C++ lab 3

Recap of few concepts you used during the third lab:

1. Operator overloading

- In `C++` it is possible to change the way operator works for all **user-defined types**.
- You can overload any of the following 45 common operators:

<table>
<tr><th colspan="7">Common operators</th></tr>
<tr><th>assignment</th><th>increment</th><th>arithmetic</th><th>logical</th><th>comparison</th><th>access</th><th>other</th></tr>
<tr><td>a = b</td><td>++a</td><td>+a</td><td>!a</td><td>a == b</td><td>a[b]</td><td>a(...)</td></tr>
<tr><td>a += b</td><td>--a</td><td>-a</td><td>a && b</td><td>a != b</td><td>*a</td><td>a, b</td></tr>
<tr><td>a -= b</td><td>a++</td><td>a + b</td><td>a || b</td><td>a < b</td><td>&a</td><td></td></tr>
<tr><td>a *= b</td><td>a--</td><td>a - b</td><td></td><td>a > b</td><td>a->b</td><td></td></tr>
<tr><td>a /= b</td><td></td><td>a * b</td><td></td><td>a <= b</td><td>a->*b</td><td></td></tr>
<tr><td>a %= b</td><td></td><td>a / b</td><td></td><td>a >= b</td><td></td><td></td></tr>
<tr><td>a &= b</td><td></td><td>a % b</td><td></td><td></td><td></td><td></td></tr>
<tr><td>a |= b</td><td></td><td>~a</td><td></td><td>a <=> b</td><td></td><td></td></tr>
<tr><td>a ^= b</td><td></td><td>a & b</td><td></td><td>(since C++20)</td><td></td><td></td></tr>
<tr><td>a <<= b</td><td></td><td>a | b</td><td></td><td></td><td></td><td></td></tr>
<tr><td>a >>= b</td><td></td><td>a ^ b</td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>a << b</td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>a >> b</td><td></td><td></td><td></td><td></td></tr>
</table>

- You cannot define new operators such as `**`, `<>`, or `&|`.
- Operator precedence is unaffected by operator overloading.
- Some operators have to be implemented as member functions: `a=b`, `a(...)`, `a[b]`, `a->b`, `a->*b`
  - Example of `operator[]` as a member function:

```
class A {
    public:
        A(int size) { this->data = new float[size]; }
        ~A() { delete[] this->data; }
        float  operator[](int i) const { return this->data[i]; }
        float& operator[](int i)       { return this->data[i]; }
    private:
        float* data;
};
int main() {
    A a(2);
    a.operator[](0) = 1.0f;
    a[1] = 3.14f;
}
```

- All other operators can be either implemented as member or non-member functions.
  - Example of `operator+` as a member function:

```
struct B {
    int val;
    B(int val) { this->val = val; }
    B operator+(const B& rhs) const { return B(this->val + rhs.val); }
};
int main() {
    B b0(1), b1(2);
    B b2 = b0 + b1;  /* same as B b2 = b0.operator+(b1) */
}
```

  - Example of `operator+` as a non-member function:

```
struct B {
    int val;
    B(int val) { this->val = val; }
};
B operator+(const B& lhs, const B& rhs) { return B(lhs.val + rhs.val); }
int main() {
    B b0(1), b1(2);
    B b2 = b0 + b1;  /* same as B b2 = operator+(b0, b1) */
}
```

  - Of course if you define those two functions at once, the operator overload will become ambiguous, and compilation will fail (the compiler cannot decide wich one to call).

- However sometimes you won't have the choice because for binary operators, `a op b` will call a.operator(b) and you can not modify the type of a.

  - For example, this will not work:

```
#include <iostream>
struct C {
std::ostream& operator<<(std::ostream& os) {
    os << "C::operator<<";
    return os;
}
};
int main() { // this will not compile !
C c;
std::cout << c;
/* The compiler will look for the definition of one of the following operators: *
 *     std::ostream::operator<<(const C&)  [member function of std::ostream]      *
 *     operator<<(std::ostream&, const C&) [non-member function]                  */
}
```

  - Because you can not add a member function to the type std::ostream (it is a type defined in the standard library!) you will have to define the non-member function.
  - Of course instead of calling `(std::cout << c)` you could break everything and call `(c << std::cout)` which would call `C::operator<<(std::ostream& os)` which is the member function of class `C` that we defined ! This wouln't be consistent with the usual (i.e. expected) behaviour of `operator<<` with an `std::ostream` as input so this is a bad practice.
  - Thus the correct ways of implementing `operator::<<` to write to the standard output is always the implementation of a non-member `operator<<`:

```
#include <iostream>
struct C {
    /* nothing there */
};
std::ostream& operator<<(std::ostream& os, const C& c) {
    os << "C::operator<<";
    return os;
}
int main() { // this will work !
    C c;
    std::cout << c;
}
```

- The standard put no constraints on what most of the overloaded operators do, or on the return type, but in general, overloaded operators are expected to behave as similar as possible to the built-in operator.

- Sometimes when you implement one operator you get some others implicitly for free:

  - Since `C++20`, when you define the spaceship `operator<=>` you get all the comparisson operators (`operator==`, `operator!=`, `operator<`, `operator>`, `operator>=` and `operator<=`).

2. Signed and unsigned integers

- A signed integer can represent both positive and negative numbers.
- An unsigned integer can only store positive numbers.
- The only fundamental integer type is `int`, but you can alter its size and signedness with type modifiers.
- Signedness type modifiers:
  - `signed`-target type will have signed representation (this is the default if omitted).
  - `unsigned`-target type will have unsigned representation.
  - This means unless `unsigned` is specified, all integers are signed by default:

```
int i;          // signed integer
signed int j;   // signed integer
unsigned int k; // unsigned integer
```

- Size type modifiers :
  - `short`-target type will be optimized for space and will have width of at least 16 bits.
  - `long`-target type will have width of at least 32 bits.
  - `long long`-target type will have width of at least 64 bits, since `C++11`.
- The keyword `int` may be omitted if any of the modifiers is used:

```
short i;         // same as short int
unsigned long j; // same as unsigned long int
signed k;        // same as int
```

- There also three fundamental character types `char`, `signed char` and `unsigned char` which all are one byte size and can be used for integer arithmetic. Note that unlike `int`, the `char` type is distinct from the `signed char` type and the signedness of `char` depends on the compiler and the target platform.
- Depending on the target architecture all those types may have different sizes:
  - You can get the underlying type size in **bytes** by using the `sizeof` operator.
  - You can get the number of **bits per byte** by using the constant `CHAR_BIT` defined in `<climits>`.
  - On most modern architectures you can assume **1 byte** = **8 bits** (i.e `CHAR_BIT=8`).
  - The only guarantees given by the standard are:

    $1 = \texttt{sizeof(char)} \leq \texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)} \leq \texttt{sizeof(long long)}$

    `CHAR_BIT*sizeof(short)` $\geq$ `16 bits`

    `CHAR_BIT*sizeof(int)` $\geq$ `16 bits`

    `CHAR_BIT*sizeof(long)` $\geq$ `32 bits`

    `CHAR_BIT*sizeof(long long)` $\geq$ `64 bits (since C++11)`
- To solve this problem, since `C++11` you can use fixed width integer types with the header `<cstdint>`:
  - `int8_t`, `int16_t`, `int32_t` and `int64_t` for signed integers (8, 16, 32 and 64 bits).
  - `uint8_t`, `uint16_t`, `uint32_t` and `uint64_t` for unsigned integers.
- Given an integer spanning on `p` bits:
  - The unsigned integer can represent values from 0 to $+2^p - 1$.
  - The C++ standard allows any signed integer representation, but the majority of modern architectures use the two's complement representation. In this case it can represent values from -2^{p-1} to $+2^{p-1} - 1$.
  - There is also another representation which was used in early architectures called the one's complement. In this case the signed integer can represent values from $-2^{p-1} - 1$ to $+2^{p-1} - 1$.
- On a 64-bit architecture like on the school computers, you should get the following integer sizes:

	Size	**Unsigned**		**Signed**	
Type	bits	min value	max value	min value	max value
`char`	8	0	255	-128	+127
`short`	16	0	65,535	-32,768	+32,767
`int` / `long`	32	0	4,294,967,295	-2,147,483,648	+2,147,483,647
`long long`	64	0	18,446,744,073,709,551,615	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807

- **Integer arithmetic** is defined differently for the signed and unsigned integer types:
  - Unsigned integer arithmetic is always performed `modulo` 2^p where p is is the number of bits of the type.
  - Signed integer arithmetic has `undefined behavior` when the operation overflows (when the result does not fit in the result type), i.e. the compiled program is not required to do anything meaningful.

3. A gentle introduction to template metaprogramming

- Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

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

int main() {
    float a=0.5f, b=2.5f;
    int c=0, d=1;

    max<float>(a,b);   /* ok, returns 2.5f */
    max<int>(a,b);     /* ok, returns 2    */

    max<int>(c,d);     /* ok, returns 1    */
    max<float>(c,d);   /* ok, returns 1.0f */

    max<float>(a,d);   /* ok, returns 1.0f */
    max<int>(b,d);     /* ok, returns 2    */

    max(a,b);  /* ok, T is deduced to be float, returns 2.5f */
    max(c,d);  /* ok, T is deduced to be int,   returns 1    */

    max(a,d);
    /* This one will not work:
        template argument deduction/substitution failed:
            deduced conflicting types for parameter 'T' ('float' and 'int')
    */
}
```

- We have also the possibility to write class templates , so that a class can have members that use template parameters as types.

```
template <typename T>
struct Point {
    T x, y;
    Point(T x, T y) {
        this->x = x;
        this->y = y;
    }
    Point operator+(const Point& other) const;
};

template <typename T>
Point<T> Point<T>::operator+(const Point& other) const {
    return Point<T>(this->x + other.x, this->y + other.y);
}

int main() {
    Point<int> p0(4, 6), p1(2, 3);
    Point<int> p2 = p0 + p1;

    Point<float> p3(1.0f, 3.14f), p4(1.0f, 0.0f);
    Point<float> p5 = p3 + p4;

    Point<int> p6 = p0 + p3; /* this expression will not compile, there is no such operator */
}
```

Note that `Point<float>` is **not the same type** as `Point<int>`. You cannot create an C-array of `Point` that would store `Point<int>` and `Point<float>` at the same time (because `Point` is not a type but a class template).

- You can also mix the concepts of class templates and function templates. For example if you wanted to define an generic `operator+` between a Point and a Point as a member function you could implement this as the following:

```cpp
template <typename T>
class Point { /* previous implementation of Point */
    template <typename U>
    Point<T> operator+(const Point<U>& other);
};

template <typename T>
template <typename U>
Point<T> Point<T>::operator+(const Point<U>& other) const {
    return Point<T>(this->x + other.x, this->y + other.y);
}

/* now the expression 'Point<int> p6 = p0 + p3;' would work */
```

- If you want to define a different implementation for a template when a specific type is passed as template parameter, you can declare a specialization of that template.

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

/* explicit specialization of max<T> for T = int */
template <>
int max(int a, int b) { return 42; }

int main() {
    float a=0.5f, b=2.5f;
    int c=0, d=1;

    max(a,b);   /* ok, returns 2.5f */
    max(c,d);   /* ok, returns 42   */
}
```

- There no real difference between a fully specialized template function and a non-template regular function other than the fact that when the compiler looks for a matching signature type for the function call, it will first pick a non-template function that matches the required signature before trying to instantiating any available template functions that may fulfill the required signature match.

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

/* explicit specialization of max<T> for T = int */
template <>
int max(int a, int b) { return 42; }

/* non-template function declaration */
int max(int a, int b) { return 43; }

int main() {
    float a=0.5f, b=2.5f;
    int c=0, d=1;

    max(a,b);        /* ok, implicit call to max<float>, returns 2.5f */
    max(c,d);        /* ok, returns 43 */
    max<int>(c,d);   /* ok, explicit call to max<int>, returns 42 */
}
```

- Because templates are only compiled when required, this forces a restriction for multi-file projects: **the implementation (definition) of a template class or function must be in the same file as its declaration**. That means that you cannot separate the interface in a separate **header file**, and that you must include both interface and implementation in any file that uses the templates.

4. Shared libraries

- A dynamic library (also called a shared library) consists of routines that are loaded into your application at run time. When you compile a program that uses a dynamic library, **the library does not become part of your executable**. On Windows, dynamic libraries typically have a `.dll (dynamic link library)` extension, whereas on Linux, dynamic libraries typically have a `.so (shared object)` extension.
- Three steps to compile and link your code with a dynamic library on linux:
  1. Make sure the compiler knows where to look for the header file(s) for the library. As usual you just add a directory to the list of places the compiler looks for headers (`*.h` files) with the `-I` option.
  2. Make sure the linker knows where to look for the library file(s). In fact you just add a directory to the list of places the linker looks for libraries (`*.so` files) with the `-L` option.
  3. Tell the linker which library files to link with the `-l` option.
     If your library file is named `libXYZ.so` just use `-lXYZ`.
- In the `Makefile` you can use the `CXXFLAGS` and `LDFLAGS` environment variables as seen in the first lab:
  - Modify CXXFLAGS to include `-I[path to library headers]`.
  - Modify LDFLAGS to include `-L[path to library files] -l[library name]`.
  - Compile a source file to an object file:
    ```
    ${CXX} ${CXXFLAGS} -c [source.cpp] -o [target.o]
    ```
  - Link objects to create a binary:
    ```
    ${CXX} ${LDFLAGS} [file1.o] [file2.o] ... -o [binary_name]
    ```
- How to run the resulting executable (on linux):
  - You are not done yet, because the shared library is not part of your executable. The operating system has to find the dynamic library files your executable depend on to load the library code in memory prior to your code.
  - The operating system will only look to standard system directories like `/lib`, `/usr/lib` and other directories configured in `/etc/ld.so.conf` and `/etc/ld.so.conf.d`. You can list all known dynamic libraries by using the command `ldconfig -p` in a terminal.
  - It is possible to look at the dependencies of your executable by using the command `ldd [your executable name]`. If you use this tool on `exercice1` you may find something like `libBigInt.so => not found`, which is normal because `libBigInt.so` in not located in a standard system directory.
  - The environment variable `LD_LIBRARY_PATH` is consulted **at time of execution**, to provide a list of additional directories in which to search for dynamically linkable libraries.
  - For exercice 1, this would work: `LD_LIBRARY_PATH=[path to libBigInt.so] ./bin/ex1_factorial`
- How to hardcode the library path in your executable to get rid of `LD_LIBRARY_PATH` ?
  - This can be done by specifying a runtime path as a linker option.
  - Basically in your `Makefile`, you should just add the following to your linker flags:
    ```
    LDFLAGS+=-Wl,rpath=[path to library files].
    ```
- How to create a shared library ?
  - Basically a library is just a grouping of object files (`*.o`) where the main function has not been defined.
  - To create a shared library with `g++`, just use the following command:
    ```
    g++ -shared [file1.o] [file2.o] ... -o libTest.so
    ```
  - Example with the big integer library:
    1. Get the source code by using the following command (you need git):
       ```
       git clone https://mattmccutchen.net/bigint/bigint.git
       cd bigint
       ```
    2. Compile all implementation files `*.cpp` to object files `*.o`:
       ```
       for f in $(ls *.cc); do g++ -fPIC -c $f; done;
       ```
    3. Create the library:
       ```
       g++ -shared Big*.o -o libBigInt.so
       ```