Table of Contents

CPPA

Course #3

Compile time computation

You must use compile time computation as often as possible. Here, factorial using templates and a `constexpr`.

```
template <unsigned n>
struct fact
{
    enum {ret = n * fact <n-1>::ret}
}

// is used to end the recurtion
template <>
struct fact<1>
{
    enum {ret = 1}
}

// constexpr: compile time computed
constexpr unsigned ffact(unsigned n)
{
```

```
    return n == 1 ? 1u : n * ffact(n - 1);
}

int main()
{
    unsigned i = fact<5>::ret, j = ffact(5);
}
```

Match the correct function

It might be important to call the right function depending on the type of args. Here, you can specify the type of value of a `typename` to specify the function to call.

```
struct toto
{
    using value = float;
}

template <typename T>
void foo(T, typename T::value)
{
    std::cout << "hello" << std::endl;
}

template <typename T>
void foo(T, typename T::other)
{

}

int main()
{
    toto t;
    foo(t, 0);
}
```

Using enable_if

To match the template with the corresponding function, thus allow to verify the type and content of the function. Here, it checks to see if the typename M is a subset of the Matrix class.

```cpp
template <typename M,
    typename ForgetIt = typename std::enable_if<std::is_base_of<Matrix,
        M>::value, void>::type>
void diagonalize(M& m)
{
    // ...
}

int main()
{
    matrix M;
    diagonalise(m);
}
```

Remove rewrite

Remove not necessary information overload. here, we remove the `<int>` carried both by the template name and the data itself.

```cpp
template <typename T>
couple<T> make_couple(const T& v1, const T& v2)
{
    return couple<T>{v1, v2};
}

int main()
{
    couple<int>{42, 96};
    make_couple(42, 69);
    return 0;
}
```

> The addition(+) does not make a copy when called. Even if overided.

Building function on the fly

```cpp
struct placeholder1
{
    template <typename T, typename U>
```

```
    T operator()(const T& t, const& U) const
    {
        return t;
    }
};
placeholder1 pl1_;

struct Expr
{
protected:
    Expr
};

template <typename L, typename R>
struct plus : Expr
{
    plus(const L& l, const R& r) : l(l), r(r) {}
    template <typename T, typename U>
    T operator()(const L& l, const U& u) const
    {
        // can use move to give all args to r.
        return l(t, u) + r(t, u);
    }
    L l, R r;
};

template <typename I
    typename ForgetIt = typename std::enable_if<std::is_base_of<Expr,E>
                                              and std::is_base_of<Expr,I>,
                                                 void>::type
    >
struct literal : Expr
{
    literal(const I& i): i(i) {}
    template <typename T, typename U>
    T operator()(const L&, const U&) const
    {
        return i;
    }
    I i;
};

template <typename E,
```

```
    typename I,
    typename ForgetIt = typename std::enable_if<std::is_base_of<Expr,E>
                                                and not std::is_base_of<Expr,I
                                                  >, void>::type
    >
auto operator+(const E& l, I r)
{
    return plus{l, r};
}

int main()
{
    auto f = pl1_ + 50;
    std::cout << f(1, 664) << std::endl; // 51
    return 0;
}
```

Course #4

Abstract classes

Not Bloated but slow. Because there is type check in runtime and not in compile time.

```
class abstract
{
public:
    virtual void m() const = 0;
}

void foo(const abstract& a)
{
    // this does not need to have the class concrete1 implemented
    a.m();
}

class concrete1 : public abstract
{
public:
    void m() const override {/* ...*/}
}

int main()
```

```
{
    auto c = new concrete1();
    foo(*c);
    return 0;
}
```

Inclusion Polymorphism

More bloated but fast. foo() is compiled for each needed type so there is multiple intances of the same function.

```
// class abstract
// {
// public:
//     virtual void m() const = 0;
// }

template <typename C>
void foo(const C& a)
{
    a.m();
}

class concrete1 //: public abstract
{
public:
    void m() const /*override*/ {/* ...*/}
}

int main()
{
    // the time is knew at compile time
    auto c = new concrete1();
    foo(*c);
    return 0;
}
```

> -std=c++20

Concepts

1988 - “Generic programming”. This should have been in `-std=c++11`. Pour `gcc` -> `-fconcepts`.

Désembiguiser un algo.

```
template<class T>
concept ImplementSort =
    requires(T container) {
        container.sort();
        // {container.sort()} -> void;
        // contraint the return type of the function
    };

template <class T>
void mysort(T& lst)
{
    // sort the list
}

// template <ImplementSort T>
template <class T>
    requires ImplementSort<t>
void mysort(T& lst)
{
    lst.sort();
}

int main()
{
    auto cont = std::list<int>{};
    if constexpr(ImplementSort<decltype(cont)>)
    {
        cont.sort();
    }
    return 0;
}
```

Templates classes

Concepts imples that methods does not need to be templated.

```
template<class T>
```

```cpp
class example
{
    void reverse{/* ... */} requires Reversable<T>;
}
```

Best Match with concepts

```cpp
template<class T>
class example
{
    void reverse{/* ... */} requires Reversable<T>;
    void reverse{/* ... */}
}
```

Concepts with multiple checks

```cpp
template<class T>
concept FastReversable =
    Reversable<T> &&
    requires {
        typename T::flat_memory_layout;
    } &&
    std::is_convertible_v<
        typename T::flat_memory_layout,
        std::true_type
    >;
```