Lecture 2

Polymorphism, Traits, Policies, and Inheritance

Kenny Erleben

Department of Computer Science
University of Copenhagen
Polymorphism 1

Traditional, define abstract base class

```cpp
class Geometry
{
  public:
    virtual std::string get_type() const = 0;
    virtual int number() const = 0;
};
```

Note:

- Common behavior in base classes
Polymorphism 2

Now make derived classes

class Sphere : public Geometry
{
    public:

    virtual std::string get_type() const { return "sphere"; }
    virtual int number() { return 1; }
};

and

class Box : public Geometry
{
    public:

    virtual std::string get_type() const { return "box"; }
    virtual int number() { return 2; }
};

and so on... (Question: why virtuals?)
Polymorphism 3

Let us define some functions that work on different geometries

```cpp
void pair_test(Geometry const * A, Geometry const * B)
{
    std::cout << "testing"
    << A->get_type()
    << " and "
    << B->get_type()
    << std::endl;
}
```

Or a little more exotic

```cpp
void collision(std::vector<Geometry *> const & geometries)
{
    for(unsigned i=0; i< geometries.size(); ++i)
        for(unsigned j=i+1; j< geometries.size(); ++j)
            pair_test(geometries[i], geometries[j]);
}
```
Polymorphism 4

Let us try our example functions

```cpp
int main()
{
    Sphere s0, s1;
    Box b0, b1, b2;

    pair_test(b2, s1);

    std::vector<Geometry*> geometries;
    geometries.push_back(&s0);
    geometries.push_back(&b0);
    geometries.push_back(&s1);
    geometries.push_back(&b1);
    geometries.push_back(&b2);

    collision(geometries);
}
```
Polymorphism 5

Observe

- Interface is bounded
- Binding of interfaces is done at run-time (dynamically)
- Easy to create heterogeneous containers

This is called

Dynamic Polymorphism

Consider the following tasks

- What if we want to extend with a new geometry type?

```cpp
class Prism : public Geometry...
```

- What if we want to extend with a method?

```cpp
virtual bool foo() const = 0;
```
Polymorphism 6

Let us try to use templates instead, no inheritance

```cpp
class Sphere
{
public:

    std::string get_type() const { return "sphere"; }
    int number() { return 1; }
};
```

and

```cpp
class Box
{
public:

    std::string get_type() const { return "box"; }
    int number() { return 2; }
};
```
Polymorphism 7

We also need to rewrite our test function

```cpp
template<typename Geometry1, typename Geometry2>
void pair_test(Geometry1 const & A, Geometry2 const & B) {
    std::cout << "testing" << A.get_type() << " and " << B.get_type() << std::endl;
}
```

and we can now use it

```cpp
int main() {
    ...
    pair_test(b0, s1);
    ... pair_test(b2, s2);
}
```
Polymorphism 8

What about?

```cpp
void collision(std::vector<Geometry *> const & geometries)
{
    for(unsigned i=0; i < geometries.size(); ++i)
        for(unsigned j = i+1; j < geometries.size(); ++j)
            pair_test(geometries[i], geometries[j]);
}
```

Answer

- Sorry, this is impossible, we cannot handle

```
std::vector<Geometry *> const & geometries
```
Polymorphism 9

Observe

- Interface is unbounded
- Binding of interfaces is done at compile-time (statically)
- Cannot create heterogeneous containers

This is called

Static Polymorphism

Consider the following tasks

- What if we want to extend with a new geometry type?

```cpp
class Prism

bool foo() const { ... };
```
Or the “bridge pattern”, idea is to switch between implementations of an interface

```cpp
class Implementation {
public:
    virtual void doit() const = 0;
};
```

Now

```cpp
class Interface {
protected:
    Implementation * m_body;
public:
    void doit() { m_body->doit() }
}
```
And we can have

class ImplA : public Implementation
{
public:
    void doit() {... }
}

and

class ImplB : public Implementation
{
public:
    void doit() {... }
}

If we know type at compile time then we can make a “static” version
template<typename Implementation>
class Interface {
  protected:
    Implementation m_body;
  public:
    void doit() { m_body.doit() }
}

Advantages

- More type safety
- No pointers (See Boost Library 1)
- Should be faster!

But we cannot swap implementation at run-time.
Fixed Traits 1

Let us study the example

template<typename T>
T accumulate( T * const begin, T * const end)
{
    T total = T();
    while(begin!=end){
        total += *begin++;
    }
    return total;
}

Problem

- The return type \( T \) may have insufficient range to store the result value
- Imagine adding 1000 values of char, is this result likely to be in range 0..255?

One solution

- Fixed Traits
Fixed Traits 2

Using partial specialization, we define results type

template<typename T>
class accumulation_traits;

template<>
class accumulation_traits<char>  
{
  public:
    typedef int result_type;
  
...  
template<>
class accumulation_traits<unsigned int>  
{
  public:
    typedef unsigned long result_type;
  
This way we can define as many “result types” as we please.
Fixed Traits 3

Now we use the new traits

```cpp
template<typename T>
typename accumulation_traits<T>::result_type
accumulate(T * const begin, T*const end)
{
    typedef typename accumulation_traits<T>::result_type result_type;
    result_type total = result_type();
    while(begin!=end)
    {
        total += *begin++;
    }
    return total;
}
```

- This is called fixed traits, because it depends only on \( T \), i.e. the caller cannot make a user-specified trait (this is called parameterized traits, more on this later).
- This is very generic, if you have an user-defined data-type class `MyBigNumber` then just create a specialization.
Fixed Traits 4

For instance somewhere in my_big_number.h (or another place) one writes

```cpp
class MyBigNumber {
  ..
};
...

template<>
class accumulation_traits<MyBigNumber>
{
  public:
    typedef MyInfinitelyBigNumber result_type;
};
```

That's it.
Fixed Traits Example

As an example let us look at a general iterator implementation using \texttt{STL}

\begin{verbatim}
template<typename iterator_type>
typename std::iterator_traits<iterator_type>::value_type accumulate(Iter begin, Iter end)
{
    typedef typename std::iterator_traits<iterator_type>::value_type value_type;
    value_type total = value_type();
    while(...){
        ...
    }
    return total;
}
\end{verbatim}

This supports both pointers and STL iterators.
Value Traits 1

What if default constructor isn’t zero?

template<>
class accumulation_traits<this_type>  
{   
  public:   
    typedef that_type result_type;   
    static result_type const zero = 0;  
};

Cool then we can write

result_type total = accumulation_traits<T>::zero;

So traits can be used to define type dependent constants.
Value Traits 2

One problem not all types (non-integral types) of static members can be initialized in header, we need a source file (Yrk!)

```cpp
that_type const accumulation_traits<this_type>::zero = 0;
```

Another more interesting idea is to use static methods,

```cpp
template<>
class accumulation_traits<this_type> {
public:
    typedef that_type result_type;
    static result_type zero() { return 0}
};
```

Cool, header-only implementation.
Parameterized Traits 1

Say we want to change the accumulation traits. We rewrite into a class implementation

```cpp
template<
    typename T, typename traits = accumulation_traits<T>
>
class Accumulation {
public:
    typename accumulation_traits<T>::result_type operator()(T * const begin, T*const end) {
        typedef typename accumulation_traits<T>::result_type result_type;
        result_type total = result_type();
        while(begin!=end){ total += *begin++; }
        return total;
    }
}
```

Oups, did you see the difference from the text-book?

- Non-static member
- We use `operator()`

This is called a functor (more about than later in course)
Parameterized Traits 2

Note the default template parameter value, this is why we need a class-implementation (template functions cannot have default values, YET)

```cpp
template<
    typename T, typename traits = accumulation_traits<T>
>
class Accumulation
{
    public:
        typename accumulation_traits<T>::result_type operator()(T * const begin, T*const end)
        {
            typedef typename accumulation_traits<T>::result_type result_type;
            result_type total = result_type();
            while(begin!=end){
                total += *begin++;
            }
            return total;
        }
    }
```
Parameterized Traits 3

One drawback, when we use it there is no template argument deduction, so we have to explicitly write

\[ \text{Accumulation<char>()(&p[0],&p[100]);} \]

This looks ugly so we create some convenience functions

```cpp
template<typename T>
typename accumulation_traits<T>::result_type accumulate(T * const ...)
{
    return Accumulation<T>()(&p[0],&p[100]);
}
```
Parameterized Traits 4

And/or

template<typename traits, typename T>
typecast traits::result_type accumulate( T * const....)
{
    return Accumulation<T,traits>()(&p[0],&p[100]);
}

That is it. Sometimes it is good practice to hide the class definition in a namespace to avoid namespace pollution.
Policies 1

One way to look at a policy

A policy can be used to define some user-specified behavior or action

As an example

```cpp
template<
    typename T,
    typename Policy = SumPolicy,
    typename traits = accumulation_traits<T>
>
class Accumulation {
public:
    typename accumulation_traits<T>::result_type operator()(T * const begin, T*const end)
    {
        typedef typename accumulation_traits<T>::result_type result_type;
        result_type total = result_type();
        while(begin!=end){ Policy::accumulate(total , *begin++ ); } 
        return total;
    }
}
```

Now we can control how “accumulation” is done
Policies 2

Say we want

class SumPolicy
{
 public:
 template<typename T1,typename T2>
 static void accumulate(T1 & total, T2 const & val) { total += val; }
}

Or

class MultPolicy
{
 public:
 template<typename T1,typename T2>
 static void accumulate(T1 & total, T2 const & val) { total *= val; }
}

Whatever we please, note

- We control the semantics
- It works as long as syntax is okay
Policies 3

Policies could be implemented differently, like

```cpp
template<
    typename T,
    typename Policy = SumPolicy,
    typename traits = accumulation_traits<T>
>
class Accumulation : public Policy {

public:
    typename accumulation_traits<T>::result_type operator()(T * const begin, T*const end)
    {
        typedef typename accumulation_traits<T>::result_type result_type;
        result_type total = result_type();
        while(begin!=end){ accumulate(total , *begin++ ); }   
        return total;
    }
```
Now one could have written

```cpp
class SumPolicy
{
public:
    template<typename T1,typename T2>
    void accumulate(T1 & total, T2 const & val) { total += val; }
}
```
The Curiously Recurring Template Pattern, well you do it like this

template<typename child_type>
class Base
{
  public:
    ...
};

template<typename T,...>
class Child : public Base< Child<T> >
{
  public:
    ...
};
This can be useful for defining common interfaces without using an abstract base class,

```cpp
template<typename child_type>
class Base {
public:

    void foo() {
        child_type & self = static_cast<child_type&> ( *this );
        self.foo();
    }

    bool goo(int cnt) const {
        child_type const & self = static_cast<child_type const &> ( *this );
        return self.goo(cnt);
    }
};
```

Now the compiler makes sure that class Child implements foo and goo.

Great for implementing concepts (facades), see Boost iterator library.
Not quite what happens if

```cpp
template<typename child_type>
class Base
{
public:
    void foo()
    {
        child_type & self = static_cast<child_type&> ( *this );
        self.foo();
    }
};

class Derived : public Base<Derived>
{
public:
};
```

An infinite loop!

Oh but shouldn’t the compiler tell us that we forgot to implement `foo` on Derived?
CRTP 4

Work around 1: “Private” Solution

```cpp
class Derived : private Base<Derived> {
public:
};
```

Work around 2: “No name clash” solution

```cpp
template<typename child_type>
class Base {
public:
    void foo() {
        child_type & self = static_cast<child_type&>(*this);
        self.goo();
    }
};
```
Another example

template<typename child_type>
class ObjectCounter
{
private:
    static unsigned int m_count = 0;
protected:
    ObjectCounter(){++m_count;}
    ~ObjectCounter(){--m_count;}
    ObjectCounter(ObjectCounter<child_type> const &){++m_count;}
public:
    static unsigned int live() { return m_count; }
};

- We share state between all derived types of same type!
Substitution Failure Is Not An Error, example

```cpp
template<typename T>
class IsClass
{
private:
    typedef char one; // size = 1 byte
    typedef struct{char a[2] } two; // size = 2 byte
    template<typename C> static one test(int C::*); // only classes
    template<typename C> static two test(...); // anything else
public:
    enum {yes = sizeof( IsClass<T>::test<T>(0) )};
    enum {no = !yes};
};
```
Now we can write

```cpp
if( IsClass<my_got_damn_type>::yes )
{
    // do something with class
}else{
    // do something with non-class
}
```

Or we might want to write pretty readable code

```cpp
template<typename T>
bool is_class(T)
{
    if(IsClass<T>()::yes)
        return true;
    return false;
}
```
So now we simply write

```cpp
my_got_damn_type dodah;
if( is_class(dodah) )
    ...
```