System Level Design (and Modelling for Embedded Systems)

10 – C++ and OOM

Kim Grüttner <kim.gruettner@dlr.de>
Jörg Walter <joerg.walter@offis.de>
Henning Schlender <henning.schlender@dlr.de>
Sven Mehlhop <sven.mehlhop@offis.de>

Distributed Computation and Communication, R&D Division Manufacturing
OFFIS – Institute for Information Technology

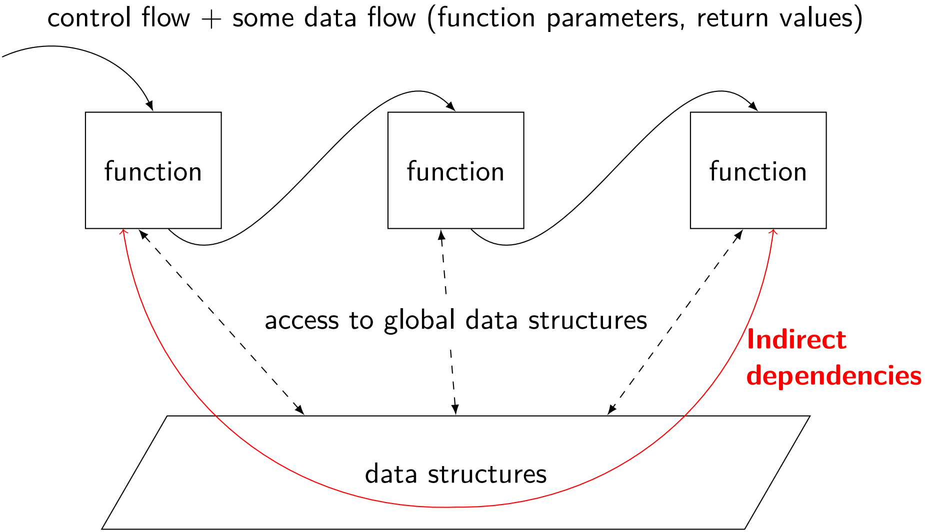
Institute of Systems Engineering for Future Mobility
German Aerospace Center (DLR-SE)

Based on the slides of M. Radetzki 2005–2008
University of Stuttgart

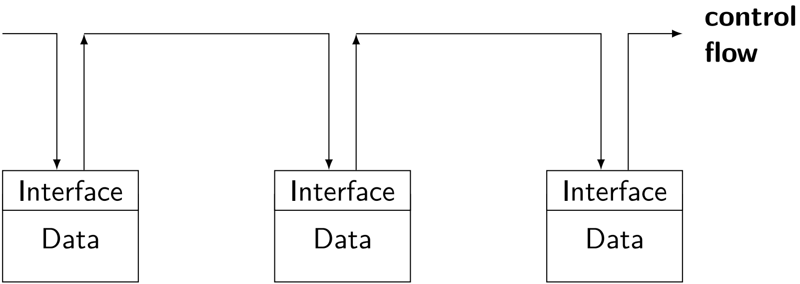
Object-Orientation (OO)

- **Programming concept first developed in the 1970s**
  - Languages, e.g.: Smalltalk, Oberon, Ada95, C++, Java
- **OO methodologies, including graphical notations**
  - e.g. OOAD, OOSE, UML (Unified Modeling Language)
- **Main concepts**
  - Encapsulation
  - Information Hiding
  - Abstraction
  - Generalization, Specialization

Traditional Programming Approach

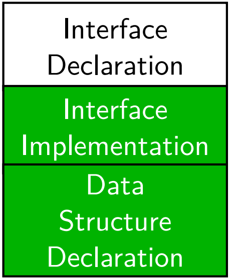


Encapsulation



- **Data encapsulated** by interface
- Access only **via interface**; ensures data consistency
- Change of internal data structures affects only the interface implementation, but not any other program parts if the interface remains the same

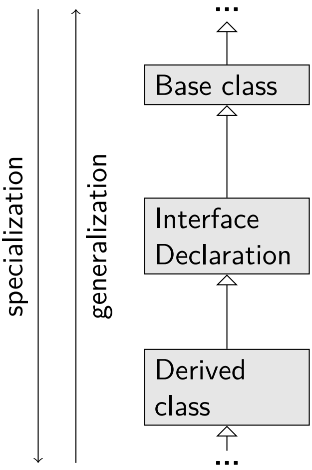
Information Hiding



- **Only interface declaration is visible** to user of data
- Interface **implementation is hidden** from user
- Data **structure layout hidden** from user
- To make use of the data, it is not necessary to know any implementation details
- Knowledge of the interface declaration is sufficient

In object-oriented methologies, *objects* and *classes* are the mechanisms for **encapsulation and information hiding.**

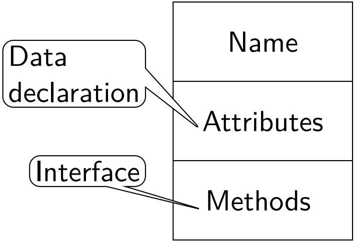
Abstraction, Inheritance



- Features of another class can be **inherited** (re-used)
- Class interface provides **abstraction** of physical things to be modelled
- Class can be extended by **deriving** from another class

Class

UML



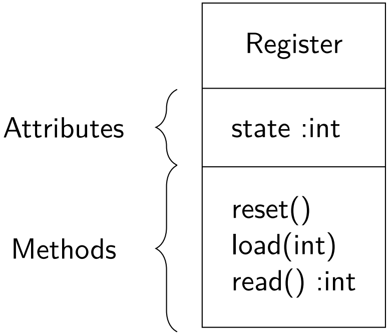
C++

File: name.cpp

```
class Name
{
  // data members (attributes)
  ...
  // member functions (methods)
  ...
};
```

Attributes and Methods

UML



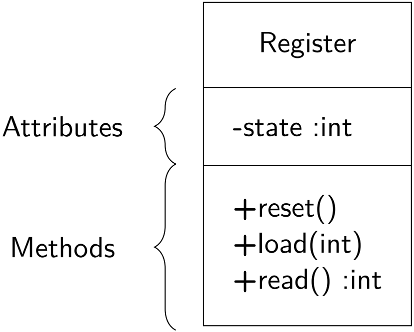
C++

File: register.cpp

```
class Register
{
  // data members
  int state;
  // member functions
  void reset();
  void load(int);
  int read() const;
};
```

Encapsulation

UML



- -: not accessible for users of class
- +: accessible for users of class

C++

File: register.cpp

```
class Register
{
private:
  int state;
public:
  void reset();
  void load(int);
  int read() const;
};
```

Class vs. Struct

File: register.h

```
class Register
{
  // data members
  int state;
public:
  // member functions
  void reset();
  void load(int);
  int read() const;
};
```

contents are **private** by default

File: register.cpp

```
struct Register
{
  // member functions
  void reset();
  void load(int);
  int read() const;
private:
  // data members
  int state;
};
```

contents are **public** by default

- aside from public / private, **class and struct are equivalent**

Information Hiding

The header file, **register.h**, is sufficient to use the class

File: register.h

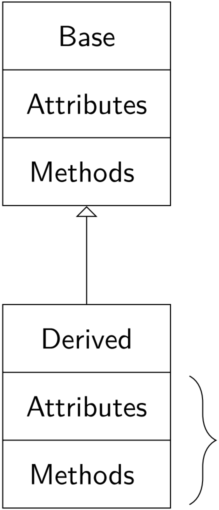
```
class Register
{
private:
  int state;
public:
  void reset();
  void load(int);
  int read() const;
};
```

The implementation details are defined in a separate file, **register.cpp**

File: register.cpp

```
#include "register.h"
void Register::reset()
{
  state = 0;
}
void Register::load(int d)
{
  state = d;
}
int Register::read() const
{
  return state;
}
```

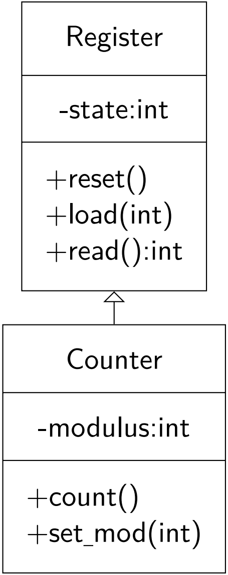
Inheritance




```
class Derived
: public Base
  {
  // new data members
  ...
  // new member functions
  ...
};
```

derived class has the attributes and methods specified here **plus** all inherited attributes and methods

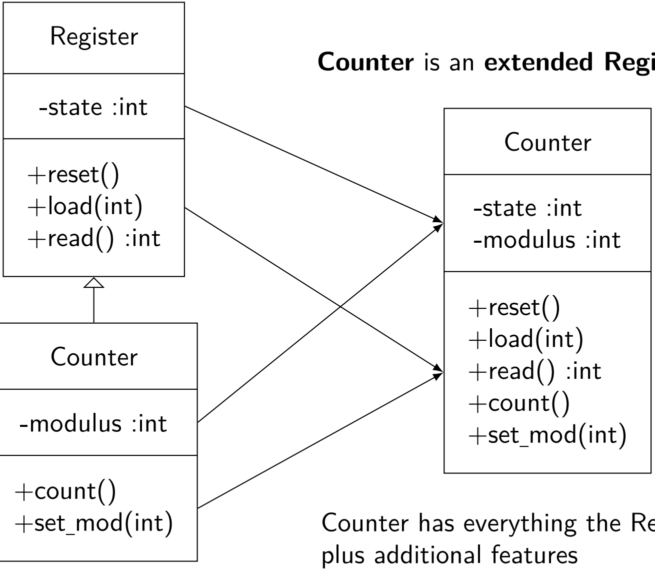
Inheritance




```cpp
class Counter
: public Register
{
  // new data members
  int modulus;
public:
  // member functions
  void count();
  void set_mod(int);
};
```

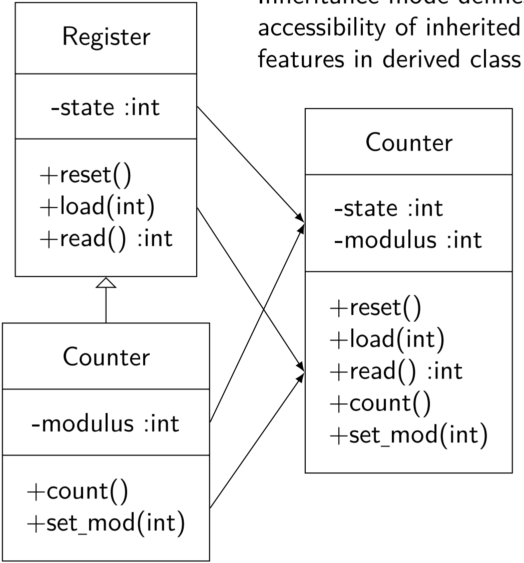
Extension by Inheritance

Counter is an **extended Register**



Counter has everything the Register has, plus additional features

Inheritance Modes in C++



Inheritance mode defines accessibility of inherited features in derived class

public inheritance: inherited features have accessibility as specified in base class

private inheritance: inherited features become private in derived class

```
class Counter
: public Register
{
   ...
};
```

Method Implementation

Implementations are inherited from Register:

- void Register::reset()...
- void Register::load(int d)...
- int Register::read() const ...

counter.cpp implements new methods:

this will not be accepted by the C++ compiler because **state** is a private attribute of Register, not accessible by Counter

```
void Count::count()
{
  state = (state + 1) % modulus;
}
void Count::set_mod (int m)
{
  modulus = m;
}
```

Access to inherited Attributes

Option 1: Access via interface methods

```
void Counter::count()
{
  load( (read() + 1) % modulus );
}
```

call to counter interface method **read()** returns value of **state**

call to counter interface method **load()** sets value of **state**

Option 2: Making attributes public

```
class Counter : public Register {
// ...
public:
  int state;
// ...
}
```

Option 3: Using C++ mode protected

```
class Counter : public Register {
// ...
protected:
  int state;
// ...
}
```

protected class members can be accessed by the class itself **and derived classes**

protected inheritance mode makes public members protected while inheriting

```
class Counter :
  protected Register
{...};
```

What is an SC_MODULE?

= =

```
SC_MODULE(Adder)
{
  sc_in<int> x;
  sc_in<int> y;
  sc_out<int> s;

  void add();
  ...
};
```

```
struct Adder
  : sc_module
{
  sc_in<int> x;
  sc_in<int> y;
  sc_out<int> s;

  void add();
  ...
};
```

```
class Adder
  :public sc_module
{
public:
  sc_in<int> x;
  sc_in<int> y;
  sc_out<int> s;

  void add();
  ...
};
```

Using A Class: Instantiation

- A class can be used to create instances, called *objects*.

reg : Register

cntr1 : Counter

cntr2 : Counter

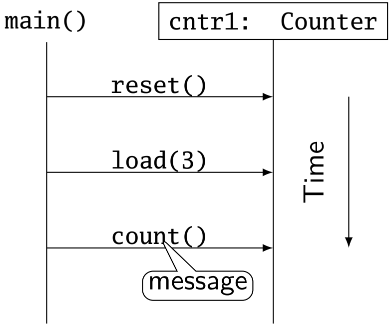
```
int main()
{
  Register reg;
  Counter cntr1;
  Counter cntr2;
  ...
}
```

- Each object has **indivual storage** for the attributes declared by its class.

Using An Object: Message Passing

- A **message** can be passed to an object
- There must be a **corresponding method** in the object‘s class
- This method is **executed and has access to the attributes** of the object

Message Sequence Chart

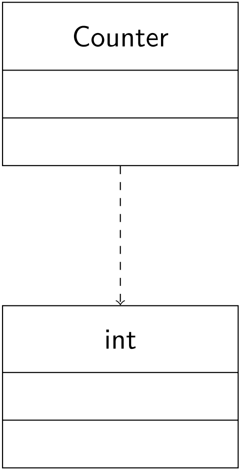



```
int main()
{
  Counter cntr1;
  ...
  cntr1.reset();
  cntr1.load(3);
  cntr1.count();
  ...
}
```

Class Relationships

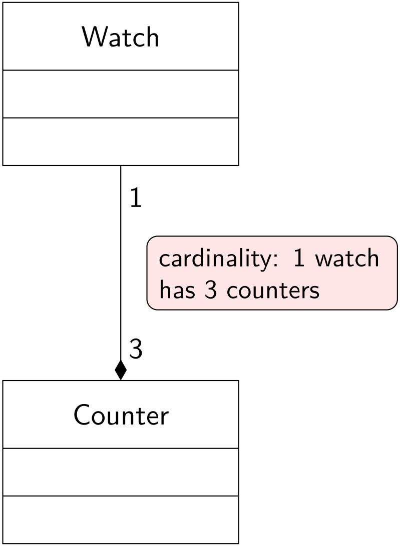
- **Inheritance: "is-a" relationship**
  - derived class is-a base class
  - *Counter* is-a *Register*
- **Composition: "has-a" relationship**
  - class C contains one or several instance(s) of another class D
  - C has-a D
  - *Watch* has-a *Counter* (for the seconds, minutes, hours)
- **Association: temporary "has-a" relationship**
  - one or several objects of a class D are associated to class C
  - e.g. students are associated to a university, association changes
- **Dependency: "uses" relationship**
  - class C uses class D, e.g. as a parameter of a method
  - e.g. *Register* uses *int* as parameter of method *load(int)*

Dependency (uses)




```
class Counter
{
  ...
  void load(int);
  ...
};
```

Composition (has-a)



File: watch.cpp

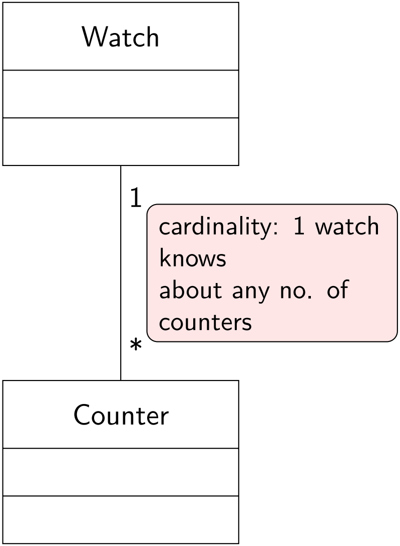
```
class Watch
{
  Counter cntr[3];
  ...
};
```

or

File: watch.cpp

```
class Watch
{
  Counter sec;
  Counter min;
  Counter hrs;
  ...
};
```

Association



File: watch.cpp

```
class Watch
{
  Counter *cntr[];
  ...
};
```

and often also (reverse pointer)

File: counter.cpp

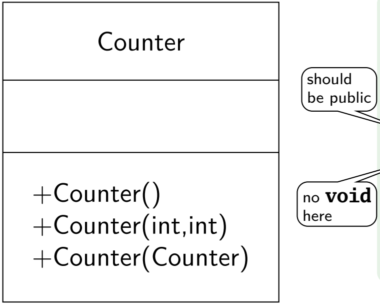
```
class Counter
{
  Watch *wtch;
  ...
};
```

the **pointers** to associated objects are initialized and **may change at run-time**

Special Methods

- Constructors
- Destructors
- Overriding of methods
- Virtual methods and dynamic binding
- Abstract (pure virtual) methods
- Interface methods and interfaces

Constructor




```
class Counter
{
public:
  Counter(); // default
  Counter(int,int);
  //copy
  Counter(Counter const&);
};
```

- A **constructor** is a method that has the name of the class.
- The constructor is called after storage allocation to **initialize** an object.
- A **default constructor** can be called without parameters.
- A **copy constructor** has exactly one parameter of the same class.

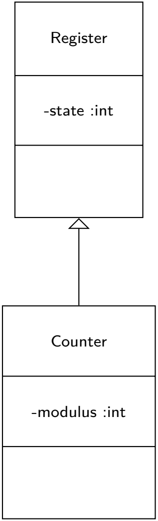
Constructor Implementation

```
Counter::Counter()
: modulus(16)
{ /* not needed */ }

Counter::Counter(int d, int m)
: Register(d)
  , modulus(m)
{}

Counter::Counter(Counter const & c)
: Register(c)
  , modulus( c.modulus )
{}
```

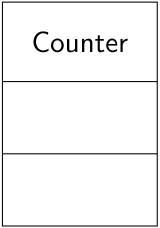
Initializer List:
-Initialize class members
-Prefer over assignment in body

Initializer List:
constructor of base class to initialize data members inherited from base class

Copy constructor:
-call copy constructor of base class(**important!**)
-copy own attributes



Absence of Constructor



If no user-defined constructor is available, C++ defines an **implicit default constructor**.

This constructor

- initializes values of built-in types (e.g. int) to default values (e.g. 0),
- calls the default constructor of the base class,
- calls the default constructors of all instantiated (has-a) objects.

It does not perform any dynamic object allocation and cannot do anything beyond the default.

Constructors in SC_MODULEs

```
SC_MODULE(adder)
{
  ...
  SC_CTOR(Adder)
  {
    ...
  }
};
```

=

```
class Adder : public sc_module
{
public:
  SC_HAS_PROCESS(Adder);
  Adder(sc_module_name);
};
Adder::Adder(sc_module_name nm)
  : sc_module(nm)
{ ... }
```

need this macro if SC_CTOR not used

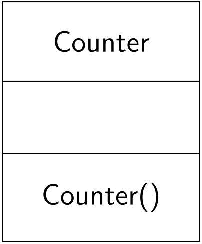
=

```
class Adder : public sc_module
{
public:
  SC_HAS_PROCESS(Adder);
  Adder(sc_module_name)
    : sc_module()
  { ... }
```

pass on module name to sc_module constructor (optional)

Note: cannot use SC_CTOR for constructor with **custom parameters**

Destructor




```
class Counter
{
  public:
  ~Counter();
  ...
};
```

should be public

no void here

- A **destructor** is a (unique method that has the name of the class, prefixed by the ∼ symbol
- The destructor is called at the **end of an object´ s lifetime** and is meant to **clean up resources** that belong to the object (has-a).
- The destructor cannot have any parameters nor return a value.
- If no explicit destructor is available, C++ defines an **implicit default destructor** which does not de-allocate any dynamically created objects.

Destructor

- **Objects in C++ have an explicit *lifetime***
  - In contrast to e.g. Java, where objects are garbage-collected, once no reference to an object exists
- **Three types of so-called *Storage Duration* exist:**
  - **Static:** Global and static objects, defined outside of classes and functions are created before `main()` starts
  - **Automatic:** local variables are deleted, when enclosing scope (function, class) is destroyed
  - **Dynamic/free:** explicitly allocated objects have to be deleted manually (**new** $\rightarrow$ **delete**)
- ***Resource Acquisition is Initialisation (RAII)***
  - Automatic variables enable **deterministic handling of resources** (memory,files,locks)
  - Acquire in constructor, release in destructor

Example: Storage duration

```
struct Watch
{ // ...
  Counter *cntr;
  Watch() : cntr( new Counter ){}
  ~Watch() { delete cntr; }
};

Watch global_watch;

int main()
{
  Watch automatic_watch;
}
```

Dynamic Allocation:

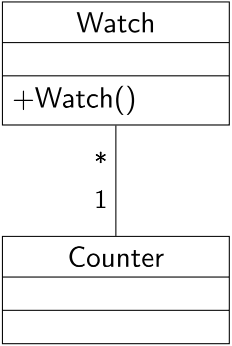
Static storage:

- order of initialisation difficult to predict

Automatic variable:

- automatic cleanup at end of function/scope
- preferred machanism in C++

Dynamic Allocation of Objects




```
class Watch
{
  Counter *cntr;
  ...
};
```

```
Watch::Watch()
{
  cntr = new Counter();
  // or: new Counter(0,32);
  // or: new Counter[16];
}
```

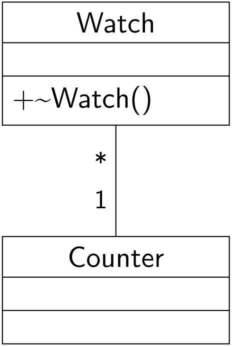
dynamic allocation

initialization (constructor)

creates array of 16 counter objects

If objects are associated (pointer to object), the pointer is often initialized in the constructor, including dynamic allocation (creation) of the object.

Deallocation of Objects




```
class Watch
{
  Counter *cntr;
  ...
};
```

```
Watch::Watch()
{
 delete cntr;
 // or: delete[] cntr;
}
```

deletes a single dynamic object

deletes an array of dynamic objects

The destructors of base classes and instantiated objects (composition) are automatically called in C++

Dynamic Allocation of SC_MODULES

automatic allocation of uut:

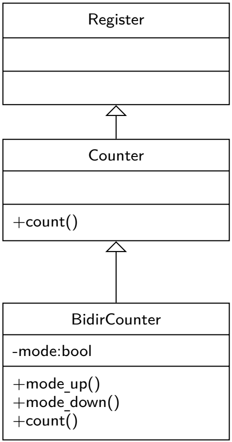
```
SC_MODULE(Testbench)
{
 Adder uut;
 sc_signal<int> ch_x, ...
 SC_CTOR(Testbench)
  : uut("uut")
  , ch_x("ch_x")
 {
   uut.x(ch_x);
   uut.y(ch_y);
   uut.s(ch_s);
   ...
 }
};
```

also possible: dynamic **sc_signal**, but only during elaboration phase

dynamic allocation of uut:

```
SC_MODULE(Testbench)
{
 Adder *uut;
 sc_signal<int> ch_x, ...
 SC_CTOR(Testbench)
  : ch_x("ch_x")
 {
  uut = new Adder("uut");
  uut->x(ch_x);
  uut->y(ch_y);
  uut->s(ch_s);
  ...
 }
 ~Testbench()
  { delete uut; } // needed!
};
```

Overriding of Methods

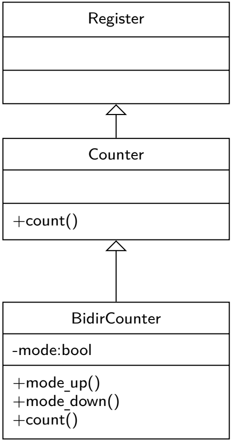



```
class BidirCounter : public Counter
{
  bool mode;
  void mode_up();
  void mode_down();
  void count(); // overriding
}
```

This method replaces the version of count() that is inherited from Counter

- Implementation of count() must be changed in bidirectional counter

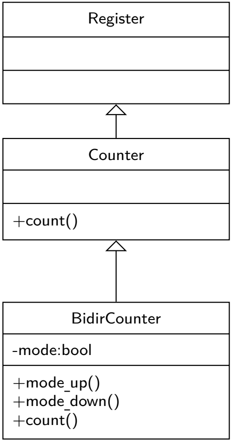
Overriding of Methods




```
void BidirCounter::mode_up()
{ mode = false; }

void BidirCounter::mode_down()
{  mode = true; }

void BidirCounter::count()
{
 if( mode )
  load( (read() - 1) % modulus);
 else
  Counter::count();
}
```

Scope resolution operator enables access to the overridden version of count()

Message Passing with Overridden Methods



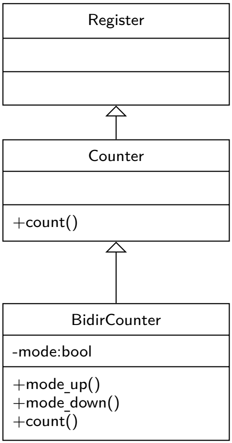

```
Register r;
Counter c;
Bidir Counter b;

Register *rp = &r;
Counter *cp = &c;
Bidir *bp = &b;

r.count();
c.count();
b.count();

rp->count();
cp->count();
bp->count();
```

Error:method count() not available for Register

Calls method count() of Counter

Calls method count() of BidirCounter

Polymorphism



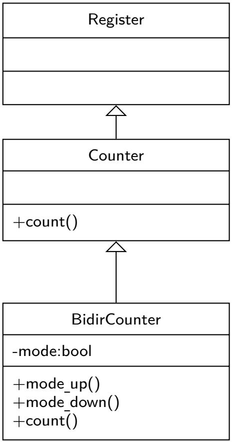

```
Counter *cp;
if(condition)
  cp = &c; // Counter object
else
  cp = &b; // Bidir Counter object
```

Polymorphism: `Counter*` may also point to object of any class derived from Counter

```
cp -> count();
```

The method to which a message is bound is statically selected by the compiler, here: `Counter::count()`

Polymorphism

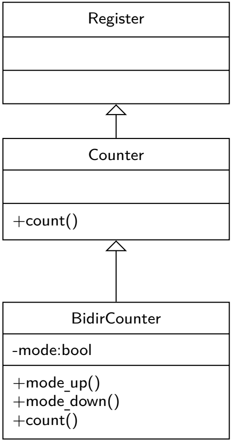



```
BidirCounter *bp = new BidirCounter;
bp->mode_down();

Counter *cp = bp;
cp->count();
```

Counter::count() is executed although cp points to a BidirCounter; counts **upwards** although **mode_down()** has been selected

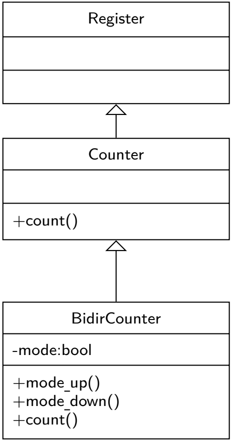
Solution: **virtual methods**

Virtual Methods




```
class Counter : public Register
{
  virtual void count();
  ...
};

class BidirCounter : public Counter
{
  bool mode;
  void mode_up();
  void mode_down();
  virtual void count();
};
```

Dynamic Binding

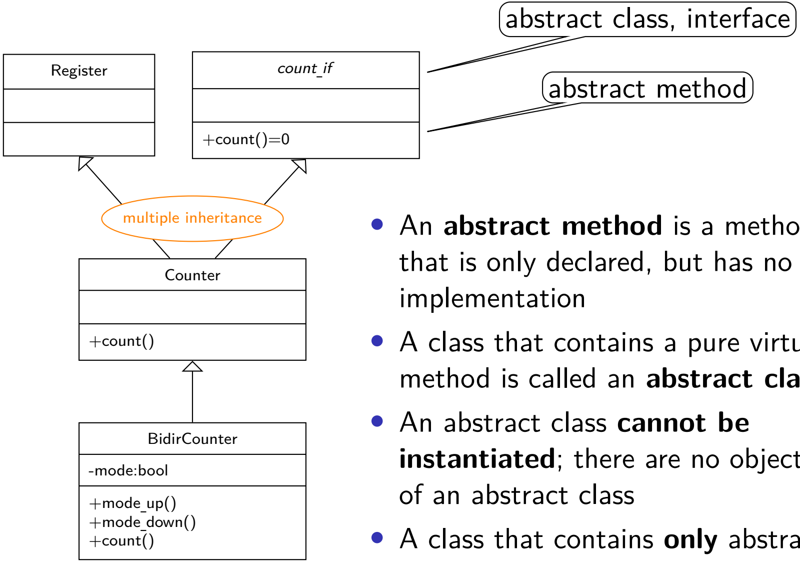



```
Counter *cp;
if(condition)
  cp = &c; // Counter object
else
  cp = &b; // BidirCounter object

cp->count();
```

Dynamic binding: for **virtual** methods, the method to which a message is bound is selected dynamically, at run-time.

cp points to Counter → Counter::count()
cp points to BidirCounter → BidirCounter::count()

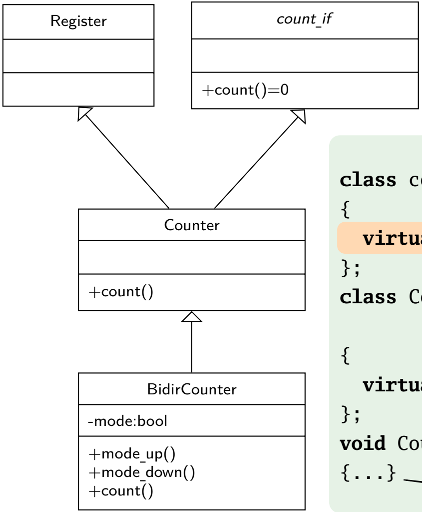
Abstract Methods and Classes



- An **abstract method** is a method that is only declared, but has no implementation
- A class that contains a pure virtual method is called an **abstract class**
- An abstract class **cannot be instantiated**; there are no objects of an abstract class
- A class that contains **only** abstract methods is called an **interface**

Pure Virtual Methods

In C++, an abstract method is called **pure virtual method**




```
class count_if
{
  virtual void count() = 0;
};
class Counter : public Register
              , public count_if
{
  virtual void count();
};
void Counter::count()
{...}
```

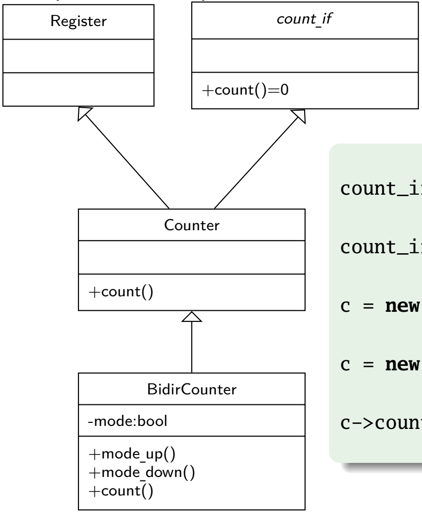
pure virtual

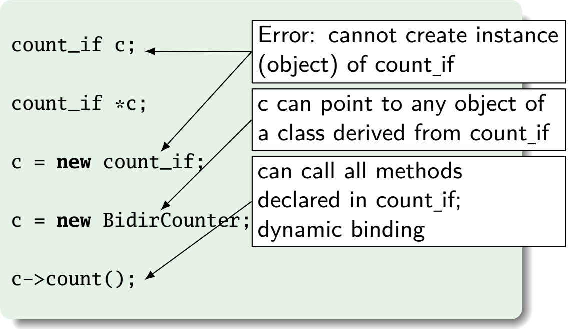
overriding

Counter must implement count()

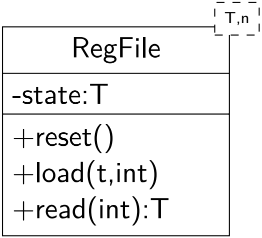
Usage of Abstract Classes

Use abstract classes to handle objects that can do similar things (here: count) in a uniform way





Templates



- **Parameterisation** of classes
- Multiple parameters possible
- Parameter can be a type or a value
- All C++ code must be in header file due to language limitations

```
template<class T, int n>
class RegFile
{
private:
  T state[n];
public:
  void reset()
   { for(int i=0; i<n; i++)
     state[i] = 0; }
  void load(T d, int i)
   { state[i] = d; }
  T read(int i) const
   { return state[i]; }
};
```

Templates

```
template<class T, int n>
class RegFile
{
private:
  T state[n];
public:
  void reset();
  void load(T d, int i);
  T read(int i);
};
...
template<class T, int n>
RegFile<T,n>::load(T d, int i)
{
  state[i] = d;
}
```

syntax for separate method implementation

implementation must still be located in the header file

Using Templated Classes

rf1 :RegFile<int, 16>

```
RegFile<int,16> rf1;
rf1.load(42,4)
```

- Object rf1 is a register file with **16** registers of type **int**

rf2 :RegFile<float, 16>

```
RegFile<float,8> rf2;
rf2.load(42.0,4)
```

- Object rf2 is a register file with **8** registers of type **float**