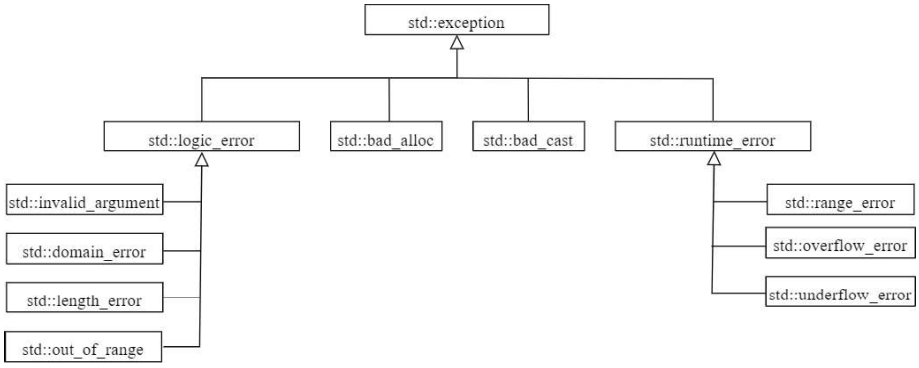
```
    }
    return 0;
}
```

```
Catch any type of exception.
```

Inside the try block, the program throws an exception with the value 1.0, which is of type double. Therefore, the line *std::cout << "This never prints\n";* won't get executed. The first catch block is designed to catch exceptions of type *int* by reference (*catch (int& e)*). However, the exception thrown is of type *double*, not *int*. Therefore, this catch block does not match the type of the thrown exception (because no implicit casting is performed) and it is skipped. The next catch block is a *catch-all* handler (*catch(...)*). This handler is designed to catch any type of exception that was not caught by the preceding specific *catch* blocks. Since the exception of type double was not caught by the earlier *catch* block for *int*, the *catch-all* handler is executed. This prints "*Catch any type of exception.\n*" .

STL exceptions

The C++ Standard Template Library (STL) provides a set of standard exceptions that can be used to report common errors. These exceptions are defined within the <*stdexcept*> header file. They inherit from the *std::exception* class, which is the base class for all C++ exceptions. The *std::exception* class contains a virtual member function *what()* that returns a C-style character string describing the current error, and this method can be overridden by its subclasses. The class hierarchy of the most commonly used exceptions from STL is depicted in the image below, and their short description is given in the next table.



Class hierarchy for the most used exception classes declared in C++

Exception class	Description
std::logic_error	Represents errors in the internal logic of the program.
std::invalid_argument	Indicates that an invalid argument has been passed to a function.
std::domain_error	Indicates that the inputs are outside of the domain on which an operation is defined.
std::length_error	Indicates a length error that results from attempts to exceed the defined length limits for an object.
std::out_of_range	Indicates attempts to access elements out of a defined range.
std::runtime_error	Represents errors that occur during the execution of the program.
range_error	Indicates range errors (the result of a computation cannot be represented by the destination type).
overflow_error	Reports arithmetic overflow errors.
underflow_error	Reports arithmetic underflow errors.
std::bad_alloc	Reports failure of memory allocation requests.
std::bad_cast	Thrown when a *dynamic_cast* to a reference type fails.

The example below demonstrates exception handling for out-of-bounds access on a std::vector. The *at* method of a *std::vector* throws a *std::out_of_range* exception if the index passed as parameter is out of the array bounds.

```
#include <iostream>
#include <vector>
#include <stdexcept>
int main() {
    std::vector<int> myVector = {1, 2, 3};

    try {
        // Attempt to access an element out of the bounds of
the vector
        int value = myVector.at(10); // This will throw a
std::out_of_range exception
```

```
    } catch (const std::out_of_range& e) {
                std::cout << "Caught an out_of_range
exception: " << e.what() << std::endl;
    }

    return 0;
}
```

```
Caught an out_of_range exception: invalid vector subscript
```

The call to *myVector.at(10)* is invalid because there is no element at index 10, and it throws an exception. The exception is then caught by the *catch* block which handles *std::out_of_range* exceptions from the main function. Within the catch block, an error message is printed to the standard output, indicating that an out-of-range exception was caught (*e.what()*).
It is not a common practice to directly throw *std::exception*, so its subclasses should be used. Another option is to write classes which inherit from *std::exception* and override the *what()* member function to provide a more detailed error message.

Stack unwinding

The main advantage of exceptions is their ability to defer error handling until a suitable catch block is found. This delegation of responsibility is the core purpose of exceptions. When an exception is thrown in C++, the runtime searches for a *catch* block capable of handling the exception. If no such block is found, the runtime initiates the process of stack unwinding. This process involves traversing back through the call stack and deallocating memory along the way, until a *catch* block for the type of the thrown exception (or a subtype of the exception class) is reached. If no suitable catch block is found, the program is terminated with a *std::terminate*.

The code below illustrates the process of stack unwinding. When used within the body of a function, the *__FUNCTION__* macro will be replaced with the name of that function when the code is compiled.

```
#include <iostream>

void stackUnwindingf4() // called by f3()
{
    std::cout << "Start "<<__FUNCTION__<<"\n";
    std::cout <<__FUNCTION__ <<" throwing exception\n";
    throw -1;
    std::cout << "End "<<__FUNCTION__<<"\n";
}

void stackUnwindingf3() // called by f2()
{
    std::cout << "Start "<<__FUNCTION__<<"\n";
    stackUnwindingf4();
    std::cout << "End "<<__FUNCTION__<<"\n";
}

void stackUnwindingf2() // called by f1()
{
    std::cout << "Start "<<__FUNCTION__<<"\n";
    try
    {
        stackUnwindingf3();
    }
    catch(float)
    {
        std::cerr << __FUNCTION__ <<" caught float
exception\n";
    }
    std::cout << "End "<<__FUNCTION__<<"\n";
}

void stackUnwindingf1() // called by main()
{
    std::cout << "Start "<<__FUNCTION__<<"\n";
    try
    {
        stackUnwindingf2();
    }
    catch (double)
    {
        std::cerr << __FUNCTION__ <<" caught double
exception\n";
    }
    catch (int)
    {
        std::cerr << __FUNCTION__ <<" caught int
```

```
exception\n";
    }
    std::cout << "End "<<__FUNCTION__<<"\n";
}
int main(){
    stackUnwindingf1();
}
```

```
Start stackUnwindingf1
Start stackUnwindingf2
Start stackUnwindingf3
Start stackUnwindingf4
stackUnwindingf4 throwing exception
stackUnwindingf1 caught int exception
End stackUnwindingf1
```

Below is a step-by-step explanation of what happens when the *main* function calls *stackUnwindingf1*:

- *stackUnwindingf1* is called by main. It prints "Start stackUnwindingf1" to indicate the start of the function.
- Inside *stackUnwindingf1*, in the try block the function *stackUnwindingf2* is called.
- *stackUnwindingf2* prints "Start stackUnwindingf2" and then calls the function *stackUnwindingf3* within its own try block.
- *stackUnwindingf3* prints "Start stackUnwindingf3" and calls the function *stackUnwindingf4*.
- *stackUnwindingf4* prints "Start stackUnwindingf4", followed by "stackUnwindingf4 throwing exception" before throwing an integer exception with *throw -1;*. The line "End stackUnwindingf4" is never reached or printed because the function exits when the exception is thrown.
- The exception thrown by stackUnwindingf4 is not caught within *stackUnwindingf3*function because there is no try-catch block in the function *stackUnwindingf3*. Therefore, *stackUnwindingf3* also doesn't reach its "End" print statement and the control returns to the try block in *stackUnwindingf2*.
- In *stackUnwindingf2*, the catch block is designed to catch exceptions of type float, not int. Therefore, it cannot catch the exception thrown by *stackUnwindingf4*, and the control is passed back to the *stackUnwindingf1* function.

- In *stackUnwindingf1*, there are two catch blocks: one for int and another for double. The exception thrown is of type int, so it is caught by the first catch block. "stackUnwindingf1 caught int exception" is printed.
- Finally, "End stackUnwindingf1" is printed as the last line of the function *stackUnwindingf1*, after exception handling.

Generic programming in C++

Templates in C++ allow for generic programming and represent a form of compile-time polymorphism. They enable the writing of code that can operate with generic data types, without having to write multiple versions of the code for each data type. Templates can be applied to both functions and classes.
To write a template function, a template parameter declaration must precede the function declaration, indicated by *template <typename T>* or by *template <class T>*. Then the placeholder datatype *T* can be used where needed, instead of an actual data type.
The example below shows a function that uses templates to compute the maximum value between two elements. Unlike a normal function, in which all the datatypes must be explicitly specified, template functions use one or more placeholder types. The template function will be used to generate overloaded functions based on the actual specified data types.

```
#include<iostream>
template <typename T>
T max(T a, T b) {
  return a > b ? a : b;
}

int main(){
  int mi = max<int>(20, 30);
  std::cout<<"maximum int is "<<mi<<std::endl;
  std::string ms = max<std::string>("Alice", "Bob");
  std::cout<<"maximum string is "<<ms<<std::endl;
  return 0;
}
```

```
maximum int is 30
maximum string is Bob
```

The generic function is called *max*. The line *template <typename T>* declares a template with a type parameter *T*, which acts as a placeholder for any data type. Then, actual implementation of the function *max* is given: the function takes

two parameters (*a* and *b*) of the same type *T* and returns a value of type *T* (which will be the maximum between *a* and *b*). The actual data type of *T* will be determined based on the argument types provided when the function is called. The *main* function shows how to use the *max* template function with different types of arguments:

- Integer types: *int mi = max<int>(20, 30)*
- String types: *string ms = max<string>("Alice", "Bob")*

The next example presents a class template for a stack data structure that can store any data type. A stack is a data structure that follows the *Last In, First Out* (LIFO) principle.

```
#include <vector>
#include <iostream>
#include <exception>

template <typename T>
class Stack {
public:
    // Add an element to the top of the stack
    void push(T item) {
        items.push_back(item);
    }

    // Remove and return the top element from the stack
    T pop() {
        if (isEmpty()) {
            throw std::out_of_range("Stack<>::pop(): empty
stack");
        }
        T el = items.back();
        items.pop_back();
        return el;
    }

    // Return the top element without removing it
    T peek() const {
        if (isEmpty()) {
            throw std::out_of_range("Stack<>::peek(): empty
stack");
        }
        return items.back();
    }
```

```
    // Check if the stack is empty
    bool isEmpty() const {
        return items.empty();
    }

private:
    std::vector<T> items; // Container for stack elements
};

// Example usage
int main() {
    // Stack of integers
    Stack<int> intStack;
    intStack.push(1);
    intStack.push(2);
    std::cout << "Top int: " << intStack.peek() <<
std::endl; // Should print "Top int: 2"
    std::cout << "Popped int: " << intStack.pop() <<
std::endl; // Should print "Popped int: 2"

    // Stack of strings
    Stack<std::string> stringStack;
    stringStack.push("Alice");
    stringStack.push("Bob");
    std::cout << "Top string: " << stringStack.peek() <<
std::endl; // Should print "Top string: Bob"
    std::cout << "Popped string: " << stringStack.pop() <<
std::endl; // Should print "Popped string: Bob"

    // Demonstrate exception handling by attempting to pop
from an empty stack
    try {
        intStack.pop(); // This will work, popping the
remaining element
        intStack.pop(); // This will throw, as the stack is
now empty
    } catch (const std::out_of_range& e) {
        std::cout << "Exception caught: " << e.what() <<
std::endl;
    }

    return 0;
}
```

```
Top int: 2
Popped int: 2
Top string: Bob
```

```
Popped string: Bob
Exception caught: Stack<>::pop(): empty stack
```

The part template `<typename T> class Stack` defines a template class called *Stack* with a type parameter *T*. This allows the stack to store elements of any data type, making it generic. The attribute *items* has the type `std::vector<T>` and it stores the elements of the stack. Since *T* is a placeholder for any data type, items can store elements of any type.
The stack class has the following methods:

- *push*, which adds an element to the top of the stack;
- *pop*, which removes and returns the top element from the stack. If the stack is empty, it throws a *std::out_of_range* exception;
- *peek*, which returns the top element of the stack without removing it. If the stack is empty, it throws a *std::out_of_range* exception;
- *isEmpty* returns true if the stack is empty, otherwise false. This method is used internally by *pop* and *peek* methods to check the stack state before proceeding.

The main function shows how to use the generic *Stack* class with different data types (integers - *Stack<int>* - and strings - *Stack<std::string> stringStack*) and handles the exceptions thrown from the class.

Non-type template parameters allow the passing of values (not types) as parameters to templates. These values can have one of the following types:

- An integer type;
- A floating point type (since C++20);
- A literal class type (since C++20);
- An enumeration type;
- std::nullptr_t;
- A pointer or reference to an object;
- A pointer or reference to a function;
- A pointer or reference to a member function.

Non-type template parameters can be used to specify things like array sizes, buffer sizes, or any compile-time constant that influences the behavior of a template.
The example below implements an *Array* class which uses non-type template parameters. The class represents a fixed-size array, allowing for element access, size retrieval, and basic array operations. The size of the array is determined at compile time through the non-type template parameter *Size*:

template<typename T, int Size> class Array.

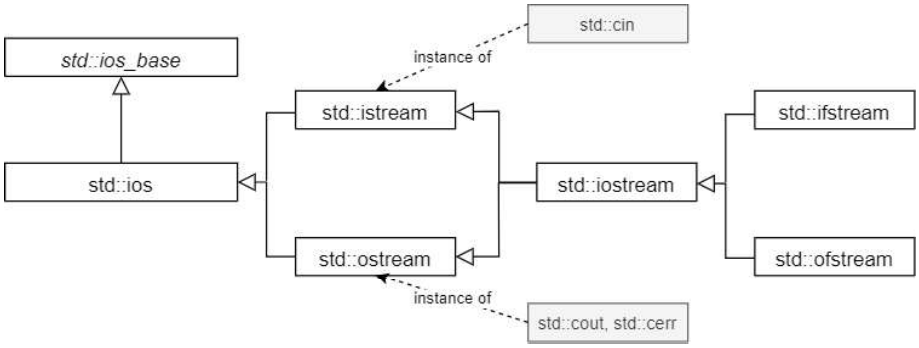
```
#include <stdexcept>
#include <iostream>

template<typename T, int Size>
class Array {
private:
    T data[Size]; // Fixed-size array of type T and size
Size

public:
    // Default constructor
    Array() = default;

    T& at(int index) {
        if (index < 0 || index >= Size) {
            throw std::out_of_range("Index out of range");
        }
        return data[index];
    }

    const T& at(int index) const {
        if (index < 0 || index >= Size) {
            throw std::out_of_range("Index out of range");
        }
        return data[index];
    }

    T& operator[](int index) {
        return data[index];
    }

    // Returns the size of the array
    int size() const {
        return Size;
    }

    // Example of a member function to fill the array with a
specific value
    void fill(const T& value) {
        for (int i = 0; i < Size; ++i) {
            data[i] = value;
        }
    }
};
```

```
// Example usage
int main() {
    Array<int, 5> myArray;

    // Filling the array with the value 10
    myArray.fill(10);

    // Accessing elements
    for (int i = 0; i < myArray.size(); ++i) {
        std::cout << myArray[i] << ' ';
    }
    std::cout << std::endl;

    // Attempt to access an element out of bounds with
exception handling
    try {
        std::cout << myArray.at(10) << std::endl;
    } catch (const std::out_of_range& e) {
        std::cout << "Exception caught: " << e.what() <<
std::endl;
    }

    return 0;
}
```

```
10 10 10 10 10
Exception caught: Index out of range
```

Streams in C++

A stream is a sequence of bytes that can be read only in a sequential manner. In C++, input-output operations are implemented with streams and are provided by the *iostream* library. Streams are powerful abstractions that allow for reading or writing data to different types of devices (without knowing their internal organization) by only interacting with the streams linked to those devices.



STL input-output library

There are two types of streams in C++: *input streams* (the *istream* class in the diagram above) which are linked to data producers (such as keyboards, input files, network sockets etc.), and output streams (the *ostream* class in the diagram above) which are used to store data from consumers (such as the screen or an output files).

We already used streams throughout this laboratory: *cin* (*istream* object linked to the standard input), *cout* (*ostream* object tied to the standard output), *cerr* (*ostream* object linked to the standard error providing unbuffered output), and *clog* (*ostream* object tied to the standard error with buffered output).

The most used operator for input streams is the stream extraction operator (operator >>) which is used to extract bytes from a stream. For instance, consider a console application requiring data input. Upon pressing keys on the keyboard, their corresponding key codes are inserted into the input stream. Within the C++ program, the developer employs the *cin* object to extract values from the standard input stream, subsequently storing them into variables.

Similarly, the most frequently used operator for output streams is the stream insertion operator, which inserts data into the output stream. For instance, the code snippet *cout << "Hello world!"*; inserts the "Hello world" string into the stream linked to the standard output, where it is then used by the data consumer (the screen in this context).

To enable custom objects to be used with streams, the stream insertion (<<) and extraction (>>) operators must be overloaded specifically for those objects. For example, if there is a *Person* class that includes name and age attributes, one would implement the stream insertion operator to output a Person object's information to an output stream, and the stream extraction operator to input information from an input stream into a *Person* object, as shown below:

```
#include <iostream>
#include <string>
using namespace std;

class Person {
private:
    string name;
    int age;

public:
    Person() : name(""), age(0) {}
    Person(string name, int age) : name(name), age(age) {}

    // Friend declarations for the stream operators
    friend ostream& operator<<(ostream& out, const Person&
person);
    friend istream& operator>>(istream& in, Person& person);
};

// stream insertion operator overload
ostream& operator<<(ostream& out, const Person& person) {
    out << "Name: " << person.name << ", Age: " <<
person.age;
    return out;
}

// stream extraction operator overload
istream& operator>>(istream& in, Person& person) {
    in >> person.name >> person.age;
    return in;
}

int main() {
    Person person;
     cout << "Please enter the person's name (no spaces)
followed by age:" << endl;

    // Reading person's details from the console
    cin >> person;

    // Outputting person's details to the console
    cout << "You entered: " << person << endl;

    return 0;
}
```

```
Please enter the person's name (no spaces) followed by age:
```

```
Alice
20
You entered: Name: Alice, Age: 20
```

Input validation with input streams

By default, C++ streams do not throw exceptions but instead use error flags to indicate errors. The base class for streams in C++ *ios_base* , defines several flags which are used to indicate the state of the stream:

- *goodbit*: indicates that everything is ok;
- *badbit:* a fatal error occurred on a read /write operation;
- *eofbit:* the stream has reached the end of a file;
- *failbit:* a non-fatal error occurred.

Also, the class provides functions to access the state of these bits: *good*(), *bad*(), *eof*() and *fail*() respectively. The *clear*() function clears all flags and restores the stream to the goodbit state.

Input validation is the process of checking whether the user input meets the expected constraints.

The code below shows an example of input validation: the user is asked to enter the fabrication year of a car and the code checks whether the value entered by the user is an integer between 1886 and 2024, inclusive.

```
#include <iostream>
#include <limits> // For std::numeric_limits

void getCarFabricationYear() {
    int year;
    bool isValid = false;

    while (!isValid) {
        std::cout << "Enter the fabrication year of the car
(1886-2024): ";
        std::cin >> year;

        // Check if the input is a valid integer and within
the expected range
        if (std::cin.fail() || year < 1886 || year > 2024) {
            std::cout << "Invalid input. Please enter a year
between 1886 and 2024." << std::endl;

            // Clear the error flag on cin
            std::cin.clear();
```

```
            // Ignore the rest of the line to avoid failing
on the same input

std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
        } else {
            isValid = true;
        }
    }

    std::cout << "You entered the year: " << year <<
std::endl;
}

int main() {
    getCarFabricationYear();
    return 0;
}
```

```
Enter the fabrication year of the car (1886-2024): 2100
Invalid input. Please enter a year between 1886 and 2024.
Enter the fabrication year of the car (1886-2024): 1789
Invalid input. Please enter a year between 1886 and 2024.
Enter the fabrication year of the car (1886-2024): testyear
Invalid input. Please enter a year between 1886 and 2024.
Enter the fabrication year of the car (1886-2024): 2020
You entered the year: 2020
```

The function *getCarFabricationYear()* prompts the user to enter a year and then performs several checks:

- checks whether the input operation failed using std::cin.fail() to check the state of the *fail* bit. An example in which the *fail* bit would be set it if the user entered non-integer values;
- it verifies whether the year entered is within the acceptable range (1886 to 2024);
- if the input is invalid for any of these two reasons, it clears the error flag on std::cin with *std::cin.clear()* to allow for further input operations;
- It uses *std::cin.ignore()* to discard the rest of the input line. This ensures that any remaining characters in the input stream do not affect subsequent input operations. The parameters to ignore tell it to discard characters until it encounters a newline ('\n') or reaches the maximum

number of characters it can ignore (std::numeric_limits<std::streamsize>::max()).

The process is repeated until a valid fabrication year is entered.

File input-output in C++

In C++, file input-output operations are implemented in three classes: *ifstream*, which is suitable for reading data from a file; *ofstream*, which is used for writing data to a file; and *fstream*, which supports both input and output operations. To use these classes in a C++ program the <*fstream*> header file must be included.

To create a file in C++ requires only to instantiate an object of the appropriate class (either *ifstream* or *ofstream*) and pass the file path/filename as a parameter. Then, the stream operators can be used to perform reading/writing operations on the file. An alternative would be to use the default constructor of the file stream classes and open the file explicitly with the *open(path, mode)* function; this function takes as parameters the path to the file and a mode in which the file should be opened, as described in the table below:

Mode	**Description**
std::ios::app	Open file in **app**end mode: seek to the end of stream before each write. So all the operations will be performed at the end of the file.
std::ios::ate	Seek to the end of stream immediately after opening, but the file pointer can be moved to another location.
std::ios::binary	Open the file in **binary** mode (and not in text mode)
std::ios::in	Open the file for reading (default value for *ifstream*); used for **in**put files
std::ios::out	Open the file for reading (default value for *ofstream*); used for **out**put files
std::ios::trunc	Open file in **trunc**ate mode: discards the contents of the stream when opening (if the file already exists, its contents is erased)

After file operations are completed, the file can be explicitly closed by invoking the close() function, or by allowing the stream object to go out of scope, at which point the destructor automatically closes the file.

Below is an example demonstrating the use of *ofstream* to write data to a file:

```
#include <iostream>
#include <fstream>

int main()
{

   // open the file in append mode (all the write operations
will be performed at the end of the file)
   // if the file already exists, we just append new data to
it
   std::ofstream outFile{ "my_file.txt", std::ios::app };

   // check that we could open the file
   if (!outFile){
       std::cerr << "Failed to open the file for writing\n";
       return -1;
   }

   // use the stream insertion operator to write data to the
file
   outFile<<"Here are the first 100 natural
numbers"<<std::endl;
   for(int x{0}; x <= 100; ++x)
       outFile<<x<<" ";

   // close the file
   outFile.close();
   return 0;
}
```

This code will create the file *my_file.txt* in the current working directory with the following content:

```
Here are the first 100 natural numbers
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62
63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82
```

```
83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

The next example illustrates how an *ifstream* object can be used to read data from the file *my_file.txt*. When reading strings, the stream extraction operator stops at spaces, so the *getline* input function should be used to read strings which contain spaces. The *ifstream* object evaluates to false if any of the error flags are set.

```
#include <iostream>
#include <fstream>

int main()
{

   // open my_file txt for reading
   std::ifstream inFile{ "my_file.txt" };

   // check that we could open the file
   if (!inFile.is_open())
   {
       std::cerr << "Failed to open the file for
reading!\n";
       return -1;
   }

   // read the first line
   std::string firstLine;
   std::getline(inFile, firstLine);
   cout<<"The first line of the file is:
"<<firstLine<<std::endl;
   unsigned int nr{0};
   while (inFile)
   {

       // then read the numbers using the stream extraction
operator
       inFile>>nr;
       // and display it on the screen
       cout<<nr<<std::endl;
   }

   return 0;
   // when inFile gets out of scope, the destructor
automatically closes it
```

```
}
```

Proposed problems

1. Write a generic function called *swapValue* that swaps the values of the two elements passed as parameters. This function should handle parameters of any data type (int, double, string, and user defined classes). Demonstrate the use of this function with at least two different data types.

2. Write a function called *findMax* that computes and returns the maximum value of two given inputs. The function should be capable of handling any comparable data type (int, double, string, as well as user-defined types, provided they have the appropriate comparison operators defined (<, >)). Demonstrate the use of this function with at least two different data types

3. Implement a generic class template named *Pair* that allows storing a pair of values of any type (such as two integers, or the name of the student and the student's grade etc.). The class should have a parameterized constructor and methods to access and modify the first and the second element from the pair. Keep in mind that the pair elements can have different types. Demonstrate the use of this class with at least two different pairs of types.

4. Design and implement a class hierarchy for exception handling when working with dynamically allocated arrays and matrices. The hierarchy should start with a base exception class, *ArrayMatrixException*, from which all specific exceptions related to array and matrix operations will inherit. The class hierarchy will include the following subclasses to represent specific types of errors:
   - *IndexOutOfBounds:* Indicates attempts to access elements outside the valid range of indices in an array or matrix;
   - *InvalidOperation:* Used when operations between two arrays/matrices are invalid (for example, element-wise addition/multiplication is applied on arrays/matrices with different sizes);
   - *AllocationFailure:* Represents failures related to memory allocation for the array or matrix, such as running out of memory.

5. Design and implement a generic class template named *Matrix* that supports basic matrix operations for any numeric data type (e.g., int, short, double). The class should provide a parameterized constructor to initialize a matrix with the given dimensions (rows and columns), a function to access and modify values at a specific position in the matrix (*row, col*), the stream insertion operator, and a function for element wise matrix addition. Use the exception class hierarchy from Exercise 4 to signal errors in the *Matrix* class.
   In the main function, create a *Matrix* object that stores double elements and two *Matrix* objects with different sizes that store *unsigned short* elements.
   Write some statements that attempt to access elements outside the matrix bounds and surround them with a *try-catch* clause to check for exceptions of type *ArrayMatrixException* (or its subclasses). Then, write a statement that adds the two *unsigned short* matrices with different sizes and surround it with a try-catch clause to check for *InvalidOperation* exceptions.
   Finally, write a menu based application which allows the user to interact with the matrices that were created. Surround this code with a *try-catch* clause which can handle **any** type of exception.

6. Design and implement a student records management system that reads student data from a CSV file. A CSV (Comma-Separated Values) file is a plain text format for tabular data. Each line represents a row in a table, with values separated by commas.
   In this problem, each row contains the student's ID, name, course, and grade separated by commas.
   An example of a CSV file is given below:

```
ID, Name, Course, Grade
1001,John Doe,Mathematics,88
1002,Jane Smith,Physics,92
1001,John Doe,Physics,85
1002,Jane Smith,Mathematics,90
```

   Create a *Student* class with the following attributes: ID (alphanumeric values), name (string), course (string), grade (integer number between 0 and 100). Implement the stream insertion of the stream extraction operator such that these functions can be used to write and read data about a student into/from CSV files.

In the main function, create a dynamic array of *Students*. Prompt the user for a CSV file path and read all the data from that file and store it in the dynamic array.

For each student, compute and display the average grade across all courses. The result should be displayed in a tabular format in the following manner: the student name should be left-aligned, matching the length of the longest name by appending trailing spaces. The average grade should be aligned right and displayed on three characters.

The program should signal invalid operations through exceptions. The following exception should be defined, thrown, and handled:

- *FileNotFound* exception: If the specified CSV file does not exist or cannot be opened, the program should throw and catch a *FileNotFound* exception to notify the user instead of crashing or having undefined behavior;
- *MalformedData* exception: if the data in the CSV file is not in the expected format (for example missing fields), the program should throw a *MalformedData* exception;
- *DuplicateRecord* exception should be a subclass of *MalformedData*. The system is not designed to handle duplicate records for the same student and course, so if it encounters such duplicates, it should throw this exception indicating a duplicate record error;
- *InvalidGradeValue* exception should be a subclass of *MalformedData*. Grades are within the range [0, 100]. If a grade outside this range is encountered, the program should handle it by throwing an exception indicating that an invalid grade value was found.

In the main function, handle all these types of exceptions.