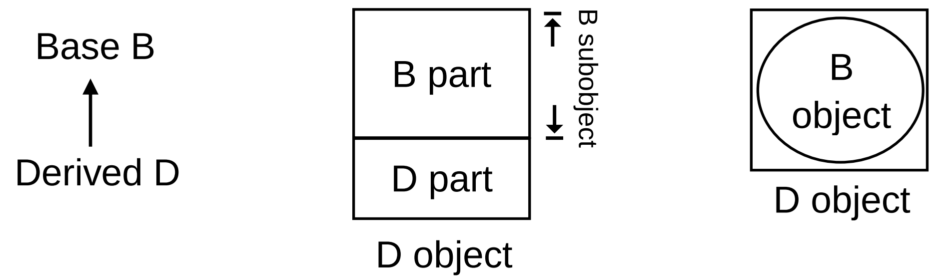
Lecture – Inheritance and OOP

Inheritance

- Inheritance is an ingredient of object-oriented programming. Programming with the class facility alone is called object-based programming.
- Inheritance by itself avoids code duplication.



Member access control

- Access control for a class

1	private	by members and friends of the class
2	protected	by members and friends of the class + members and friends of derived classes
3	public	by everyone

- Access control for a base class

1	private	public/protected of B $\rightarrow$ private of D
2	protected	public/protected of B $\rightarrow$ protected of D
3	public	public/protected of B $\rightarrow$ public/protected of D

  The private members of B remain inaccessible to D unless D is a friend of B.

- Example

```
class deque {
public:
// ctor/copy ctor/copy assignment/dtor
   deque();
   deque(const deque&);
   ~deque();
   deque& operator=(const deque&);
// capacity
   bool empty() const;
// modifiers
   void push_front(int);
   void push_back(int);
   void pop_front();
   void pop_back();
// element access
   int& front();
   const int& front() const;
   int& back();
   const int& back() const;
private:
   struct node {
      node(int,node*,node*);
      int datum;
      node *pred,*succ;
   };
   node* head;          // point to the header node of
};                      // a doubly linked list

class stack : private deque {
public:
   void push(int);
   void pop();
   int& top();
   const int& top() const;
   bool empty() const;
};
private: deque d;       // layering
```

deque
↑ private
stack

- Example (Cont'd)

```
                                          d.push_back(n);
void stack::push(int n) { push_back(n); }
void stack::pop() { pop_back(); }
int& stack::top() { return back(); }
const int& stack::top() const { return back(); }
bool stack::empty() const { return deque::empty(); }
                                                   d.empty()
```

Private inheritance means "is-implemented-by"
Private inheritance inherits implementation only.

In this example, a stack is implemented by a deque.

stack object   stack object

Special member functions

Clearly, we don't need to define a ctor for the **`stack`** class by ourselves, because

1) a **`stack`** object contains only a **`deque`** subobject that can only be initialized by a **`deque`**'s ctor, and
2) the **`deque`** class has only a default ctor

We also needn't define a dtor, a copy ctor, and a copy assignment operator for the **`stack`** class by ourselves, because the **`stack`** class doesn't allocate dynamic storage.

The special member functions for the **`stack`** class, if implicitly generated, look like:

```
// ctor
inline stack::stack() {}
```

which is equivalently to

```
                         d()
inline stack::stack() : deque() {}
```

- Example (Cont'd)

// dtor

```
inline stack::~stack() {}      d.~deque()
```

Before it returns, **`deque::~deque()`** is invoked automatically.

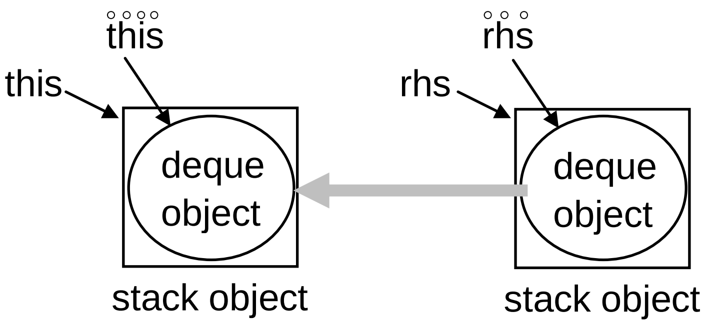
// copy ctor

```
inline stack::stack(const stack& rhs)
:  deque(rhs)                  d(rhs.d)
{}          implicit upcast    no conversion involved

   deque::deque(const deque& rhs);
```

// copy assignment operator

```
inline stack& stack::operator=(const stack& rhs)
{                              d=rhs.d
   if (this!=&rhs) deque::operator=(rhs);
   return *this;
}                          implicit upcast

   deque& deque::operator=(const deque& rhs);
```

In single-inheritance, the upcast only converts the type of the actual parameter from **`const stack`** to **`const deque&`**. The actual and formal parameters refer to the same beginning memory location of the **`deque`** subobject and the **`stack`** object.



The upcast is applied most often to the implicit object parameter.

```
stack::stack(stack* this,const stack& rhs)
:  deque(this,rhs)
{}              implicit upcast

   deque::deque(deque* this,const deque& rhs);
```

Special member functions (revisited)

- Default ctor
  Perform default initialzation of its subobjects (of class type).
  Direct base classes first, and then nonstatic data members
- Copy ctor
  Perform memberwise copy of its subobjects.
  Direct base classes first, and then nonstatic data members
- Copy assignment operator
  Perform memberwise assignment of its subobjects.
  Direct base classes first, and then nonstatic data members
- Dtor
  Call dtors for members and direct bases in the reverse order of their construction.

Member initializer list (revisited)

- Member initializer list

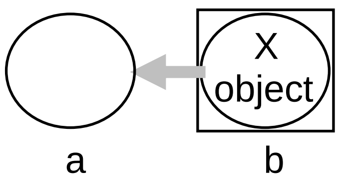
  ```
  X::X(...) : mem_id(exp), ... {}
  ```

  where **`mem_id`** must name one of
  1) a nonstatic data member
  2) a direct base, and
  3) a virtual base

  If a (direct or virtual) base class is not named by a **`mem_id`** in the initializer list, it is default-initialized.

- Initialization order of bases and members
  1 Virtual base classes are initialized in certain order (see later)
  2 Direct base classes are initialized in declaration order
  3 Nonstatic data members are initialized in declaration order
  4 The body of the ctor is executed

Initialization (revisited)

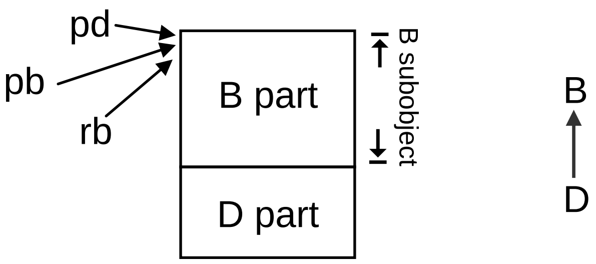


- Direct initialization **`X a(b);`**
  Call the appropriate constructor, if possible
- Copy initialization **`X a=b;`**
  Call the appropriate ctor, if **`b`**'s type is the same class as, or <u>a derived class of</u>, **`X`**; otherwise, create a temporary as before.

Upcast

- Derived-to-base conversion, **D* → B*** or **D → B&**

```
D* pd=new D;
B* pb=pd;
B& rb=*pd;
```



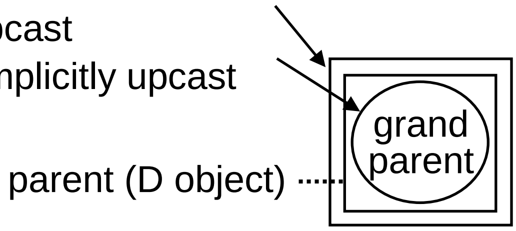
D object

- An implicit upcast can be done if the base class is
  1 accessible (i.e. if an invented public member of the base class is accessible), and
  2 unambiguous (in the presence of multiple inheritance)

  Where an implicit upcast is allowed, **static_cast** may be used explicitly.

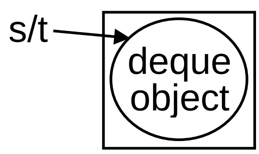
- Access control for a base class (revisited)

1	private	members and friends of D may implicitly upcast
2	protected	members and friends of D + members and friends of <u>derived classes of D</u> may implicitly upcast
3	public	everyone may implicitly upcast



- Example

```
void p()
{
    deque* s=new stack;      // stack* → deque*
    deque& t=*new stack;     // stack → deque&
    s->push_front(7);
    t.push_front(7);
}
```



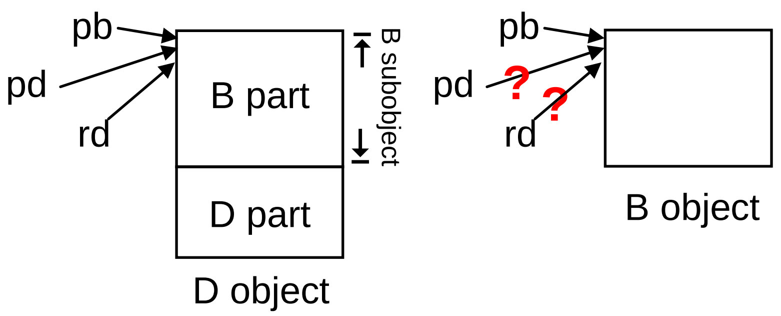
stack object

Both upcasts are disallowed.
Were they allowed, one could access the private member **push_front()** of the class **stack** through **s** or **t**.
To break down the protection, use **reinterpret_cast** or C-style cast.

Downcast

- Base-to-derived conversion, **`B*` → `D*`** or **`B` → `D&`**

```
B* pb=new D;
D* pd=pb;
D& rd=*pb;

B* pb=new B;
D* pd=pb;    ?
D& rd=*pb;   ?
```



- There is no implicit downcast.
  Use **`static_cast`** or C-style cast for explicit downcast. It is valid when the corresponding upcast is valid and when the B object is actually a subobject of a D object. Otherwise, the result is undefined.

- Example

  Another way of defining the copy assignment operator of the **`stack`** class:

```
inline stack& stack::operator=(const stack& rhs)
{
   if (this==&rhs) return *this;
   return
   static_cast<stack&>(deque::operator=(rhs));
}
                      const stack → const deque&
                deque → stack&
     deque& deque::operator=(const deque& rhs);
```

  Cf. the definition given earlier

```
inline stack& stack::operator=(const stack& rhs)
{
   if (this!=&rhs) deque::operator=(rhs);
   return *this;
}
```

Member name lookup

- The name **`f`** in **`exp.f`** or **`exp.A::f`** is looked up in the static type of **`exp`** or the qualified type **`A`** (which must be a base type of the static type of **`exp`**), respectively.
  (In case **`exp`** is missed, **`*this`** is assumed.)

- When looking up a name **`f`** in a class scope, any declaration of **`f`** that is hidden is eliminated from consideration.

```
B f           [ B f
 ↑            |  [ D f
 |            |  |       D::f hides B::f
D f           |  |
```

- Example

```
deque   operator=(const deque&)        [ deque
  ↑     push_back                      |  [ stack
  |                                    |  |
stack   operator=(const stack&)        |
        push
```

```
void stack::push(int n) { this->push_back(n); }

inline stack& stack::operator=(const stack& rhs)
{
   if (this!=&rhs) this->deque::operator=(rhs);
   return *this;
}
```

Note that **`deque::operator=`** and **`stack::operator=`** are not overloaded, as they are defined in distinct scopes.

Thus, inside the definition of **`stack::operator=`**, both

```
   operator=(rhs)
```

and

```
   operator=(static_cast<const deque&>(rhs))
```

resolve to a recursive call. But, of course, the latter results in a compile error, for it demands an implicit downcast.

- Example

Stack application – Combination generation

```
int c(int n,int k)
{
   static stack s;
   if (k==0||n==k) {
      cout << s;
      for (int i=k;i>=1;--i) cout << i;
      cout << endl;
      return 1;
   } else {
      s.push(n); int r=c(n-1,k-1); s.pop();
      return r+c(n-1,k);
   }
}
```

Attempt 0 – Doesn't work. Friendship isn't transitive.

deque (private data member, friend class stack)

↑ private

stack (friend operator<< for stack)

Method 1 – For illustration only (add an operation to the base)

deque (private data member, friend operator<< for deque)

↑ private

stack (friend operator<< for stack)

```
ostream& operator<<(ostream& os,const deque& d)
{
   deque::node* p=d.head->succ;
   while (p!=d.head) {      // deque's friend may access
      os << p->datum;       // private data
      p=p->succ;
   }
   return os;
}
```

- Example (Cont'd)

```
ostream& operator<<(ostream& os,const stack& s)
{
   return os << static_cast<const deque&>(s);
}
```

// stack's friend may upcast

Note that these two operators are overloaded, as they are both defined in the global scope.

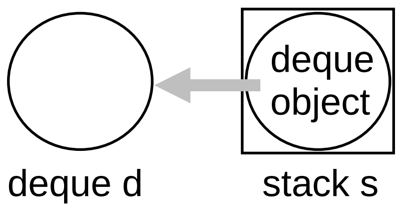
Method 2 – For illustration only (send out a spy to the base)

deque (private data member, friend operator<< for stack)

↑ private

stack (friend operator<< for stack)

```
ostream& operator<<(ostream& os,const stack& s)
{
   const deque& d=s;          //* stack's friend may upcast
   deque::node* p=d.head->succ;
   while (p!=d.head) {        // deque's friend may access
      os << p->datum;         // private data
      p=p->succ;
   }
   return os;
}
```



Don't write the starred line as

```
deque d=s;
```

or

```
deque d(s);
```

stack → deque

// stack's friend may upcast

as it will invoke **deque**'s copy ctor   stack → deque&

```
deque::deque(const deque& rhs);
```

to copy the **deque** subobject of **s** to **d**. Although it works, it wastes time to construct and then destruct a **deque** object.

- Example (Cont'd)

Method 3

deque (protected data member)
↑ private
stack (friend operator<< for stack)

```
ostream& operator<<(ostream& os,const stack& s)
{
//  if (s.deque::empty()) return os;  // upcast to parent
    deque::node* p=s.head->succ;       // no upcast
    while (p!=s.head) {
        os << p->datum; p=p->succ;
    }
    return os;
}
```

Drawback – Not every client needs to display the contents of a stack.

Method 4 – Printable stack

deque (protected data member)
↑ protected
stack — stackP and its friends may access deque's public/protected members
↑ public
stackP (friend operator<< for stackP)

```
class deque {
public: ...
protected:
    struct node;
    node* head;
};
class stack : protected deque { ... };
class stackP : public stack {
friend ostream& operator<<(ostream&,const stackP&);
};
```

- Example (Cont'd)

```
ostream& operator<<(ostream& os,const stackP& s)
{
//   if (s.deque::empty())        // upcast to grandparent
//      return os;
   deque::node* p=s.head->succ;          // no upcast
   while (p!=s.head) {
      os << p->datum; p=p->succ;
   }
   return os;
}

int c(int n,int k)
{
   static stackP s;
   … cout << s; … s.push(n); …
}
```

Remarks
A printable stack is a stack.
A stack is implemented by a deque.

Private/protected inheritance means "is-implemented-by"
Private inheritance inherits implementation only.
Protected inheritance inherits implementation that may further be inherited.

Public inheritance means "isa"
Public inheritance inherits interface as well as implementation.

If class D publicly inherits from class B, then

1) B is more general than D (or D is more specialized than B)
2) every D object isa B object (due to implicit upcast), but *not vice versa*.
   In other words, everything that is applicable to B objects is also applicable to D objects. For example,

   `void p(B&);` or `void p(B);`

   can be invoked by

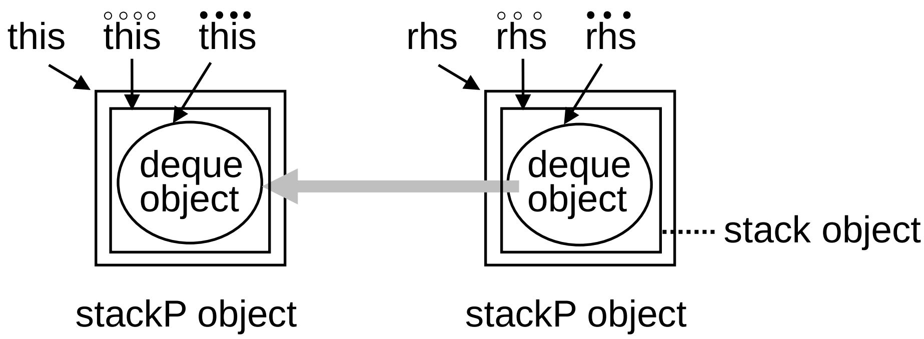
   `D d; p(d);`

- Example (Cont'd)

  Special member functions

  It should now be clear that implicitly generated special member functions suffice for the **`stackP`** class (and the **`stack`** class). For examples,

```
// copy ctor
inline stackP::stackP(const stackP& rhs)
:  stack(rhs)
{}                                  implicit upcast

inline stack::stack(const stack& rhs)
:  deque(rhs)
{}                      implicit upcast

deque::deque(const deque& rhs)
: head(...) { ... }        // create a doubly linked list
```

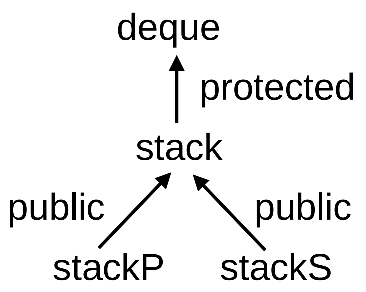



```
// dtor
inline stackP::~stackP()  { }
                                      call it and then return
inline stack::~stack()  { }
                                   call it and then return
inline deque::~deque() { … }   // destroy the doubly
                               // linked list
```

- Example – Summable stack

  Stack application – Divisible group sums

  Determine the number of groups of *k* integers, chosen from an array of *n* integers, whose sum is divisible by integer *d*.

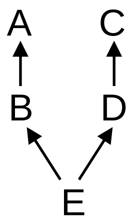


// A summable stack is a stack.

```
class stackS : public stack {
public:
   int sum();
};
int stackS::sum()
{
   int sum=0;
   node* p=head->succ;
   while (p!=head) { sum+=p->datum; p=p->succ; }
   return sum;
}
int dgs(int* a,int n,int k,int d)
{
   static stackS s;
   if (k==0||n==k) {
      int sum=s.sum();
      for (int i=0;i<k;i++) sum+=a[i];
      return sum%d==0;
   } else {
      s.push(a[n-1]);
      int r=dgs(a,n-1,k-1,d);
      s.pop();
      return r+dgs(a,n-1,k,d);
   }
}
```

Multiple inheritance

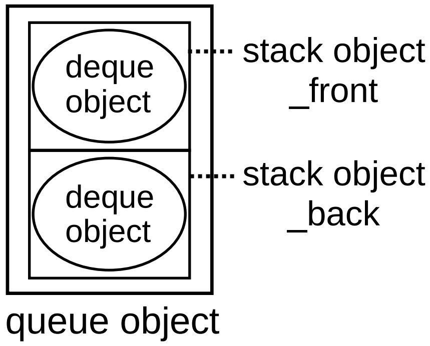
- A class can have one or more direct base class.
  The use of one direct base class is called single inheritance.
  The use of more than one direct base class is called multiple inheritance.



- A class cannot be a direct base class more than once.
  A class can be an indirect base class more than once.
  A class can be a direct and an indirect base class.
- The order of derivation determines the execution order of ctors and dtors.
- Example – Queue as a pair of stacks

  Method 1 – Layering (Recall from Lecture on Class and ADT)

```
class queue {
public:
   void push(int);
   void pop();
   int& front();
   const int& front() const;
   bool empty() const;
private:
   void _check();
   stack _front,_back;
};
```



Layering (composition, containment, embedding) is the process of building a class on top of another class.

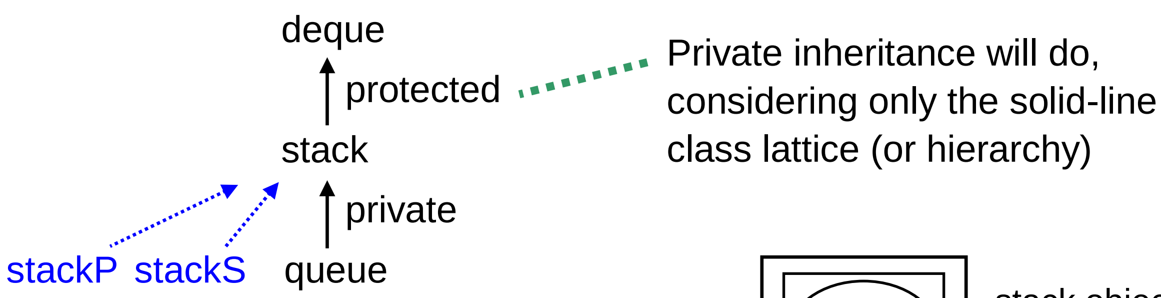
Layering means either "has-a" or "is-implemented-by".

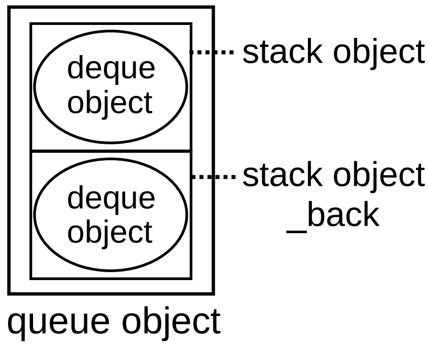
For example, a course has a name, an instructor, and at most 200, say, students:

```
class course {
public: ...
private:
   string name,instructor,students[200];
};
```

- Example (Cont'd)

Method 2 – Layering + Single inheritance






```
class queue : private stack {
public: ...
private:
   void _check();
   stack _back;
};
```

```
int& queue::front() { return top(); }
const int& queue::front() const { return top(); }
bool queue::empty() const { return stack::empty(); }
void queue::push(int n) { _back.push(n); _check(); }
void queue::pop() { stack::pop(); _check(); }
void queue::_check()
{
   if (stack::empty())
      while (!_back.empty()) {
         stack::push(_back.top()); _back.pop();
      }
}
```

Selected special member functions

```
inline queue::queue() : stack(), _back() {}
```

Direct base is constructed first and then nonstatic data member

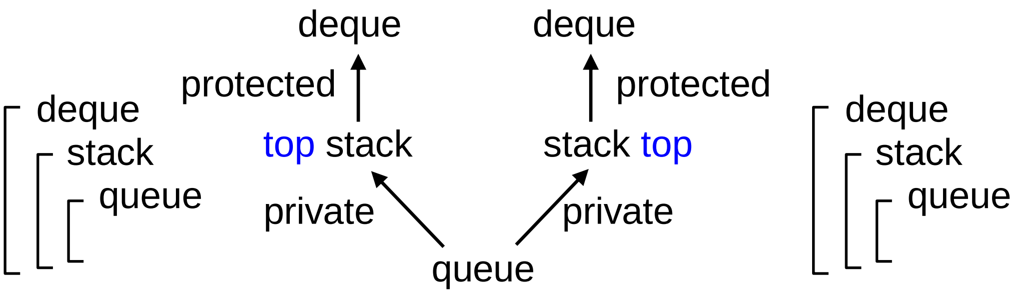
```
inline queue::~queue() {}
```

Before it returns,
`_back.~stack()` and `this->stack::~stack()`
are called in order.

- Example (Cont'd)

  Method 3 – Multiple inheritance

  Attempt 1

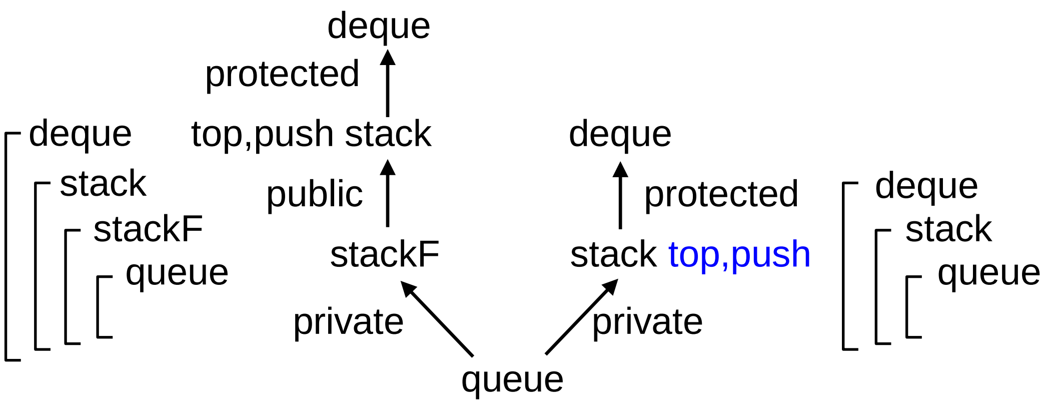



```
class queue : private stack, private stack {
public:
   int& front() { return top(); }
                 :
};
```

Ambiguous! Which **top?**

This class hierarchy is illegal, as there is no way to distinguish member function calls of one stack subobject from another.

Attempt 2



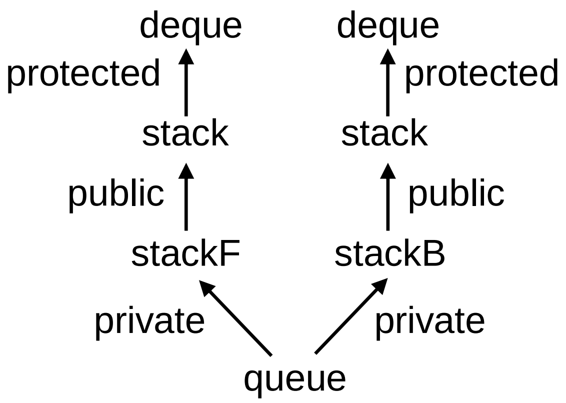

```
class stackF : public stack {};
class queue : private stackF, private stack {
public:
   int& front() { return stackF::top(); }
   void push(int n) { push(n); _check(); }
             :
};
```

Ambiguous!
How to access **stack::push** of the direct base?

This class lattice is legal; but a compiler usually warns you of the inaccessibility of the direct base **stack**.

- Example (Cont'd)

Attempt 3 (Done)



The classes **`stackF`** and **`stackB`** serve as stepping stones in the lattice and shall be protected against the outsider.

Try 1

```
class stackF: protected stack {};
class stackB: protected stack {};
stackF s;          // Ok, invoke the implicitly generated
                   // default ctor
s.push(3);         // No, push is protected
```

This is incorrect for two reasons.

1 It is conceptually incorrect, since **`stackF`** and **`stackB`** are not implemented by **`stack`**.

2 It only protects "non-special" member functions.
Implicit defined special member functions are still public.
Note: Special member functions are not inherited.

Try 2

```
class stackF: public stack {
protected:
   stackF() : stack() {}
};
stackF s;          // No, cannot create a stackF object by
                   // the protected default ctor
s.push(3);         // No, s doesn't exist.
```

- Example (Cont'd)

Even if the default ctor is protected, a client can still create and manipulate standalone **stackF** objects by other implicitly generated special member functions.

```
queue q;
stackF s(reinterpret_cast<stackF&>(q));
s=s;
s.~stackF();
```

The following design prohibits the manipulation of standalone **stackF** objects, but not of embedded **stackF** objects such as

```
reinterpret_cast<stackF&>(q).push(2);     // (*)
```

If we want to bar (*), we have to resort to protected inheritance. But, this is conceptually incorrect. In this regard, we shall treat (*) as a penalty of using **reinterpret_cast**.

Conclusion – Protect all special member functions

```
class stackF: public stack {              // isa
protected:
   stackF() : stack() {}
   stackF(const stackF& rhs) : stack(rhs) {}
   ~stackF() {}
   stackF& operator=(const stackF& rhs)
   {
      if (this!=&rhs) stack::operator=(rhs);
      return *this;
   }
};
```

The class **stackB** is defined in a similar manner.

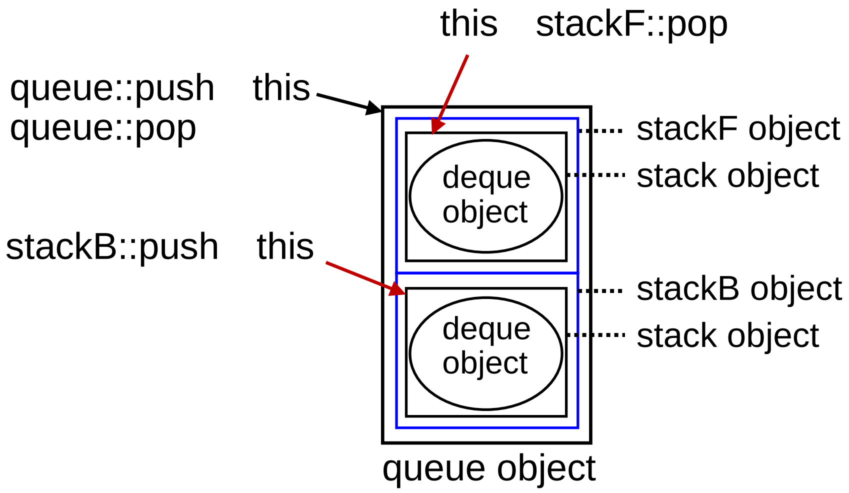
Upcast in multiple inheritance

```
class queue : private stackF, private stackB {
public: ...
private:
   void _check();
};
```

- Example (Cont'd)

```
void queue::push(int n)
{  stackB::push(n); _check(); }

void queue::pop()
{  stackF::pop(); _check(); }
```

and so on



In multiple-inheritance, an upcast may have to adjust the pointer value.
To see why, observe that the **queue** object and the **stackF::stack** object share the same beginning address, but the **stackB::stack** object has different beginning address. Thus,

queue* → **stackF::stack*** no adjustment needed
queue* → **stackB::stack*** adjustment needed

Selected special member functions

// default ctor
```
inline queue::queue() : stackF(), stackB() {}
```
Direct base classes are initialized in declaration order.

// dtor
```
inline queue::~queue() {}
```
Before it returns,
this->stackB::~stackB() and **this->stackF::~stackF()** are called in order.

- Example (Cont'd)

Layering vs private/protected inheritance

Use private/protected inheritance to model "is-implemented-by" whenever there are protected members and/or virtual functions; otherwise, use layering.

```
class deque {
protected:
   node* head;
};

class stack : protected deque {
public:
   int& top()
   {
      return head->pred->datum;      // i.e. back()
   }
};

class stack {                        // has to be d.back()
public:
   int& top() { return d.head->pred->datum;  }
private:
   deque d;
};
```

The former is OK, but the latter isn't – only inheritance gives access to protected members.

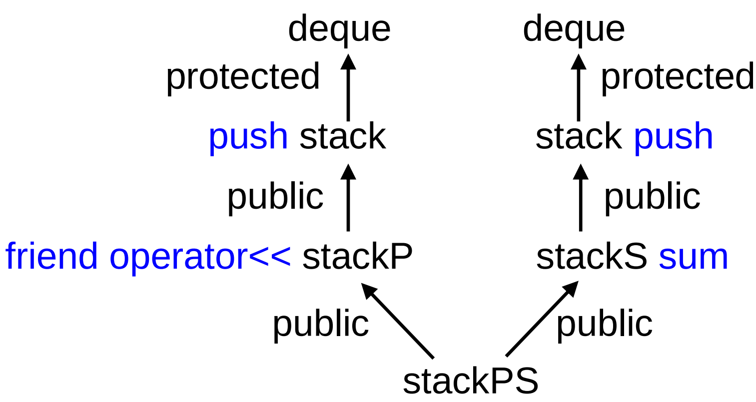
Virtual inheritance

- A base class specified with the keyword **`virtual`** is a virtual base class; otherwise, it is a nonvirtual base class.
- Each *occurrence* of a nonvirtual base class B in the class lattice of the most derived class corresponds to a B subobject within the most derived object.
- Each virtual base class B corresponds to a single B subobject within the most derived object.
- Example

  Stack application – Enumerated divisible group sums

  Determine not only the number of groups but also the groups themselves that satisfy the requirement.

  Method 1



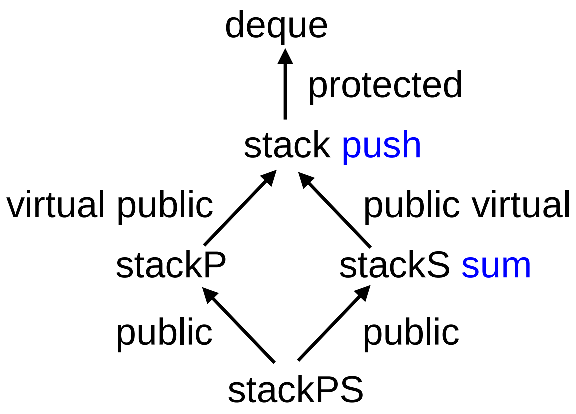

```
class stackPS : public stackP, public stackS {};
```

The most derived **`stackPS`** object *undesirably* contains two distinct **`stack`** subobjects. Nonetheless, it is worthy to investigate member name lookup:

`stackPS s;`		
`s.push(a[n-1]);`	X	Ambiguous
`s.stackP::push(a[n-1]);`	O	
`s.stackS::push(a[n-1]);`	O	
`s.sum()`	O	**`stackS::sum`**
`cout << s;`	O	implicitly upcast to **`stackP`**

- Example (Cont'd)

Method 2

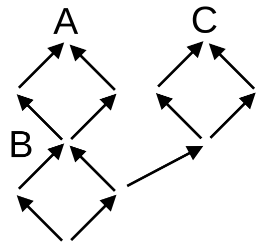



```
class stackP : virtual public stack {
friend ostream& operator<<(ostream&,const stackP&);
};

class stackS : virtual public stack {
public: int sum();

};

int dgs(int* a,int n,int k,int d)
{
   static stackPS s;
   if (k==0||n==k) {
      int sum=s.sum();
      for (int i=0;i<k;i++) sum+=a[i];
      if (sum%d!=0) return 0;
      else {
         for (int i=0;i<k;i++) cout << a[i];
         cout << s;
         return 1;
      }
   } else {
      s.push(a[n-1]);
      int r=dgs(a,n-1,k-1,d);
      s.pop();
      return r+dgs(a,n-1,k,d);
   }
}
```

Special initialization semantics

- Virtual base classes are initialized first, only for the ctor of the most derived class, in the order of depth-first, left-to-right traversal of the class lattice (i.e. the directed acyclic graph (DAG)).

- Example (Cont'd)

  Here are the implicitly generated ctors:

```
stack::stack()     : deque()  {}
stackP::stackP()   : stack()  {}
stackS::stackS()   : stack()  {}
stackPS::stackPS()  : stack(),stackP(),stackS(){}
```

  Note that in a nonvirtual inheritance, a derived class can only initialize its direct base classes. However, in a virtual inheritance, a derived class can initialize its indirect virtual base class

  As shown, all of the last three ctors contain a call to initialize the virtual base **`stack`**. However, only the most derived class will activate the call.

  Consider the following declarations:

  `stackP s;`

  `stackP` is the most derived class. The order of ctor calls is:
  `stackP();`
  `stack();` // called from **`stackP()`**

  `stackS s;`

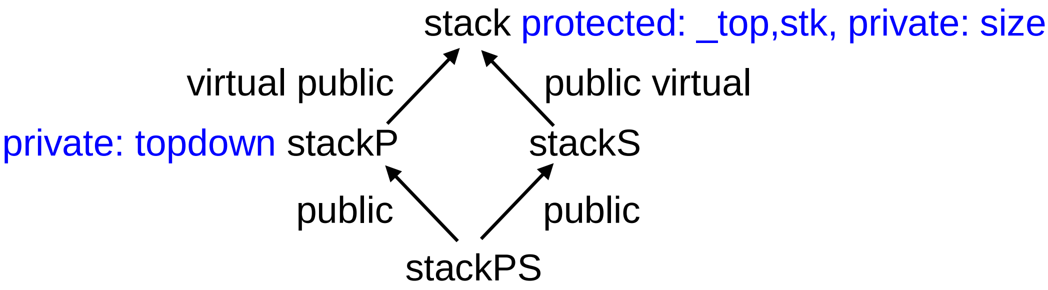
  `stackS` is the most derived class. The order of ctor calls is:
  `stackS();`
  `stack();` // called from **`stackS()`**

  `stackPS s;`

  `stackPS` is the most derived class. The order of ctor calls is:
  `stackPS();`
  `stack();` // called from **`stackPS()`**
  `stackP();` // call of **`stack()`** is suppressed automatically
  `stackS();` // call of **`stack()`** is suppressed automatically

- Example

For this example, let's assume that a stack is implemented by a (semi-)dynamic array of a specified size, and that the elements of a stack can be printed top-to-bottom or bottom-to-top.



Method 1

```
stack::stack(int n)
: _top(-1), stk(new int[n]), size(n) {}

stackP::stackP(int n,bool td)
: stack(n), topdown(td) {}

stackS::stackS(int n) : stack(n) {}

stackPS::stackPS(int n,bool td)
: stack(n), stackP(n,td), stackS(n) {}
```

Drawback: The **`stackPS`**'s ctor passes the argument **`n`** to the ctors of **`stackP`** and **`stackS`** unnecessarily.

Method 2

For classes **`stack`**, **`stackP`**, and **`stackS`**, besides the original public ctors (that will be invoked when the corresponding class is the most derived), three protected ctors (that will be invoked only when the **`stackPS`** class is the most derived) are added :

```
stack::stack() {}
stackP::stackP(bool td)
: stack(), topdown(td) {}
stackS::stackS() : stack() {}
```

As for class **`stackPS`**, its ctor is modified as follows:

```
stackPS::stackPS(int n,bool td)
: stack(n), stackP(td), stackS() {}
```

- Example (Cont'd)

  Here is the order of ctor calls for **`stackPS s(80,true)`**:

```
stackPS(80,true);
stack(80);
stackP(true);  // call of stack() is suppressed
stackS();      // call of stack() is suppressed
```

  Notice that even though the protected ctor **`stack::stack()`** is never called, it must be explicitly defined and accessible.

- Copy ctors, being ctors, obey the same semantics.

  Example (Cont'd)

  We need only define a copy ctor for the **`stack`** class (since it allocates storage dynamically):

```
stack::stack(const stack& rhs)
:  _top(rhs._top), stk(new int[rhs.size]),
   size(rhs.size)
{
   for (int i=0;i<=_top;i++) stk[i]=rhs.stk[i];
}
```

  and rely on the implicitly generated copy ctors for the remaining classes (since they don't allocate storage dynamically):

```
stackP::stackP(const stackP& rhs)
:  stack(rhs), topdown(rhs.topdown) {}

stackS::stackS(const stackS& rhs)
:  stack(rhs) {}

stackPS::stackPS(const stackPS& rhs)
:  stack(rhs), stackP(rhs), stackS(rhs) {}
```

  Alternatively, we may define our own copy ctor for **`stackPS`** to avoid unnecessary passing of the argument **`rhs`**.

```
stackPS::stackPS(const stackPS& rhs)
:  stack(rhs), stackP(rhs.topdown), stackS() {}
           // or, stackP(rhs)
```

Dtor and copy assignment operator

- There is nothing special for the dtor. The order of (virtual and nonvirtual) base class dtor invocation is guaranteed to be the reverse order of ctor invocation.
- The semantics of the copy assignment operator is essentially unchanged, except that it is unspecified whether the subobject representing a virtual base is assigned more than once by the implicitly defined copy assignment operator.
  (This is an efficiency issue, rather than a semantics issue.)
- Example (Cont'd)

  As usual, we need only define a copy assignment operator for the **stack** class (since it allocates storage dynamically):

```
stack& stack::operator=(const stack& rhs)
{
   if (this!=&rhs) {
      delete [] stk;
      stk=new int[rhs.size];
      _top=rhs._top;
      for (int i=0;i<=_top;i++) stk[i]=rhs.stk[i];
   }
   return *this;
}
```

  and rely on the implicitly generated copy assignment operators for the remaining classes (since they do not allocate storage dynamically):

```
stackP& stackP::operator=(const stackP& rhs)
{
   if (this!=&rhs) {
      stack::operator=(rhs);     // *
      topdown=rhs.topdown;
   }
   return *this;
}
```

- Example (Cont'd)

```
stackS& stackS::operator=(const stackS& rhs)
{
   if (this!=&rhs) stack::operator=(rhs);   // *
   return *this;
}
```

```
stackPS& stackPS::operator=(const stackPS& rhs)
{
   if (this!=&rhs) {
      stackP::operator=(rhs);
      stackS::operator=(rhs);
   }
   return *this;
}
```

Case 1: The **stack** subobject is copied once.
In this case, one of the two starred calls is suppressed in the course of executing **stackPS::operator=**.

Case 2: The **stack** subobject is copied twice.
In this case, both starred calls are executed. The net effect is the same as above, but there is a redundant copy.

Principle
For compiler-independent guaranteed efficiency, define copy assignment operators for classes with virtual bases.

In either case, we may define our own **stackPS::operator=**

```
stackPS& stackPS::operator=(const stackPS& rhs)
{
   if (this!=&rhs) stackP::operator=(rhs);
   return *this;
}
```

In case the **stackS** class has private data members, we may

1) declare them protected and modify them in **stackPS::operator=**, or
2) define a protected member function to modify them and invoke it from **stackPS ::operator=**

Virtual functions

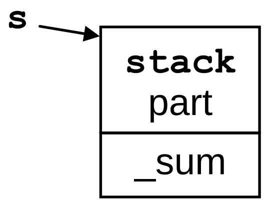
- Virtual functions support object-oriented programming.
- Example – Alternative design of the `stackS` class

```
class stackS : virtual public stack {
public:
   stackS();
   void push(int);
   void pop();
   int sum();
private:
   int _sum;
};
stackS::stackS() : stack(), _sum(0) {}
void stackS::push(int n) { _sum+=n; stack::push(n); }
void stackS::pop() { _sum-=top(); stack::pop(); }
int stackS::sum() { return _sum; }
```

Note that the `stackS` class redefines the member functions `push` and `pop` publicly inherited form the `stack` class.

As it is, this definition is problematic. To see why, let

```
stack* s=new stackS;
```



`stackS` object

and consider

```
s->push(n)
```

Shall it invoke `stack::push` or `stackS::push`?
Clearly, it should invoke `stackS::push`.
Why?

Because the object in existence is indeed a `stackS` object. Invoking `stack::push` will make it inconsistent.

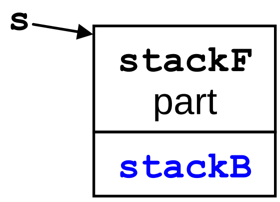
In other words, since a `stackS` object is a specialized `stack` object, it should use the specialized, rather than the general, implementation of `push`.

- Example (Cont'd)

However, as written, it will invoke **`stack::push`**, because
1) **`push`** is a non-virtual function, and
2) the static (i.e. declared) type of **`s`** is **`stack*`**

As a contrast example, let — or **`stackF::stack*`**

```
stack* s=reinterpret_cast<stackF*>(new queue);
```



`queue` object

and consider

```
s->push(n)
```

Again, shall it invoke **`stackF::push`** or **`queue::push`**?

Undoubtedly, it should invoke **`stackF::push`**, since a **`queue`** object *isn't* a **`stack`** object.

Of course, invoking **`stackF::push`** will make the **`queue`** object inconsistent; but this penalty is what we shall pay for the unsafe use of **`reinterpret_cast`**.

Comment
We won't say that **`queue::push`** redefines **`stack::push`** because the former isn't a specialized implementation of the latter.

Principle
Never redefine a publicly inherited nonvirtual member function.

A nonvirtual function specifies an invariant over specialization.

Public inheritance of a nonvirtual function inherits a function interface as well as a *mandatory* implementation.

Public inheritance of a virtual function inherits a function interface as well as a *default* implementation that may be redefined if need be.

- Example (Cont'd)

```
class stack : protected deque {
public:
    virtual void push(int);
    virtual void pop();
    int& top();
    const int& top() const;
    bool empty() const;
};
```

```
class stackS : virtual public stack {
public:
    stackS();
    virtual void push(int);     // optional
    virtual void pop();         // optional
    int sum();
private:
    int _sum;
};
```

The **`virtual`** specifiers can only appear in the declaration of nonstatic member functions.
The **`virtual`** specifiers are optional when redefining **`push`** and **`pop`** in the derived class.

```
stack* s=new stackS;
s->push(n);              // call stackS::push
stack* s=new stack;
s->push(n);              // call stack::push
```

OOP = inheritance + virtual function

Polymorphism

- A class that declares or inherits a virtual function is called a polymorphic class.
- Virtual functions support run-time polymorphism by flexible late binding (i.e. dynamic binding).
  Overloading supports compile-time polymorphism by fixed early binding (i.e. static binding).

Virtual functions

- Let vf be a virtual function declared in B, and

```
B vf          ┌ B vf
 ↑            │ ┌D vf
 │            │ │   D::vf overrides B::vf
D vf          └ └
```

  1. Any **`D::vf`** that has the same parameter list as **`B::vf`** is also virtual (whether or not it is so declared), and
  2. **`D::vf`** overrides **`B::vf`**.
     `D::vf` is the overriding function and **`B::vf`** is the overridden function.
     For convenience, we say that a virtual function overrides itself.

- A virtual function call through a pointer or reference to an object depends on the type of the object (i.e. the dynamic type).

  A non-virtual function call or a virtual function call through an object depends on the type of the expression denoting the object (i.e. the static type).

```
stack& s=*new stackS;
s.push(n);                // call stackS::push
stack s=*new stackS;      // only the stack part is copied
s.push(n);                // call stack::push
```

- Qualification suppresses the virtual call mechanism, e.g.

```
void stackS::push(int n)
{
   _sum+=n; stack::push(n);   // call stack::push
}
```

Virtual function lookup

- A virtual function call is determined in two steps:

  1 Use member name lookup to resolve the function as usual. (The **`virtual`** specifier is ignored in this step.)

  2 If the function name is unambiguously resolved, then

  2.1 If it isn't virtual or the object isn't pointed or referenced or qualified type is used, done. (Inside a member function, the object is tacitly pointed by **`this`**.)

  2.2 If it is virtual, find the unique final overrider[†] of the virtual function along the path(s)[✟] from the object's dynamic type to the class containing the resolved function.

  † Every virtual function declared or inherited in a class must have a unique final overrider; otherwise, the class is illegal.

  ✟ With single inheritance, the inheritance structure is a tree, and there is a single path.
  With multiple inheritance, the inheritance structure is a DAG, and there may be multiple paths.

- Note on the differences between step 1 and step 2.2

Step 1

Where to start	the static or qualified type
Where to search	all base types of the static or qualified type
Condition	hiding (parameters aren't considered)

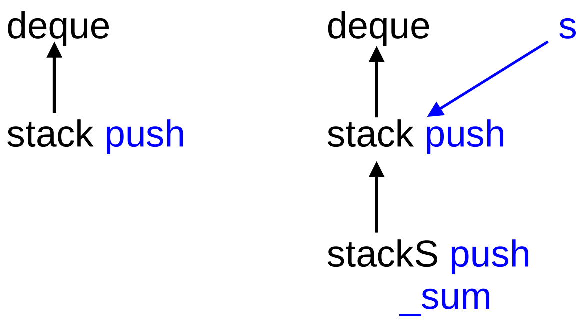
Step 2.2

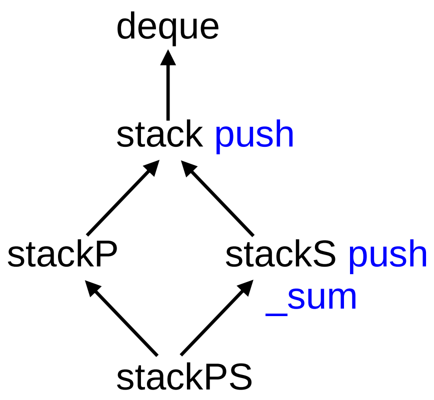
Where to start	the dynamic type
Where to search	only the path(s) from the dynamic type to the class containing the resolved function
Condition	overriding (parameters are considered)

```
B  int f;
  D
   int f();

D* pd=new D;
pd->f;      // error
```

```
B  virtual int f();
  D
   [virtual] int f(int);

B* pb=new D;
pb->f();      // B::f
```

- Example





stack* s=new stack;
s->push(n);
1 reslove to **stack::push**
2.2 call **stack::push,** the final overrider of **stack::push**

stack* s=new stackS;
s->push(n);
1 resolve to **stack::push**
2.2 call **stackS::push,** the final overrider of **stack::push**

stackS* s=new stackPS;
stackPS* s=new stackPS; ♥
s->push(n);
1 resolve to **stackS::push**
♥ **stackS::push** hides **stack::push**, even If the latter can be reached along the path **stackPS-stackP-stack**
2.2 call **stackS::push,** the final overrider of **stackS::push**

stack* s=new stackPS;
stackP* s=new stackPS; ♦ the diamond property
s->push(n);
1 resolve to **stack::push**
2.2 call **stackS::push,** the final overrider of **stack::push** (since it, being the overrider along the r.h.s path, overrides the overrider **stack::push** along the l.h.s path)

The diamond property
A call of a virtual function through one path in an inheritance structure may result in the invocation of a function redefined on another path.

- Example (Cont'd)

Inheritance via dominance

In the diamond-shaped inheritance graph, the redefinition of **`push`** in **`stackS`** is said to *dominate* the original definition in stack, because **`s->push(n)`** always calls **`stackS::push`** as long as the dynamic type of **`*s`** is **`stackPS`**.
To break down the dominance, use qualification.

A highlighted example

```
// stack: Combination, backtracking
int c(int n,int k,stack& s)
{
   … s.stack::push(n-1); s.stack::push(k); …
}
// stackP: Combination generation
int c(int n,int k,stackP& s)
{
   … cout << s; … s.stackP::push(n); …
}
// stackS: Divisible group sums
int dgs(int* a,int n,int k,int d,stackS& s)
{
   … s.sum(); … s.push(a[n-1]); …
}
// stackPS: Enumerated divisible group sums
int dgs(int* a,int n,int k,int d,stackPS& s)
{
   … s.sum(); cout << s; … s.push(a[n-1]); …
}
int main()                    // Upcasts to parent are better
{                             // than upcasts to grandparent.
   stackPS s;
   cout << c(4,2,static_cast<stack&>(s));
   cout << c(4,2,s);          // stackPS->stackP& is better
   int a[6]={1,2,3,4,5,6};
   cout << dgs(a,6,3,5,static_cast<stackS&>(s));
   cout << dgs(a,6,3,5,s);
}
```

- Example

Fig.1

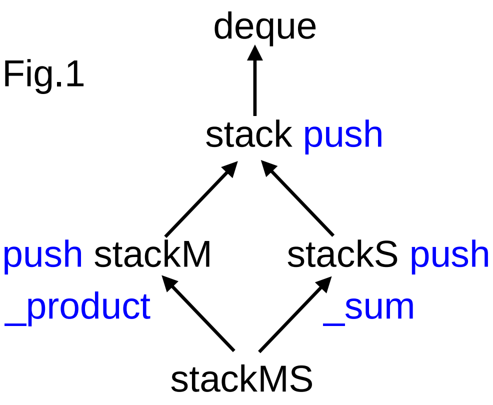
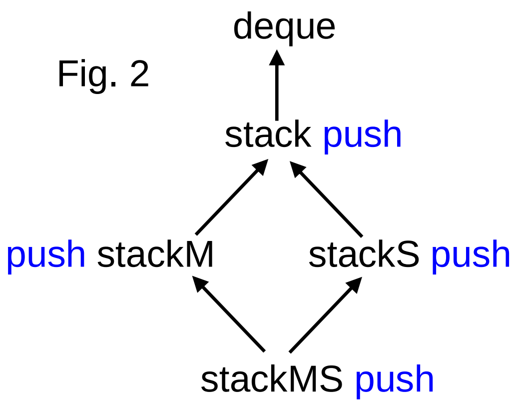


Fig. 2



The **stackM** class records the product of all stack elements. In Fig. 1, only **stackM::push** is redefined:

```
void stackM::push(int n)
{
    _product*=n; stack::push(n);
}
```

This class lattice is illegal, because not every virtual function in it has a unique final overrider.

1) **stackM::push** has itself as the unique final overrider.
2) **stackS::push** has itself as the unique final overrider.
3) **stack::push** has **stackM::push** and **stackS::push** as its final overriders.

Lesson
Be careful when redefine a virtual function along more than one path in the inheritance structure.

In this case, **stackMS::push** must also be redefined (where **_sum** and **_product** are protected):

```
void stackMS::push(int n)
{
    _product*=n; _sum+=n; stack::push(n);
}
```

The class lattice of Fig. 2 is legal, since every virtual function in it has **stackMS::push** as its final overrider.

Virtual destructor

- If the base class's dtor is nonvirtual, the result of deleting a derived class object through a base class pointer is undefined.

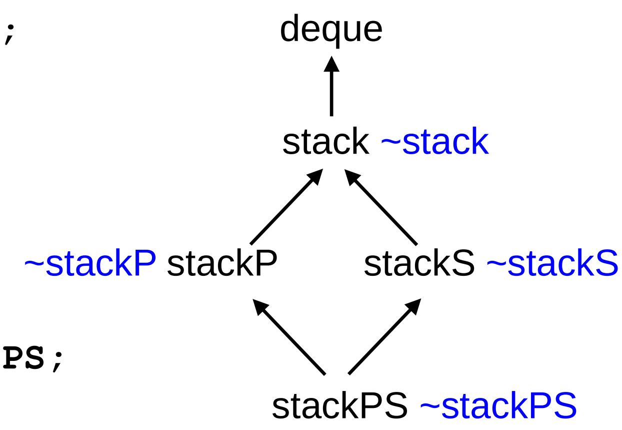
- Example

  ```
  stackP* s=new stackPS;
  delete s;     // undefined, since the dtors are nonvirtual
  ```

  Usually, `~stackP()` is called, but `~stackPS()` is expected.

  Principle
  Declare the base class dtor virtual when someone will delete a derived class object via a base class pointer.

  ```
  class stack : private deque {
  public:
     virtual ~stack();
           ⋮
  };
  ```




  ```
  stackP* s=new stackPS;
  delete s;
  s->~stackP()
  ```

  1 resolve to `stackP::~stackP()`
  2.2 call `stackPS::~stackPS()`, the final overrider of `stackP::~stackP()`

  Note that since no one shall delete a `stack` object via a `deque` pointer, the `deque`'s dtor needn't be virtual.

- If `B::~B()` is virtual, so is `D::~D()`.

- Even if the dtor isn't inherited, `D::~D()` overrides `B::~B()`.

Pure virtual function

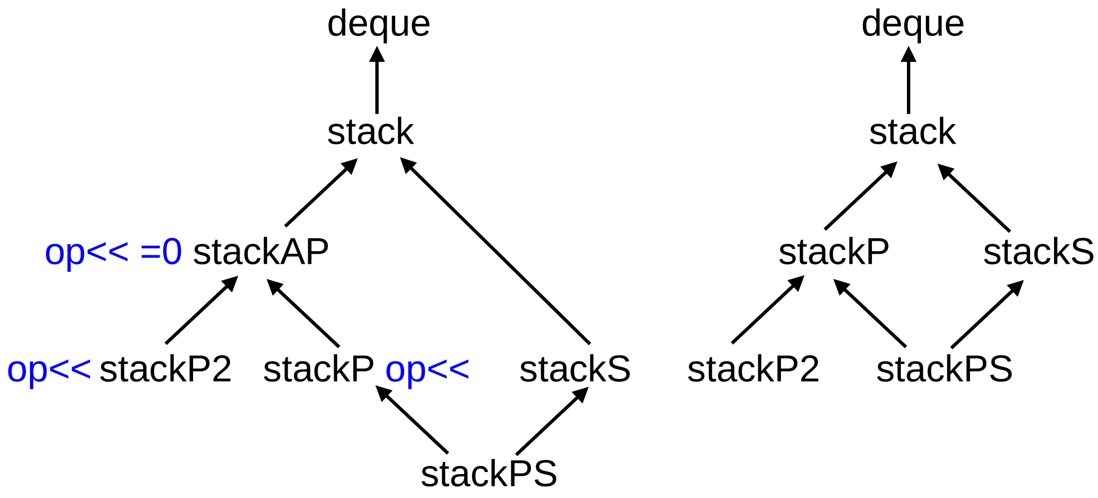
- A pure virtual function is a virtual function whose declaration in the class declaration ends with the pure specifier =0.
- A class is *abstract* if it contains or inherits at least one pure virtual function for which the final overrider is pure virtual. Otherwise, it is *concrete*.
- An abstract class can only be used a base class of some other class.
  In other words, no objects of an abstract class can be created except as subobjects of a class derived from it.
  (Pointers and references to an abstract class can of course be declared.)
- An abstract class is often used to represent an abstract concept – It defines an interface, but doesn't necessarily provide implementations for all its member functions.
  A concrete class derived from it implements all the missing functionalities.
- Public inheritance of a pure virtual function inherits only a function interface.
  The concrete class that inherits it has to provide its own implementation.
- A pure virtual function may or may not be defined.
  If it is defined, it can only be called with qualified-id syntax, for unqualified-id syntax can only call the final overrider declared in a concrete class derived from the abstract class.
  (Exception: A pure virtual dtor must always be defined, and will be invoked tacitly when a derived class dtor is invoked.)
- The declaration and definition of a pure virtual function cannot be written together, e.g.

```
struct A {
   virtual void pvf(){}=0;     // ill-formed
};
```

- Example

  Stack application – Coin change

  Count the number of ways and generate all the ways to make change.



// Abstract class **stackAP**

```
class stackAP : virtual public stack {
friend ostream& operator<<(ostream&,const stackAP&);
private:
   virtual ostream& operator<<(ostream&) const=0;
};

ostream& operator<<(ostream& os,const stackAP& s)
{
   return s << os;
}
```

// Concrete class **stackP**

```
class stackP : public stackAP {
friend ostream& operator<<(ostream&,const stackP&);
private:
   virtual ostream& operator<<(ostream&) const;
};

ostream& operator<<(ostream& os,const stackP& s)
{
   return s << os;
}
```

- Example (Cont'd)

```
ostream& stackP::operator<<(ostream& os) const
{
   node* p=head->succ;
   while (p!=head) { os << p->datum; p=p->succ; }
   return os;
}
```

// Concrete class **stackP2**

```
class stackP2 : public stackAP {
friend ostream& operator<<(ostream&,const stackP2&);
private:
   virtual ostream& operator<<(ostream&) const;
};

ostream& operator<<(ostream& os,const stackP2& s)
{
   return s << os;
}

ostream& stackP2::operator<<(ostream& os) const
{
   node* p=head->succ;
   if (p==head) return os;
   int d=p->datum,c=1;          // 1(10) 5(2) 10(3) 50(1)
   while ((p=p->succ)!=head)
      if (p->datum==d) c++;
      else {
         os << d << "(" << c << ") ";
         d=p->datum; c=1;
      }
   return os << d << "(" << c << ")";
}
```

// Application

```
int main()
{
   int d[4]={1,5,10,50};
   cout << cc(10,4,d,stackP());
   cout << cc(100,4,d,stackP2());
}
```

- Example (Cont'd)

```
int cc(int n,int k,int* d,const stackAP& s)
{
   if (n==0) {
      cout << s << endl;
      return 1;
   } else if (n<0||k==0) return 0;
   else {
      const_cast<stackAP&>(s).push(d[k-1]);
      int x=cc(n-d[k-1],k,d,s);
      const_cast<stackAP&>(s).pop();
      return x+cc(n,k-1,d,s);
   }
}
```

Alternative design

An (impure?) virtual function provides an interface and a defualt implementation that is inherited *automatically* if the derived class doesn't redefine it. This is unsafe in the sense that if the derived class has to provide a redefinition but forgets to do so, the error cannot be detected at compile time.
A safer design is to inherit the default implementation *manually*.

// The abstract class **stackAP** provides an interface as well as
// an *on-rqueset* default implementation.

```
class stackAP : virtual public stack {
friend ostream& operator<<(ostream&,const stackAP&);
protected:
   virtual ostream& operator<<(ostream&) const=0;
};

ostream& stackAP::operator<<(ostream& os) const
{
   node* p=head->succ;
   while (p!=head) { os << p->datum; p=p->succ; }
   return os;
}
```

- Example (Cont'd)

// The concrete class **stackP** explicitly requests to inherit the
// default implementation.

```
ostream& stackP::operator<<(ostream& os) const
{
   return stackAP::operator<<(os);
}
```

// All the others remain unchanged.

- Summary

Inheritance	Virtual or not?	What is (are) inherited?
private protected	NA	implementation
public	non-virtual	interface *mandatory* implementation
	virtual	interface *auto default* implementation
	pure virtual w/o implementation	interface
	pure virtual w. implementation	interface *manual default* implementation

Appendix

The **deque** class runs in this lecture.

```
deque::node::node(int d,node* p,node* s)
:  datum(d), pred(p), succ(s) {}

deque::deque()
: head((node*)operator new(sizeof(node)))
{
   head->pred=head->succ=head;
}

deque::deque(const deque& rhs)
:  head((node*)operator new(sizeof(node)))
{
   head->pred=head->succ=head;
   node* p=rhs.head->succ;
   while (p!=rhs.head) {
      push_back(p->datum); p=p->succ;
   }
}

deque& deque::operator=(const deque& rhs)
{
   if (this!=&rhs) {
      while (!empty()) pop_front();
      node* p=rhs.head->succ;
      while (p!=rhs.head) {
         push_back(p->datum); p=p->succ;
      }
   }
   return *this;
}

deque::~deque()
{
   while (!empty()) pop_front();
   operator delete(head);
}
```

```
void deque::push_front(int d)
{
   head->succ=head->succ->pred
                     =new node(d,head,head->succ);
}

void deque::push_back(const int d)
{
   head->pred=head->pred->succ
                     =new node(d,head->pred,head);
}

void deque::pop_front()
{
   if (!empty()) {
      head->succ=head->succ->succ;
      delete head->succ->pred;
      head->succ->pred=head;
   }
}

void deque::pop_back()
{
   if (!empty()) {
      head->pred=head->pred->pred;
      delete head->pred->succ;
      head->pred->succ=head;
   }
}

bool deque::empty() const { return head->succ==head; }
int& deque::front() { return head->succ->datum; }
const
int& deque::front() const { return head->succ->datum; }
int& deque::back() { return head->pred->datum; }
const
int& deque::back() const { return head->pred->datum; }
```