LECTURE 8

Abstract classes

```
1  class IFunction
2  {
3  public:
4     virtual ~IFunction(){}
5     virtual double operator()(const BVector & bvARg) const =0;
6  };
```

- `IFunction` is a special base class; one of its virtual functions is not implemented, the syntax enforcing this is the "=0" at the end of the function declaration.
- The non-implemented operator `()` member is called **pure virtual**, this kind of class if referred to as **pure virtual** or **abstract** class.
- **Abstract classes cannot be instantiated**, however classes derived from an abstract class can be, given that all their virtual members are implemented.
- All right, `IFunction` cannot be instantiated, then what is it good for?

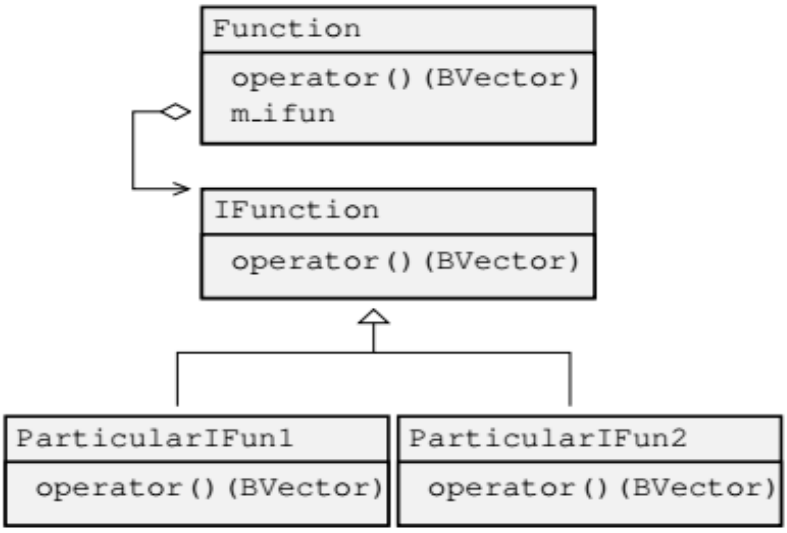
Abstract classes applied - an example

The **pointer to implementation idiom** aka **bridge pattern**

```
1  class Function
2  {
3  public:
4     explicit Function(IFunction * ifunptr) : m_ifun(ifunptr) {}
5     double operator()(const BVector & bvArg)
6     {
7        return (*m_ifun)(bvArg);
8     }
9  private:
10    IFunPtr m_ifun;//smart pointer to IFunction
11 };
```

- `Fun` has a pointer to the abstract class `IFunction` and an operator `()` calling the operator `()` of `IFunction`.
- In practice, the pointer member will point to non-abstract (i.e. **particular**) instances of classes derived from the abstract `IFunction`.
- Note: the type of `Function` does not depend on what derived class the pointer member actually points to.
- The pointer can be **redirected** during run time to other instances of possibly other type, **changing the behaviour** of the `Function` instance, but not its type.
- The implementation of the particular pointed class is **hidden** from the user of `Function` and can be modified without any changes in the user's code.

Bridge pattern graphically



- The composition of objects through pointer is denoted by ◇→
- The arrow —▷ indicates the derived-base relationship (as earlier).

Particular classes of an abstract class

```
1  class IFunctionConst : public IFunction
2  {
3  public:
4     IFunctionConst(double dArg=0.0) : m_dConst(dArg) {}
5     double operator()(const BVector & bvArg) const
6     {
7        return m_dConst;
8     }
9  private:
10    double m_dConst;
11 };
```

- `IFunctionConst` is a particular class derived from the abstract `IFunction`.
- `IFunctionConst` is no longer abstract, all its virtual functions are implemented.
- `IFunctionConst` is a function object returning a constant which is specified when constructing the instance of the class.

Particular classes of an abstract class

```
class IFunctionCoordinate : public IFunction
{
public:
   IFunctionCoordinate(BVector::size_type index=0) : m_index(index) {}
   double operator()(const BVector & bvArg) const
   {
      return bvArg[m_index];
   }
private:
   BVector::size_type m_index;
};
```

- `IFunctionCoordinate` is an other particular class derived from the abstract `IFunction`.
- `IFunctionCoordinate` is no longer abstract, all its virtual functions are implemented.
- `IFunctionCoordinate` is a function object behaving like the identity function; it returns the value of the input argument's component corresponding to `m_index`.

Particular classes of an abstract class

```
template<typename BinaryOp>
class IFunctionBinaryComp : public IFunction
{
public:
   IFunctionBinaryComp(IFunPtr ifun1, IFunPtr ifun2, BinaryOp binaryOp=
        BinaryOp())
      : m_binaryOp(binaryOp), m_ifun1(ifun1), m_ifun2(ifun2) {}
   double operator()(const BVector & bvArg) const
   {
      return m_binaryOp((*m_ifun1)(bvArg),(*m_ifun2)(bvArg));
   }
private:
   BinaryOp m_binaryOp;//outer bunary function object
   IFunPtr m_ifun1;//first inner function object
   IFunPtr m_ifun2;//second inner function object
};
```

- This is an example for the mixed used of derived classes and templates.
- The particular class `IFunctionBinaryComp` is the key class to implement binary compositions of `Function`'s (see next slide).
- `IFunctionBinaryComp` stores to pointers to `IFunction` and a binary function object. When the operator `()` is called, the argument is passed to both pointed `IFunction`'s, their operator `()` is evaluated, and the results are passed to the binary operator.

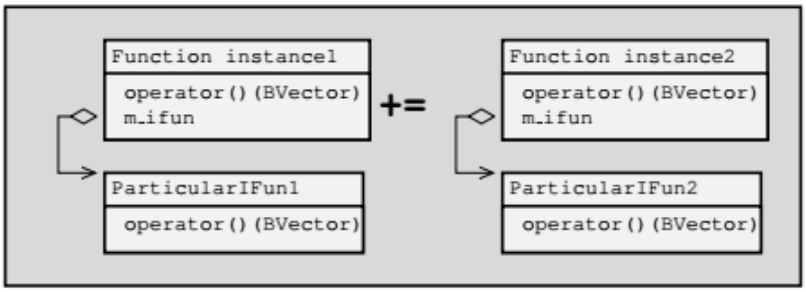
Particular classes of an abstract class

Implementing the operator += member of `Fun`

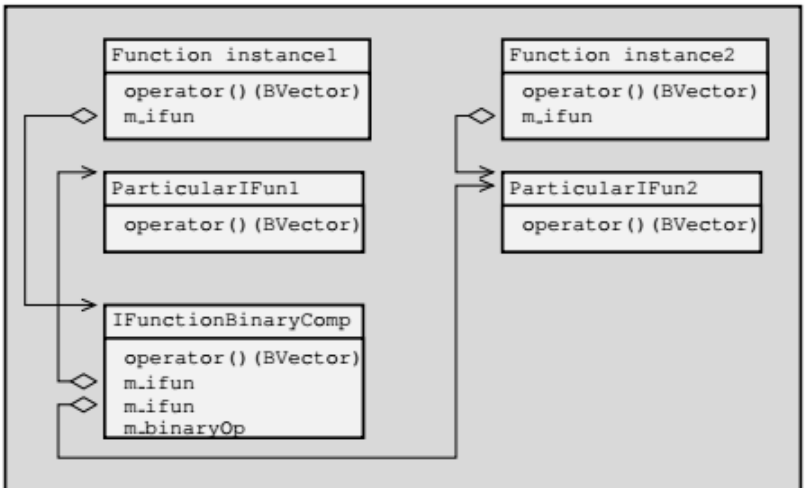
```
1 Function & exercises::Function::operator+=(const Function & fArg)
2 {
3    m_ifun=IFunPtr(new IFunctionBinaryComp<plus<double> >(m_ifun,fArg.
         m_ifun));
4    return *this;
5 }
```

- The implementation of operator += takes the pointer member of `this` and the pointer member of the argument `Function` and compose them using a `IFunctionBinaryComp` equipped with the binary operator `plus<double>`.
- Note that the pointer to `IFunction` can actually point to an instance of `IFunctionBinaryComp` since the latter class is derived from the former one, so the pointer redirecting above is legal.

Function::operator+=()



The outcome of +=:



`Function::operator+=()` - comments

- Calling `instance1+=instance2` only modifies one data member of `instance1`.
- `instance1` is not replaced by any other object and its type is not modified.
- Note that two object has pointer members pointing to an instance `ParticularIFun2`, in other words that instance is **shared** by two objects.
- Sharing via raw pointers would lead to difficulties, however using `shared_ptr`'s is one way to overcome those difficulties.
- When the operator `()` of `instance1` is called with argument `dArg`
  - `bvArg` is passed to an instance of `IFunBinaryComp`
  - the instance of `IFunctionBinaryComp` passes `dArg` to the instances `ParticularIFfun1` and `ParticularIFun2` and calls their operator `()`
  - the two values returned by the called functions are passed to the operator `()` of `m_binaryOp` which is the binary function `plus<double>` in this particular case.
  - the value returned by `m_binaryOp()` is passed to `instance1`
  - this value is returned by the operator `()` of `instance1`.
- Note: since the particular classes derived from `IFunction` can be shared by several `Function` instances, they better not change their state or behaviour when their operator `()` is executed. If they did, that would have an impact on all the `Function` instances they are shared by, which might not be desired.

Some comments on the bridge pattern

- Recall when function objects are passed to stl algorithms, they are passed by value.
- So if the function object is derived from a class, the derived part is sliced.
- Therefore derived function objects are not allowed.
- However using the bridge pattern one can overcome this.
- In particular a monomorphic function object with a pointer member is not subject to the slicing problem, since by copying the pointer member that is the address of a derived object, the derived object will not be sliced.

Templates vs inheritance

Polymorphism based on inheritance and `virtual` members (Run-time/dynamic polymorphism)

- Realizes an "is-a" relationship
- Enforces a common and explicit interface
- Can be exploited using reference/pointer semantics
- More flexible (run-time) and maintainable then template based polymorphism
- Abstraction penalty (5-10% performance overhead due to virtual method lookup)
- One can implement heterogeneous containers, polymorphic return types, etc.

Template based polymorphism (compile-time/static polymorphism)

- No inheritance hierarchy
- No common interface enforced but implicit constraints
- No abstraction penalty, faster execution
- Both reference and value semantics supported
- Does not realize an "is-a" relationship
- Types are fixed at compile time

Templates vs inheritance

Which one to use

- Be aware of the pros and cons
- Be aware of the trade-off between flexibility and performance
- Do not overuse inheritance
- Do not overuse multiple inheritance
- Google up alternative (combined) solutions

"Curiously recurring template pattern"

```
1  template<typename derived>
2  class Base {
3  public:
4    virtual ~Base(){}
5    void someMemberFunInBase();
6    double polymorphicFun(double dArg) {
7      return static_cast<derived*>(this)->implementation(dArg);
8    }
9  };
10
11 class ParticularDerived : public Base<ParticularDerived> {
12 public:
13   void someMemberFunInDerived();
14   double implementation(double dArg)
15   { //...}
16 };
```

- The CRTP pattern makes it possible to implement inheritance of members and polymorphic behaviour without `virtual` functions: each member of `Base` is also a member of `ParticularDerived` minimizing code duplication, and some of the common members are given derived-specific behaviour without performance overhead.
- However it does not yield a proper "is-a"relationship (`ParticularDerived` is not a `Base` but is a `Base<ParticularDerived>`)

Summary

- abstract (pure virtual) classes cannot be instantiated, their purpose is to set the common framework for particular derived classes
- the bridge pattern implements polymorphic behaviour, however does not change the type of the high-level object
- the bridge pattern implements dynamic (run-time) polymorphism
- template based polymorphism vs `public` inheritance