

## Séance 4 : The STL: containers and iterators

### Ressources

- **Containers** <https://isocpp.org/wiki/faq/containers>
- **iterators**: <https://cplusplus.com/reference/iterator/>
- **Full STL documentation**: [https://www.cppreference.com/Cpp\\_STL\\_ReferenceManual.pdf](https://www.cppreference.com/Cpp_STL_ReferenceManual.pdf)

### Summary of the session:

- Writing a wrapper around an int and char array[];

```
template <typename T = int /* default */>
class number {
private:
    T c;
public:

    typedef T value_type;

    // main.c: number N1(0); // 1
    // main.c: number N2(N1); // 2
    // main.c: number N; // 3
    /* 1 */ number(T const & val) : c(val) {}
    /* 2 */ number(number const & other)
        : c(other.get()) {}
    /* 3 */ number() : c(0) {}

    /* Regular getter/setter */

    T get(/*this,*/ void) const {
        return (c);
    }

    T set(T const & newVal) {
        this->c = newVal;
        return (this->get());
    }

    T set(number const & newVal) {
        this->newVal;
        return (this->get());
    }
};
```

```
/*
** See https://cplusplus.com/reference/array/array/
**
** Template on type T and size S
** size_t is also std::size_t
*/
template <typename T, size_t S>
class myArray {
private:
    T _buffer[S]; // Abstracted array here
public:
    void fill( const T & value ) {
        for (size_t i = 0 ; i < S ; ++i)
            _buffer[i] = value;
    }

    // Regular const getter
    T & get(size_t pos) const {
        return _buffer[pos];
    }

    // operator[] overload (const and non-const)
    // called with: `a[5]`, or `a.operator[](5)`
    T & operator[](size_t pos) {
        return _buffer[pos];
    }

    T const & operator[](size_t pos) const {
        return _buffer[pos];
    }

    size_t size() const { return S; }
};
```

## Séance 5 : Operators and streams

**Class** abstract/wrap features and implement additional functions on it, moreover, unlike **struct** they are considered as a type on its own in the same way as an **int** or **float**

```
template <typename T>
void doStuff(T a, T b) {
    T result = 0;

    a = a + 5;
    a = b * 2;
    result = a + a + a + a + a;
    // etc...
}
```

1. ``DoStuff()`` is a basic function doing arithmetic
2. We should be able to do that with any type  
\* ``int``, ``short``, ``float``, ``Class Number``
3. Including more complexe one  
\* ``char *``, ``char &``: Pointers/ref are also a type  
\* ``char const *(*)(T short, ...)`` complexeStuff

### Main example:

```
#include <iostream>
#include "number.hpp" // Implement this

template <typename T>
void doStuff(T a, T const &pi, int b) {
    T result = b;
    std::string token;
    int tokenCount = 0;

    a.set(a.get() + 5);
    a += pi * 2 + a;
    result = (a += pi) + a + a + a;
    std::cout << result.get() << std::endl; // Getter version (only work with classes)
    std::cout << result << std::endl; // overload version (fully generic)

    for (tokenCount = 0 ; std::operator>>(std::cin, token) ; ++tokenCount); // Complete form
    for (tokenCount = 0 ; operator>>(std::cin, token) ; ++tokenCount); // Simple form
    for (tokenCount = 0 ; std::cin >> token ; ++tokenCount); // Operator form
    std::cout << "found 0" << std::oct << tokenCount << " tokens in the istream (base 8)" << std::endl;
    std::cout << "found " << std::dec << tokenCount << " tokens in the istream (base 10)" << std::endl;
    std::cout << "found 0x" << std::hex << tokenCount << " tokens in the istream (base 16)" << std::endl;
}

int main() {
    number nInt = 0;
    number<float> nFloat(1.337);
    number<int> pi = 314;

    doStuff(nInt, pi, (int)nInt);
}
```

### Exercise

Upgrade the **class Number** to support operators so it works with the `DoStuff()` function

## Complete class Number implementation:

```
#include <ostream> // iostream contains too much, we only need the "output" part

template <typename T = int /* default */>
class number {
private:
    T c;
public:
    /* 1 */ number(T const & val) : c(val) {}
    /* 2 */ number(number const & other) : c(other.get()) {}
    /* 3 */ number() : c(0) {}
    /* Operator overload */
    // One-liners (can return T too, because we can construct with it if needed)
    // T      operator+(T const &v) const {return c + v;} // Local scope
    number<T> operator+(number<T> const &o) const {return c + o.c;} // Local scope
    template <typename T1> friend T operator+ (int, number<T1> const &o); // Parent scope

    // This will not compile if we do "a += b * 2 + a;"
    // That's why we need to return this
    // T      operator+(number<T> const &o) const {return c + o.c;}
    number<T> operator*(T const &v) const {return c * v;}

    // Several cases (each returning `*this` to allow a chain of operation):
    // * Regular return
    // * one liner with operator '+', '*'
    // * reuse of operator+=(T const &v)
    number<T> & operator+=(T const &v) { this->c += v; return (*this); }
    number<T> & operator+=(number<T> const &o) { return (c += o.c, *this); }
    number<T> & operator=(T const &v) {return (c = v, *this); }
    number<T> & operator=(number<T> const &o) { return ((*this = o.c), *this); } // Reuse

    // Cast operators (adding explicit to deny implicit cast, for the exercise)
    explicit operator int () const { return c; }

    /* Regular getter/setter */
    T get(/*this,*/ void) const {return c;}
    T set(T const & newVal) {this->c = newVal; return (this->get()); }
    T set(number const & newVal) { this->newVal; return (this->get()); }

    // Over operator<<(): See https://isocpp.org/wiki/faq/input-output#output-operator
    template <typename T1> // Declare external function as friend
    friend std::ostream& operator<< (std::ostream& out, number<T1> const &o);
};

template <typename T> // Implement function
T operator+ (int a, number<T> const &o){return a + o.c;}

template <typename T> // Implement function
std::ostream& operator<< (std::ostream& out, number<T> const &o)
{
    out << "MyValue is: " << o.c; // Can use private directly because of friend
    return out;
}
```

## Makefile used

```
TARGET := a.out

SRCS := main.cpp
OBJS := $(SRCS:.cpp=.o)

CXXFLAGS += -W -Wall -Wextra -std=c++17 -g3

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CXX) $(CXXFLAGS) $^ -o $@

clean:
    -rm -f $(OBJS)

fclean: clean
    -rm -f $(TARGET)

re: fclean all

test: main_test.o
    $(CXX) $(CXXFLAGS) $^ -o $@
    -$$@

.PHONY: all clean fclean re test

cd "/home/1laurenj/Ensta/Cours ensta IN204/Session 1/cours_5_operator_stream/"
make re
echo "1 2 3 abcd 5 6 1337 8 9 10 11 12 13 14 15 16 17 18 19 20" | ./a.out

## rm -f main.o
## rm -f a.out
## g++ -W -Wall -Wextra -std=c++17 -g3 -c -o main.o main.cpp
## g++ -W -Wall -Wextra -std=c++17 -g3 main.o -o a.out
## 3808
## MyValue is: 3808
## found 00 tokens in the istream (base 8)
## found 0 tokens in the istream (base 10)
## found 0x0 tokens in the istream (base 16)
```

## Ressources

Main ressources:

- **Basic rules and idioms for operator overload:**
  - <https://stackoverflow.com/questions/4421706/what-are-the-basic-rules-and-idioms-for-operator-overloading>
- **Operators:**
  - **Operators:** [https://en.cppreference.com/w/cpp/language/operator\\_precedence](https://en.cppreference.com/w/cpp/language/operator_precedence)
  - **Operators:** [https://cs.smu.ca/~porter/csc/ref/cpp\\_operators.html](https://cs.smu.ca/~porter/csc/ref/cpp_operators.html)
  - **IO Tutorial:** <https://www.learncpp.com/cpp-tutorial/overloading-the-io-operators/>
  - **IO Overload:** <https://isocpp.org/wiki/faq/operator-overloading>
- **IOLibrary:** <https://cplusplus.com/reference/iolibrary/>
- **IOStream:** <https://isocpp.org/wiki/faq/input-output>

## Séance 6 : C++20: Contracts, specialization and advanced notions

### Current course progress reminder:

IN204 : *Programmation Objet & Génie Logiciel*

- ☒ **Séance 1** : Introduction aux objets
- ☒ **Séance 2** : Dérivation & Héritage
- ☒ **Séance 3** : Les Modèles & la Généricité
- ☒ **Séance 4** : The STL: containers and iterators
- ☒ **Séance 5** : Operators and streams\*\*
- ☐ -
- ☐ **Séance 6: C++20: Contracts, specialization and advanced notions** <- We are here
- ☐ -
- ☐ **Séance 7** : Les exceptions
- ☐ **Séance 8** : L'héritage et le polymorphisme
- ☐ **Séance 9** : Parallélisme & Programmation Asynchrone
- ☐ **Séance 10** : Evaluation au moment de la compilation

### Ressources

- **Official course::** <https://perso.ensta-paris.fr/~bmonsuez/Cours/doku.php?id=in204:seances:seance6>
- **C++2a and constraints:**
  - **isocpp guide:** <https://isocpp.org/blog/2021/11/cpp-20-concepts> (very good)
  - **cppreference:** <https://en.cppreference.com/w/cpp/language/constraints>
  - **others:** <https://www.cppstories.com/2021/concepts-intro/>
- **Iterators:**
  - **Link 1:** <https://www.geeksforgeeks.org/introduction-iterators-c/>
  - **Link 2:** <https://www.geeksforgeeks.org/iterators-c-stl/>
  - **Custom iterators:** <https://www.internalpointers.com/post/writing-custom-iterators-modern-cpp>

## main.cpp

```
#include <iostream> /* std::cout */
#include "defines.hpp" /* For the LOG and LOG_DECL_VAR macro */
#include "prototypes.hpp" /* For the LOG and LOG_DECL_VAR macro */
#include "codelocks.hpp" // Implement this

////////////////////////////////////
// Toying with concepts

void test_concepts() {
    int i = 1;
    float f = 2.2;
    double d = 4.4;
    custom::Vector v{1,2,3};

    // To remove the "unused variable" warning
    // (we explicitly assess that it is not used, useful when generating code sometime)
    (void)v;

    regular_add(i, i); // Regular C call
    template_add(i, i); // Deduce template from parameter (int)
    template_add(f, f); // `` `` `` (float)
    template_add<float>(f, f); // Explicit call of one version

    concept_add_long(i, i); // Also deduce from parameter but using concept
    concept_add_long(f, f); //
    concept_add_long(d, d); //

    concept_add_short(i, i); // Also deduce from parameter but using concept
    concept_add_short(f, f); //
    concept_add_short(d, d); //

    concept_add_short(v, v); //

    // concept_add(v, v); // Concept compiler error
}

////////////////////////////////////
// Toying with codelocks

namespace cc = ::IN204::codeCrackingExo; // https://en.cppreference.com/w/cpp/language/namespace\_alias

template <typename T> requires cc::hasToString<T>
void codelock_counting(T const & codelock)
{
    std::cout << codelock.toString() << std::endl;
}

void test_codelocks() {
    cc::digit d;
    // custom::Vector v;
    cc::codelock_3_dials three_dials(123);
    cc::codelock_4_dials four_dials(1998);
}
```

```

cc::digital_5_dials five_dials(31337);

std::cout << d.toString() << std::endl;
std::cout << three_dials.toString() << std::endl;
std::cout << four_dials.toString() << std::endl;
codelock_counting(five_dials);
// codelock_counting(v); // Constraints violation, simple error message
}

int main(){
    test_concepts();
    test_codelocks();

    return 0;
}

```



## defines.hpp

```
#ifndef DEFINES_HPP_
#define DEFINES_HPP_

// ////////////////////////////////////// LOGS + SOME DEFINE
// // C++20 for std::cout formatting ala printf (unsupported by compilers yet)
// // Otherwise, see: https://en.cppreference.com/w/cpp/io/manip
// # include <format> /* for std::format() */
// # include <iomanip>

// Some colors, because why not
#ifdef USE_COLOR
#define CLR_RST "\x1b[0m"
#define CLR_GRN "\x1b[32m"
#define CLR_BLU "\x1b[34m"
#define CLR_YEL "\x1b[33m"
#define CLR_BOLD "\x1b[1m"
#else
#define CLR_RST ""
#define CLR_GRN ""
#define CLR_BLU ""
#define CLR_YEL ""
#define CLR_BOLD ""
#endif // !USE_COLOR

// In case the pretty_function macro is not defined (ex: visual studio on windows)
// if !defined(__PRETTY_FUNCTION__) && !defined(__GNUC__)
#define __PRETTY_FUNCTION__ __FUNCSIG__
// endif

// Some inline logging to simplify the code later during debug
#define LOG_DECL_VAR static size_t g_log_line; // Zeroed by default because of the static keyword
#define LOG(v) (std::cout << "[" << std::setw(2) << ++g_log_line << "]" " \\  
    << CLR_BLU CLR_BOLD << __FILE__ << CLR_RST \\  
    << ":" << CLR_YEL << __LINE__ << CLR_RST \\  
    << ":\t" << CLR_GRN << __PRETTY_FUNCTION__ << CLR_RST \\  
    << "{" << #v << " = " << (v) << "}" \\  
    << std::endl)

LOG_DECL_VAR; // To instantiate the static global variable (for the log line number)

#define ADD_CODE { LOG(a + b); return a + b; }

#endif /* !DEFINES_HPP_ */
```

## prototypes.hpp

```
#ifndef PROTOTYPES_HPP_
#define PROTOTYPES_HPP_

#include <ostream> // std::ostream

namespace custom { // A toy namespace
    struct Vector {
        int x;
        int y;
        int z;
        Vector operator+(auto const &o) const {
            return Vector{x + o.x, y + o.y, z + o.z};
        }

        // // Compilation error without the cast operator:
        // defines.hpp:33:42: error: cannot convert 'custom::Vector' to 'int' in return
        //      33 | #define ADD_CODE { LOG(a + b); return a + b; }
        operator int () const { return this->x; } // for "return (Vector + Vector);" to works

        friend std::ostream& operator<< (std::ostream& out, Vector const &o) {
            return out << "\n\t{x=" << o.x << ", y=" << o.y << ", z=" << o.z << "}";
        }
    };

    // Creating some concept as a general exercise
    template <typename T> concept addable = requires(T a, T b){a + b};
    template <typename T> concept isNotIntOrFloat = !(std::integral<T> || std::floating_point<T>);
} // !namespace custom

/* ***** */ int regular_add(int a, int b) ADD_CODE
template <typename T> int template_add(T a, T b) ADD_CODE
template <> /* Specialized */ int template_add(float a, float b) ADD_CODE

// Full syntax
template <typename T> requires std::integral<T> int concept_add_long(T a, T b) ADD_CODE
template <typename T> requires std::floating_point<T> int concept_add_long(T a, T b) ADD_CODE
template <typename T> requires custom::isNotIntOrFloat<T> int concept_add_long(T a, T b) ADD_CODE

// Abbreviation syntax (we can use typename, class, or now a constraint for T)
template <std::integral T> /**/ int concept_add_short(T a, T b) ADD_CODE
template <std::floating_point T> int concept_add_short(T a, T b) ADD_CODE
template <custom::isNotIntOrFloat T> int concept_add_short(T a, T b) ADD_CODE

#endif /* ! PROTOTYPES */
```

## codeLocks.hpp

```
#ifndef CODELOCKS_HPP_
#define CODELOCKS_HPP_

#include <string>
#include <array>

namespace IN204 {
    namespace codeCrackingExo {

        template <typename T>
        concept hasToString = requires (T t)
        {
            t.toString();
        };

        //////////////////////////////////////
        /// digit

        class digit {
        private:
            char d;
            std::string const base;
        public:
            digit(char d = 0, std::string const & base = "0123456789") : d(d), base(base) {}

            std::string toString() const { return std::string() + base[d % base.length()]; }
            digit &operator=(int v) { return (d = v, *this); }
        }; // !class

        //////////////////////////////////////
        /// codeLock

        template <std::size_t S, typename D = digit>
        class codeLock {
        private:
            std::array<D, S> code;
        public:
            // @param defaultCode The value to initialize the codeLock at
            codeLock(int defaultCode) : code{0} {
                // Initialize
                for (auto it = code.rbegin(); (it != code.rend() && defaultCode != 0); ++it) {
                    *it = (defaultCode % 10);
                    defaultCode /= 10;
                }
                // if (defaultCode != 0)
                //     std::throw // TODO
            }

            std::string toString() const {
                std::string rval;

                for (auto const & item : code)
```

```

    rval += item.toString();
return rval;
}
}; // !class

////////////////////////////////////
/// aliases

template <std::size_t S, typename D = digit>
class digitalCodelock : public codelock<S, D> {
};

template <std::size_t S, typename D = digit>
class verboseCodelock : public codelock<S, D> {
};

using codelock_3_dials = codelock<3>;
using codelock_4_dials = codelock<4>;
using digital_5_dials = digitalCodelock<5>;

} // ! codeCracking
} // !IN204

// typedef 4dials_combo codelock<>;

#endif /* ! CODELOCKS_HPP_ */

```

## Makefile used

```
TARGET := a.out
SRCS := main.cpp
OBJS := $(SRCS:.cpp=.o)

# Ubuntu and MinGW: sudo apt-get install gcc-10 g++-10
CXX = g++-10 # overwrite default g++ on my system which is version 9
CXXFLAGS += -W -Wall -Wextra -std=c++20

all: $(TARGET)

color: CXXFLAGS += -DUSE_COLOR
color: fclean all

$(TARGET): $(OBJS)
    $(CXX) $(CXXFLAGS) $^ -o $@

clean:
    -rm -f $(OBJS)

fclean: clean
    -rm -f $(TARGET)

re: fclean all

.PHONY: all clean fclean re test color

cd "/home/1laurenj/Ensta/Cours ensta IN204/Session 1/cours_6_Cpp20_and_contracts/"
make re
./a.out

## rm -f main.o
## rm -f a.out
## g++-10 -W -Wall -Wextra -std=c++20 -c -o main.o main.cpp
## g++-10 -W -Wall -Wextra -std=c++20 main.o -o a.out
## [ 1] prototypes.hpp:31: int regular_add(int, int){a + b = 2}
## [ 2] prototypes.hpp:32: int template_add(T, T) [with T = int]{a + b = 2}
## [ 3] prototypes.hpp:33: int template_add(T, T) [with T = float]{a + b = 4.4}
## [ 4] prototypes.hpp:33: int template_add(T, T) [with T = float]{a + b = 4.4}
## [ 5] prototypes.hpp:36: int concept_add_long(T, T) [with T = int]{a + b = 2}
## [ 6] prototypes.hpp:37: int concept_add_long(T, T) [with T = float]{a + b = 4.4}
## [ 7] prototypes.hpp:37: int concept_add_long(T, T) [with T = double]{a + b = 8.8}
## [ 8] prototypes.hpp:41: int concept_add_short(T, T) [with T = int]{a + b = 2}
## [ 9] prototypes.hpp:42: int concept_add_short(T, T) [with T = float]{a + b = 4.4}
## [10] prototypes.hpp:42: int concept_add_short(T, T) [with T = double]{a + b = 8.8}
## [11] prototypes.hpp:43: int concept_add_short(T, T) [with T = custom::Vector]{a + b =
## {x=2, y=4, z=6}}
## 0
## 123
## 1998
## 31337
```

## Course 7 : Error management and exceptions

### Current course progress reminder:

IN204 : *Programmation Objet & Génie Logiciel*

- ☒ **Course 1** : Introduction to objects
- ☒ **Course 2** : Derivation and heritage
- ☒ **Course 3** : Template and genericity
- ☒ **Course 4** : The STL: containers and iterators
- ☒ **Course 5** : Operators and streams\*\*
- ☐ **Course 6**: C++20: Contracts, specialization and advanced notions
- ☐ -
- ☐ **Course 7 : Exceptions** <- We are here
- ☐ -
- ☐ **Course 8** : Heritage and polymorphism
- ☐ **Course 9** : Asynchronous and parallel programming
- ☐ **Course 10** : Compile time evaluation

## Error management

### Screen.hpp

```
#ifndef SCREEN_HPP_
#define SCREEN_HPP_

#include <vector>
#include "Pixel.hpp"

// To test `Screen(someRandomStruct iAmObviouslyNotASize)`
// Note: "new Pixel[iAmObviouslyNotASize]" works because
//       we got a cast operator here (implicit conversion)
// Note2: Don't do that in your project :p, it's for the test
struct someRandomStruct {
    operator int() { return -1; }
};

class Screen {
    // By default, every attribute is private within a class
    // (but we explicit the 'private:' anyway for readability
private:
    // Using a vector would be better, but we do it also by hand for
    // the exercise with new/delete.
    std::vector<Pixel> pixels_vector;

    Pixel * pixels_manual;
    size_t size;
public:
    // https://en.cppreference.com/w/cpp/language/nullptr
    // https://en.cppreference.com/w/cpp/language/new
    Screen(size_t size = 0) : pixels_manual(nullptr), size(size) {
        pixels_manual = new Pixel[size]; // Can throw
        pixels_vector.resize(size);
    }
    Screen(someRandomStruct iAmObviouslyNotASize) {
        std::cout << __PRETTY_FUNCTION__ << ": Before exception" << std::endl;
        pixels_manual = new Pixel[iAmObviouslyNotASize]; // Will throw std::bad_array_new_length
        std::cout << __PRETTY_FUNCTION__ << ": After exception" << std::endl;
    }
    // NO DEFAULT, OR WE WILL GET MEMORY CORRUPTION ON THE SECOND DESTRUCTOR
    // --> { Screen a(10); Screen b(a); } // Program could crash here (double memory free)
    // Screen(Pixel const &) = default;
    ~Screen() {
        delete pixels_manual;
        // No need to delete pixels_vector, it will get destroyed implicitly
        // (because Vector<> has a destructor that will get called)
    }
    auto operator=(Screen const &o) -> Screen & {
        // Vector version
        { // Just an extra local stack scope (to group code and prevent local variable to spread)
            pixels_vector = o.pixels_vector;
        }
        { // Just an extra local stack scope (--)
```

```

        // Manual version
        auto tmp = new Pixel[size]; // Can throw
        // We delete AFTER (in case the new operator throws, so the class Screen never has invalid memory)
        delete pixels_manual;
        // Old C style copy (but no memcpy or std::copy, we don't know if Pixel is a PoD)
        // See https://en.cppreference.com/w/cpp/language/classes#POD\_class
        for (size = 0 ; size < o.size ; ++size)
            tmp[size] = o.pixels_manual[size]; // pixel.operator=(...) noexcept
        pixels_manual = tmp; // Assign once everything is ready and return safely
    }
    return *this;
}
}; // !class

#endif /* !SCREEN_HPP_ */

```

## Pixel.hpp

```

#ifndef PIXEL_HPP_
#define PIXEL_HPP_

class Pixel {
private:
    std::uint32_t v;

public:
    // The 4 methods to respect the Coplan form (Default CTor/DTor + copy + operator=())
    // (CTors/DTors == Constructor/Destructor)
    Pixel(std::uint32_t argb = 0) : v(argb) {}
    Pixel(Pixel const &) = default;
    ~Pixel() = default; // Defaulted, a memory copy works here
    auto operator=(Pixel const &) noexcept -> Pixel & = default; // Alternate auto syntax
    //////////

    auto A() const -> std::uint8_t { return ((v >> 8*0) & 0xFF); }
    auto R() const -> std::uint8_t { return ((v >> 8*1) & 0xFF); }
    auto G() const -> std::uint8_t { return ((v >> 8*2) & 0xFF); }
    auto B() const -> std::uint8_t { return ((v >> 8*3) & 0xFF); }

}; // !class

#endif /* !PIXEL_HPP_ */

```



## main.cpp

```
#include <iostream>
#include "Screen.hpp"
#include "Pixel.hpp"

// using namespace std; // Don't do that, please ... just don't
// See: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rs-using

// @return the value of Pi
float i_am_bad_at_math() {
    float yes = 1;
    float ofcourse = 2;
    return (1 + 1 == 3 ? yes : ofcourse); // Returns the value of Pi
}

// @return 0 on success, -1 on failure
int classic_error_management() {
    float homework;
    float isGood;

    isGood = 3.14; // Reference value, sometime it's easier to read (avoid hardcoded unnamed values)
    homework = i_am_bad_at_math();
    // if (homework == 3.14) { // Avoid hardcoded values as much as possible, name them !
    if (homework == isGood) {
        return 0;
    }
    return -1; // Any case that is not explicitly a success is a failure, by default (safer)
}

// @brief A test function to toy with exceptions
void testException__bad_array_new_length()
{
    try { // 1
        try { // 2
            someRandomStruct testStruct;
            Screen s(testStruct); // Exception here
        } catch (std::exception& e) { // 2
            std::cout << __PRETTY_FUNCTION__ << ": Catching exception n'1 and rethrowing" << std::endl;
            throw; // re-throw the current exception for fun
        }
    } catch (std::exception& e) { // 1
        std::cout << __PRETTY_FUNCTION__ << ": Catching exception n'2 and ignoring" << std::endl;
        throw; // re-catching, and rethrow for the caller to handle this
    }
}

int main(){

    std::cout << "=====" << std::endl;
    std::cout << "== main() called" << std::endl << std::endl;

    try {
```

```

    testException__bad_array_new_length();
} catch (...) { // A catch all guard
    std::cout << __PRETTY_FUNCTION__ << ": "
        << "Called code is rethrowing to the caller function as expected"
        << std::endl;
}

///// Classic error management from caller function

{
    int rval = classic_error_management();
    std::cout << __PRETTY_FUNCTION__ << ": classic_error_management returned a " << (rval == -1 ? "Fail
}

std::cout << std::endl << "===== " << std::endl;
return 0;
}

```

## Makefile used

```
TARGET := a.out
SRCS := main.cpp
OBJS := $(SRCS:.cpp=.o)

# Ubuntu and MinGW: sudo apt-get install gcc-10 g++-10
CXX = g++-10 # overwrite default g++ on my system which is version 9
CXXFLAGS += -W -Wall -Wextra -std=c++20

all: $(TARGET)

color: CXXFLAGS += -DUSE_COLOR
color: fclean all

$(TARGET): $(OBJS)
    $(CXX) $(CXXFLAGS) $^ -o $@

clean:
    -rm -f $(OBJS)

fclean: clean
    -rm -f $(TARGET)

re: fclean all

.PHONY: all clean fclean re test color
```

## Running output

```
cd "/home/llaurenj/Ensta/Cours ensta IN204/Session 1/cours_7_exceptions/"
make re
valgrind ./a.out ## Calling with valgrind to check for memory leaks
## Pay attention to that line: "All heap blocks were freed -- no leaks are possible"

## rm -f main.o
## rm -f a.out
## g++-10 -W -Wall -Wextra -std=c++20 -c -o main.o main.cpp
## g++-10 -W -Wall -Wextra -std=c++20 main.o -o a.out
## ==449510== Memcheck, a memory error detector
## ==449510== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
## ==449510== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
## ==449510== Command: ./a.out
## ==449510==
## =====
## == main() called
##
## Screen::Screen(someRandomStruct): Before exception
## void testException__bad_array_new_length(): Catching exception n'1 and rethrowing
## void testException__bad_array_new_length(): Catching exception n'2 and ignoring
## int main(): Called code is rethrowing to the caller function as expected
## int main(): classic_error_management returned a Failure
## =====
## ==449510==
## ==449510== HEAP SUMMARY:
## ==449510==      in use at exit: 0 bytes in 0 blocks
## ==449510==    total heap usage: 3 allocs, 3 frees, 76,936 bytes allocated
## ==449510==
## ==449510== All heap blocks were freed -- no leaks are possible
## ==449510==
## ==449510== For lists of detected and suppressed errors, rerun with: -s
## ==449510== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

## Ressources

- Entry slides:
  - **Error Handling, by David Svoboda:** [https://resources.sei.cmu.edu/asset\\_files/Presentation/2016\\_017\\_101\\_484207.pdf](https://resources.sei.cmu.edu/asset_files/Presentation/2016_017_101_484207.pdf)
- Main documentation:
  - **Exceptions (guide isocpp):** <https://isocpp.org/wiki/faq/exceptions>
  - **Exceptions (google coding style):** <https://google.github.io/styleguide/cppguide.html#Exceptions>
  - **C++ Coding guidelines:** <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
  - **Reference documentation with exercises (hard):** <https://stroustrup.com/>
  - **Official course::** <https://perso.ensta-paris.fr/~bmonsuez/Cours/doku.php?id=in204:seances:seance7>
- Remember when I talked about the different stages of a variable (allocation, initialization, usage and cleanup) along with good programming practices ? here are some good readings:
  - **RAII:** <https://en.cppreference.com/w/cpp/language/raii>
  - **Rule of 3:** [https://en.cppreference.com/w/cpp/language/rule\\_of\\_three](https://en.cppreference.com/w/cpp/language/rule_of_three)
  - **man errno:** <https://man7.org/linux/man-pages/man3/errno.3.html>
- [https://github.com/JBL-Repo/IN204/blob/main/cours\\_recap.pdf](https://github.com/JBL-Repo/IN204/blob/main/cours_recap.pdf)