# Conception Objets Avancée

## Metaprogramming

Giuseppe Lipari

CRIStAL - Université de Lille

# Outline

# Outline

# Basic template

- A template is a way to generate code. The following program generates three functions

```
template<typename T>
void swap(T& a, T&b) {
    T t = a;
    a = b;
    b = t;
}
int main() {
    char z = 1, w = 2;
    int x = 3, y = 4;
    double r = 5, s = 6;

    swap(z, w);
    swap(z, y);
    swap(r, s);
}
```

```
void swap(char &a, char &b) {
    char t = a;
    a = b;
    b = t;
}
void swap(int &a, int &b) {
    int t = a;
    a = b;
    b = t;
}
void swap(double &a, double &b) {
    double t = a;
    a = b;
    b = t;
}
```

# Incomplete instantiation

- Remember that templates are not generated if not used.

```
template <class T>
class MyClass {
    T *obj;
public:
    MyClass(T *p) : obj(p) {}
    void call_fun() {
        obj->fun();
    }
    T* getPtr() { return ptr; }
};

MyClass<int> my(new int(6));
cout << my.getPtr() << endl;
```

- In the code above, function `call_fun()` is never called, so it is not compiled and the code is correct (it would give error on a `int` type!)

# Template specialization

- Another important feature is template specialization
  - Given a template with one or more parameters, we can provide special versions of the template where some of the parameter is assigned specific types

```
// general version
template<class T> class MyClass {
    ...
};

// special version with T = int
template <> class MyClass<int> {
};

MyClass<double> m;  // uses the general version

MyClass<int> n;     // uses the special version
```

# Partial template specialization

- When we have more than one type parameter, we can specialize a subset of the types

```
// general version
template<typename T, typename U> class A { ... };

// specialization on T
template<typename U> class A<int, U> { ... };

// specialization on both
template<> class A<int, int> { ... };

A<double, double> a; // uses general version
A<int, double> b;    // uses first specialization
A<int, int> a;       // uses second specialization
```

# Partial template specialization II

- Be careful with partial template specialization

```
// general version
template<typename T, typename U> class A { ... };

// specialization on T
template<typename U> class A<int, U> { ... };

// specialization on U
template<typename T> class A<T, int> { ... };

A<int, int> a;        // Error! ambiguos
```

# Specialization for functions

- Partial template specialization does not apply to functions

```
template <class T, class U> T fun(U obj) {...}

// the following is illegal
template <class U> void fun<void, U>(U obj) {...}

// the following is legal (overloading)
template <class T> T fun(Window obj) {...}
```

- The second example is not a specialization, but an overload
  - it turns out you can specialize a function by doing overload

# Integral parameters

- The parameters of a template can be constant numbers instead of types
- By constant I mean that they are known at compile time

```
template <unsigned n>
struct Num {
    static const unsigned num = n;
};

Num<5> x;
Num<8> y;

cout << x.num << endl;
cout << y.num << endl;
cout << "sizeof(Num<5>) = " << sizeof(x) << endl;
```

- Notice that the size of the struct is 1 byte (it contains nothing)
- We can do calculations with `n` at compile time

# Metaprogramming

- In C++, template metaprogramming is a set of techniques to generate code depending on certain configurations
  - all computation is performed at compilation time
- In practice, we program the compiler to perform certain computation that can be used to generate code in one way or another
- The STL provide a lot of helper functions to ease the task of template metaprogramming

# Outline

# Fibonacci at compilation time

- Consider the following code:

```
template<int n>
struct Fibonacci {
   static const int value = Fibonacci<n-1>::value + Fibonacci<n-2>::value;
};
template<>
struct Fibonacci<0> {
   static const int value = 0;
};
template<>
struct Fibonacci<1> {
   static const int value = 1;
};

int main() {
   cout << Fibonacci<10>::value << "\n";
}
```

- The program will output 55
- The number is computed at compilation time!

# How does it work ?

- We define a generic template with integral parameter
  - the template defines a constant `value` as a function of the same template with `n-1` and `n-2`
  - this value must be computed by the compiler by doing recursion on the template itself
- To stop the recursion we define two specialization of the same template
- Exercise: Write the factorial of a number in template metaprogramming

# Outline

# Very simple types

- C++ does not require much for defining a type
  - the following are all different types:

```
struct Simple {};
struct AnotherSimple {};
struct YetAnotherSimple {};
```

- Of course, you cannot do much with such types, except telling the compiler that a certain type exist
- at run time there is no code associated with it
- `sizeof(Simple)` is 1

# Defining types

- Observe the following code

```
template<typename T>
struct Cont {
    typedef T MyType;
};

Cont<int>::MyType anInteger = 2;
Cont<double>::MyType aDouble = 0.5;
```

- `Cont` does not contain anything, only the definition of another type, which is only used by the compiler, but not at run-time
- `Cont<int>` is not different from `Simple`
- However, it allows us to read the type in the template

# A selection

- Observe the following code

```
template<bool f, typename T, typename U>
struct select {};

template<typename T, typename U>
struct select<true, T, U> { typedef T Type; }

template<typename T, typename U>
struct select<false, T, U> { typedef U Type; }

int main()
{
    const int x = 5;
    select< x<10, int, double>::Type y = 5;
    select< x>=10, int, double>::Type z = 5;
    cout << y << endl;
    cout << z << endl;
}
```

# Selecting a type

- In the previous code, a type is selected depending on a constant
- The type is then used for declaring a variable in the main
- This is quite common in metaprogramming,
  - our `select` structure is the equivalent of an if/then/else in standard procedural programming

# Standard functions

- So common that the standard defines a few structures for us:

```
// in header <type_traits>
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { typedef T type; };

template<bool B, class T, class F>
struct conditional { typedef T type; };

template<class T, class F>
struct conditional<false, T, F> { typedef F type; };
```

# SFINAE

- SFINAE: Substitution Failure Is Not An Error
- Context: substituting template parameters in function templates
- When substituting a parameter in overloaded function templates, if the substitution fails (produces an error), the error is not reported, and the function is removed from the set of overloads

# SFINAE Example

- Example :

```
struct Test {
    typedef int foo;
};

template <typename T>
void f(typename T::foo) {} // Definition #1

template <typename T>
void f(T) {}               // Definition #2

int main() {
    f<Test>(10); // Call #1.
    f<int>(10);  // Call #2. Without error
                 // (even though there is no int::foo)
                 // thanks to SFINAE.
}
```

# Outline

# Run time dispatching

- Run-time dispatching is something that depends on the values of certain variables that are only known at run-time
  - It can be performed by using *if-then-else* or *switch-case* statements
  - It can also be performed using virtual functions and dynamic binding
  - the cost of doing this is often negligible
- you can also do run-time dispatching based on compile-time constants
  - however this is not always possible

# Selection example

- Suppose you are designing a container to contain pointers to objects

```
template<class T>
class MyContainer {
    vector<T*> v ;
  public:
    void push(const T * obj) { ... }
};
```

- You want to provide the ability to copy objects of type `MyContainer` by copying all contained objects (deep copy)
  - it must duplicate the pointed objects

# Polymorphic types

- There are two ways of copying an object
  - if not polymorphic, by using the copy constructor
  - if polymorphic, by using the `clone()` method
- Which one should we use?
  - We would like to make our container generic:
  - if it contains polymorphic objects, it should call the `clone` method
  - if it is not polymorphic, it should call the copy constructor

# std::is_polymorphic

- The standard provides the template class `is_polimorphic`

```
class Shape { ... }; // polymorphic hierarchy
class Triangle : public Shape { ... };
class Rectangle : public Shape { ... };

class MyClass { ... }; // not polymoprhic

int main()
{
    cout << is_polymorphic<Shape>::value << std::endl;    // true
    cout << is_polymorphic<Triangle>::value << std::endl; // true
    cout << is_polymorphic<MyClass>::value << std::endl;  // false
}
```

# First try

```
template <class T>
class MyContainer {
    ...
    MyContainer(const MyContainer & other) {
        for (T* p : other.v) {
            T* p2 = nullptr;
            if (std::is_polymorphic<T>::value) {
                *p2 = p->clone();
            } else {
                *p2 = new T(*p);
            }
            v.push_back(p2);
        }
    }
};
```

- This does not work
  - the compiler tries to compile both branches
  - if `T` has no `clone()` function, the *then* branch fails
  - if `T` is polymorphic and the base class is abstract, the *else* branch fails

# Second try

- We need to use SFINAE (to compile only the right code)
- We try with template function specialization

```
class MyContainer {
    template<bool isPolymorphic>
    T* copy_element(T *p) {
     return new T(*p);
    }

    template<>
    T* copy_element<true>(T *p) {
     return p->clone();
    }
  public:
    MyContainer(const MyContainer & other) {
          for (T* p : other.v) {
           T* p2 = copy_element<std::is_polymorphic<T>::value>>(p);
           v.push_back(p2);
         }
    };
```

- unfortunately, we cannot use partial template specialization on functions!

# Solution

- Use function overload instead of partial template specialization
- The idea is to
  1. transform the boolean into a type
  2. use the type as a function parameter, so that we can call the "right" function
- We need generate two different classes depending on a boolean

```
template<bool flag>
class Bool2Type {
    enum { value = flag };
};
```

# Solution

```
template<class T>
class MyContainer {
    std::vector<T *> v;
    T* copy_element(const T *p, Bool2Type<false>) {
        return new T(*p);
    }

    T* copy_element(const T *p, Bool2Type<true>) {
        return p->clone();
    }
public:
    MyContainer(const MyContainer &other) {
        for (int i=0; i<other.v.size(); ++i)
            T *p = copy_element(other.v[i],
                                Bool2Type<is_polymorphic<T>::value>());
            v.push_back(p);
    }
};
```

# C++17

- To simplify the code, in C++17 is now possible to use a new construct `if constexpr`
- In the previous example:

```
template <class T>
class MyContainer {
    ...
    MyContainer(const MyContainer & other) {
        for (T* p : other.v) {
            T* p2 = nullptr;
            if constexpr (std::is_polymorphic<T>::value) {
                p2 = p->clone();
            } else {
                p2 = new T(*p);
            }
            v.push_back(p2);
        }
    }
};
```

- Basically, only the branch where the result of the boolean expression evaluate to true is compiled
    - notice that the boolean expression must be a constant

# Selecting the type

- Now suppose that in the container we want to store
  1. Pointers, if the type is polymorphic
  2. Objects otherwise
- How do we declare the type of the internal vector?
  - solution: use `std::conditional`

```
template <typename T>
class MyContainer {
    using ValueType=typename conditional<is_polymorphic<T>, T*, T>::type;
    std::vector<ValueType> v;
    ...
};
```

# Example 2: optimizing intersection

- See `examples/fun_policy.cpp`

# Other examples of useful metaprogramming

- Optimized memory allocator
  - use different allocation algorithms for small objects and for large objects
  - `sizeof(t)` is computed at compilation time, you can use it to dispatch a different allocation function depending on the size of the object to allocate

# Outline

# Variadic templates

- since C++11, the standard provides compiler support for typelists

```
template<typename... Arguments>
class VariadicTemplate {
   ...
};
```

- `...Arguments` denotes a list of template arguments, separated by commas

```
VariadicTemplate<MyClass, int, string> object;
```

- This allows to (metaprogram) more complex structures by using recursion

# A safe printf

- The basic recursive function

```
template<typename T, typename ...Args>
void safe_printf(const char *s, T value, Args... args)
{
    while (*s) {
        if (*s == '%') {
            if (*(s + 1) == '%') {
                ++s;
            }
            else {
                std::cout << value;
                safe_printf(s + 1, args...); // call even when *s == 0
                return;
            }
        }
        std::cout << *s++;
    }
    throw std::logic_error("extra arguments provided to printf");
}
```

# A safe printf

- End of recursion

```
void safe_printf(const char *s)
{
    while (*s) {
        if (*s == '%') {
            if (*(s + 1) == '%') {
                ++s;
            }
            else throw std::runtime_error("invalid format string: missing ar

        }
        std::cout << *s++;
    }
}
```

# Using multiple inheritance to add capabilities to classes

- The example below shows how we can add "properties" to a class.

```
class IntProp {
    int x;
public:
    int get() const { return x; }
    void set(int p) { x = p; }
};

class StringProp {
     string s;
public:
     void setString(string x) { s = x; }
     string getString() const { return s; }
};

template <typename ...Tp>
class MySimpleClass : public Tp... {
};

MySimpleClass<IntProp, StringProp> obj;
obj.set(10);
obj.setString("COA");
```

# tuples

- A tuple is a sequence of objects of different type
  - an extension to the `std::pair<>` template

```
std::tuple<int, std::string, std::vector<int>> t;
std::get<0>() = 42;
std::get<1>() = "Giuseppe";
std::get<2>() = {1, 2, 3};
cout << std::get<0> << endl;
```

- Elements are accessed by index with `get` template function
- Fetching any element with index more than number of elements encapsulated by tuple will cause compile time error.
- The index must be a compile-time constant!

# Example

- You can "unpack" a tuple with "tie"

```
std::tuple<int,char> foo (10,'x');
auto bar = std::make_tuple("test", 3.1, 14, 'y');

std::get<2>(bar) = 100;

int myint; char mychar;

std::tie (myint, mychar) = foo;
std::tie (std::ignore, std::ignore, myint, mychar) = bar;

mychar = std::get<3>(bar);

std::get<0>(foo) = std::get<2>(bar);
std::get<1>(foo) = mychar;

std::cout << "foo contains: ";
std::cout << std::get<0>(foo) << ' ';
std::cout << std::get<1>(foo) << '\n';
```

# Outline

# Curiously Recursive Template Pattern

- This is a very simple pattern that can be used in some practical situation when dynamic binding and virtual functions cannot be used
- The basic patterns consists in a base class that has the derived class as a template parameter

```
template<class X>
class Base { ... };
class Derived : public Base<Derived> { ... };
```

# Example of use: separate counters

- One very simple use is to count the number of instances of a class

```
template<typename X>
class Counter {
    static int counter;
    int index;
public:
    Counter() : index(counter++) {}
    static int howMany() { return counter; }
};
template<typename X> int Counter<X>::counter = 0;

class MyClass : public Counter<MyClass> { ... };

cout << "There are " << MyClass::howMany() << " objects" << endl;
```

# Simulating Virtual function

- A more general example of use is to simulate virtual functions at compilation time
  - The idea is that a base class interface function performs a cast to the derived class before invoking a function
  - In this way, we can call the overloaded function in the derived class from a base class function
  - This is similar to what happens with virtual functions

# Virtual functions

```
class Base {
public:
    int f() { impl_f(); }
    int g() { impl_g(); }
    int h() { impl_h(); }

    virtual void impl_f() { ... }
    virtual void impl_g() { ... }
    void impl_h() { ... }
};
class Derived : public Base {
    ...
    virtual void impl_f() { ... }
    void impl_h() { ... }
};

Derived x;
x.f();  // calls Derived::impl_f()
x.g();  // calls Base::impl_g()
x.h();  // calls Base::impl_h()
```

# The same thing with CRTP

```
template<typename T>
class Base {
public:
    void f() { static_cast<T*>(this)->impl_f(); }
    void g() { static_cast<T*>(this)->impl_g(); }
    void h() { impl_h(); }

    void impl_f() { cout << "Base::impl_f();" << endl; }
    void impl_g() { cout << "Base::impl_g();" << endl; }
    void impl_h() { cout << "Base::impl_h();" << endl; }
};

class Derived : public Base<Derived> {
public:
    void impl_f() { cout << "Derived::impl_f();" << endl; }
    void impl_h() { cout << "Derived::impl_h();" << endl; }
};

Derived x;
x.f();  // calls Derived::impl_f()
x.g();  // calls Base::impl_g()
x.h();  // calls Base::impl_h()
```

# Differences

- Of course, virtual functions cannot be entirely substituted
  - In the previous example, the binding is `static` because it is made by the compiler
  - If you need a real dynamic binding, you cannot escape virtual functions
- Example:

```
DerivedA x
DerivedB y;
Base &r = ...; // one between x and y
r.f();         // here, a real dynamic binding is performed!
```

- There is no way to simulate this behaviour with CRTP

# Where to use

- This technique can be used in embedded systems where:
  - We may actually know all objects at compile time
  - We want to avoid the overhead of RTTI and dynamic binding
  - We still want to be flexible in using inheritance and the ability to choose the implementation at a later stage without recoding

# Polymorphic copy constructor

- Another example: how to implement the clone only once

```
// Base class has a pure virtual function for cloning
class Shape {
public:
    virtual ~Shape() {}
    virtual Shape *clone() const = 0;
};
// This CRTP class implements clone() for Derived
template <typename Derived>
class Shape_CRTP : public Shape {
public:
    virtual Shape *clone() const {
        return new Derived(static_cast<Derived const&>(*this));
    }
};
// Every derived class inherits from Shape_CRTP instead of Shape
class Square: public Shape_CRTP<Square> {};
class Circle: public Shape_CRTP<Circle> {};
```