

## Uses of virtual functions

- Virtual functions can be used to define *interfaces* independently from *implementations*.

```
class bufdef // abstract base class
{ public:
  virtual int put(char x) = 0;
  virtual char get() = 0;
  // These are "pure" virtual functions: they
  // have no bodies.
  // Note: no constructor!
};
```

Note: No instances of `bufdef` can ever be created; its only use is as **base** for derived classes (like *mammal*).

It is a pure **interface** definition.

Clients can use `bufdef` as in

```
void copy( bufdef& b )
{ ... b.put(c); ...
  ... x = b.get(); ...
}
```

When an application of `copy` is type-checked, an argument of any class derived from `bufdef` which defines `put` and `get` is OK.

So `copy` is a *generic* function, but not fully polymorphic—its argument's type must be derived from `bufdef`.

Classes derived from `bufdef` add implementations:

```
class fixbuf : public bufdef
{ public:
  fixbuf(){ ... }
  int put(char); // put and get have the same signatures
  char get(); // as their virtual counterparts.
  private:
  ...
};
```

```
class linkbuf : public bufdef
{ public:
  linkbuf(){ ... }
  int put(char);
  char get();
  private:
  ...
};
```

```
void main()
{ fixbuf f; linkbuf g;
  copy (f); // uses put and get of fixbuf
  copy (g); // uses put and get of linkbuf
}
```

☞ `fixbuf`  $\subset$  `bufdef` and `linkbuf`  $\subset$  `bufdef`, i.e., each has all of `bufdef`'s public members.

Hence an instance of either `fixbuf` or `linkbuf` can satisfy any requirement for an instance of `bufdef`.

Java has a separate kind of declaration for interfaces:

```
public interface bufdef {
  int put( char x );
  char get();
}
...
class fixbuf implements bufdef {
  fixbuf(){ ... }
  public int put( char x ){ ... }
  public char get(){ ... }
}
...
class linkbuf implements bufdef {
  linkbuf(){ ... }
  public int put( char x ){ ... }
  public char get(){ ... }
}
```

In Haskell, interfaces can be defined using type classes:

**class** Bufdef a **where**

```
put :: (a,Char) -> Int
get :: a -> Char
```

**instance** BufDef fixbuf **where**

```
put (b,c) = ...
get b = ...
```

**instance** BufDef linkbuf **where**

```
put (b,c) = ...
get b = ...
```

```
copy :: Bufdef a => a -> ()
copy b = ... put (b,c) ... x = get b ...
```

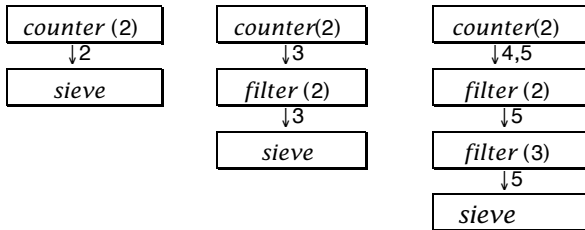
A C++ abstract base class can have some implemented methods (i.e., member functions). Like a Haskell class's default definitions, these could be inherited by subclasses/instance types.

In both languages, they would be overridden by definitions in

- derived classes (i.e., subclasses) in C++
- instance declarations in Haskell

## 2. Virtual functions can be used to alter a binding from one member function to another **during execution**.

Example: Prime numbers again, this time in C++.



First attempt:

```

class Counter
{ int value;
public:
  Counter(int v){ value = v; }
  int out(){ return value++; }
};

class Filter
{ int factor;
  ??? *source;
public:
  Filter(??? *src, int f)
  { source = src; factor = f; }
  int out();
};
  
```

```

class Sieve
{ ??? *source;
public:
  Sieve(??? *src) { source = src; }
  int primes();
};

void main()
{ Counter c(2);
  Sieve s(&c);
  for(;;)
  { int next = s.primes();
    cout << next << ", ";
  }
}

int Filter::out()
{ for(;;)
  { int n = source->out();
    if (n % factor) return n;
  }
}

int Sieve::primes()
{ int n = source->out();
  source = new Filter(source,n);
  return n;
}
  
```

Problem: what type is '???' ?

- sometimes Counter
- sometimes Filter.

A solution using class derivation:

```

class Emitter
{
public:
  virtual int out()=0; // place-holder
};

class Counter : public Emitter
{ int value;
public:
  Counter( int v ){ value = v; }
  int out(){ return value++; }
};

class Filter : public Emitter
{
  int factor;
  Emitter *source;
public:
  Filter( Emitter* src, int f )
  { source = src; factor = f; }
  int out();
};

int Filter::out()
{
  for(;;)
  { int n = source->out();
    if( (n % factor) > 0 ) return n;
  }
}
  
```

```

class Sieve
{
  Emitter* source;
public:
  Sieve( Emitter* src ){ source = src; }
  int primes();
};

int Sieve::primes()
{
  int n = source->out();
  source = new Filter( source, n );
  return n;
}

void main()
{
  Counter c(2);
  Sieve s( &c );
  for(int k = 0; k < 100; k++ )
  {
    int next = s.primes();
    cout << next << ", ";
  }
}
  
```