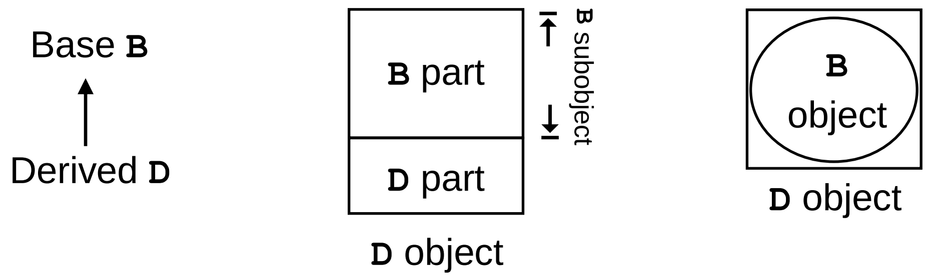
W.P.Hwang
CS-CCS-NCTU

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** → private of **D**
  2 protected public/protected of **B** → protected of **D**
  3 public public/protected of **B** → public/protected of **D**

  Comment

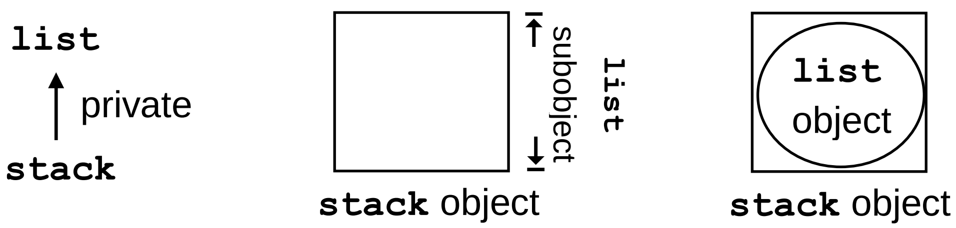
  The private members of **B** remain inaccessible to **D** unless **D** is a friend of **B**.

- Example

```
template<typename T>
class stack : private list<T> {
public:
   typedef typename list<T>::size_type size_type;
   void push(const T& val)
   {
      this->push_back(val);
   }
   void push(T&& val)
   {
      this->push_back(std::move(val));
   }
   void pop() { this->pop_back(); }
   T& top() { return this->back(); }
   const T& top() const { return this->back(); }
   size_type size() const
   {
      return list<T>::size();
   }
   bool empty() const { return list<T>::empty(); }
};
```

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

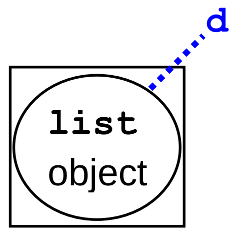
In this example, **stack** is implemented by **list**.



- Example (Cont'd)

Inheritance vs layering

```
template<typename T>
class stack {
public:
   typedef typename list<T>::size_type size_type;
   void push(const T& val)
   {
      d.push_back(val);
   }
   void push(T&& val) {
   {
      d.push_back(std::move(val));
   }
   void pop() { d.pop_back(); }
   T& top() { return d.back(); }
   const T& top() const { return d.back(); }
   size_type size() const { return d.size(); }
   bool empty() const { return d.empty(); }
private:
   list<T> d;
};
```



stack object

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)

  Special member functions

  In either case, since we have not declared any ctor for the **`stack`** class, the compiler will generate a default ctor for the **`stack`** class, using **`list`**'s default ctor to initialize the **`list`** subobject contained in a **`stack`** object.

  One the other hand, we needn't define the dtor, copy/move ctor, and copy/move assignment operator for the **`stack`** class by ourselves, since no dynamic storage is allocated within the **`stack`** class.

  For inheritance, the implicitly generated special member functions for the **`stack`** class are defined as

```
template<typename T>
class stack : private list<T> {
public:
   stack() : list<T>() {}
   ~stack() {}          // invoke list<T>::~list<T>()
   stack(const stack<T>& rhs)
   :  list<T>(rhs) {}
   stack(stack<T>&& rhs)
   :  list<T>(std::move(rhs)) {}
   stack<T>& operator=(const stack<T>& rhs)
   {
      if (this!=&rhs) list<T>::operator=(rhs);
      return *this;
   }
   stack<T>& operator=(stack<T>&& rhs)
   {
      if (this!=&rhs)
         list<T>::operator=(std::move(rhs));
      return *this;
   }
};
```

- Example (Cont'd)

For layering, the implicitly generated special member functions for the **`stack`** class are defined as

```
template<typename T>
class stack {
public:
   stack() : d() {}
   ~stack() {}             // invoke d.~list<T>()
   stack(const stack<T>& rhs) : d(rhs.d) {}
   stack(stack<T>&& rhs) : d(std::move(rhs.d)) {}
   stack<T>& operator=(const stack<T>& rhs)
   {
      if (this!=&rhs) d=rhs.d;
      return *this;
   }
   stack<T>& operator=(stack<T>&& rhs)
   {
      if (this!=&rhs) d=std::move(rhs.d);
      return *this;
   }
};
```

Upcast

A nonstatic member function of a class expects a **`this`** pointer pointing to an object of that class.

In this context, a nonstatic member function of class **`list<T>`** expects a **`this`** pointer pointing to the **`list<T>`** subobject contained in a **`stack<T>`** object.

For inheritance, an upcast is needed:

```
void stack<T>::pop() { this->pop_back(); }
```

is compiled to

```
void stack<T>::pop(stack<T>* this)
{
   pop_back(static_cast<list<T>*>(this));
}
```

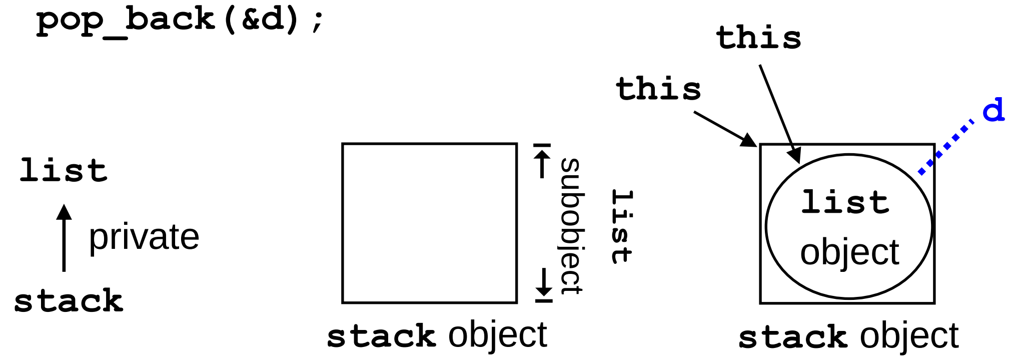
- Example (Cont'd)

For layering, no conversion is required:

```
void stack<T>::pop() { d.pop_back(); }
```

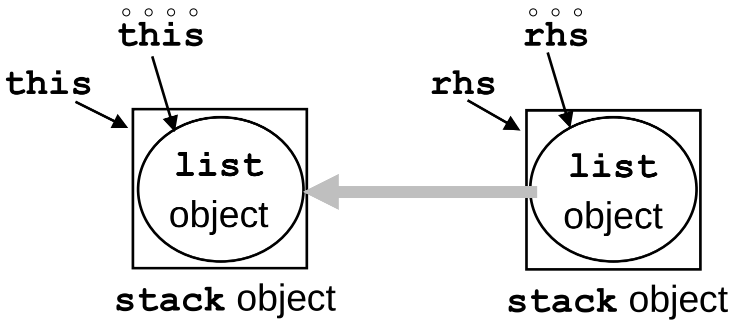
is compiled to

```
void stack<T>::pop(stack<T>* this)
{
   pop_back(&d);
}
```



In single-inheritance, the upcast only converts the type of **this** from **stack<T>*** to **list<T>***. Both pointers point to the same beginning memory location of the **stack<T>** object and the **list<T>** subobject.

```
stack<T>::stack(const stack<T>& rhs)
: list<T>(rhs)        d(rhs.d) no conversion involved
{}             implicit upcast

list<T>::list(const list<T>& rhs);
```

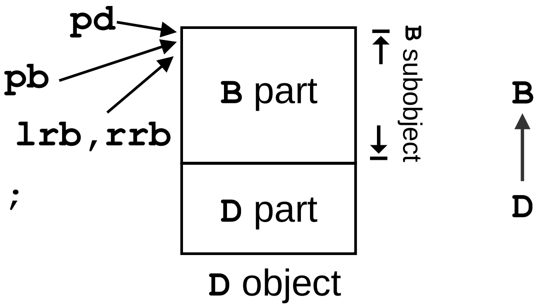



```
stack<T>::stack(stack<T>* this,const stack<T>& rhs)
: list<T>(this,rhs)
{}             implicit upcast

list<T>::list(list<T>* this,const list<T>& rhs);
```

Upcast

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

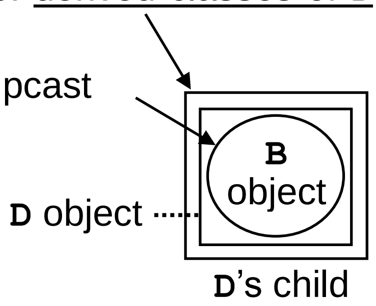
```
D* pd=new D;
B* pb=pd;
B& lrb=*pd;
B&& rrb=std::move(*pd);
```



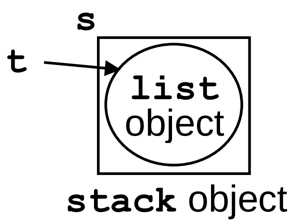
- An implicit upcast can be done if the base class is
  1. accessible (i.e. if an invented public member of the base class is accessible), and
  2. unambiguous (in the presence of multiple inheritance)

  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()
{
   stack<int> s;
   list<int>& t=s;      // stack<int> → list<int>&
   t->push_front(7);    // disorder stack s
}
```

Were the upcast allowed, one could disorder stack **`s`**.

To break down the protection, use **`reinterpret_cast`** or C-style cast.

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/move ctor
  Perform memberwise copy/move of its subobjects.
  Direct base classes first, and then nonstatic data members
- Copy/move assignment operator
  Perform memberwise copy/move 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(expressions), ...
              mem_id{expressions}, ... {}
  ```

  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
  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

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

```
class stack : private list<int> {
public:
    void push(const int& val)
    {
        push_back(val);
    //  this->push_back(val)
    }
    void pop() { pop_back(); }
    bool empty() const
    {
        return list<int>::empty();
    //  return this->list<int>::empty();
    }
    // other members omitted
};
```

```
list
push_back, pop_back, empty
  stack
  push, pop, empty
```

- Unqualified name lookup

  1 Non-dependent names are looked up at the point of template definition.
  2 Dependent names are looked up at the point of template instantiation. (To be explained in more detail later)

- The declarations in a dependent base aren't examined during unqualified name lookup (either at the point of template defini -tion or at the point of template instantiation).

- Example // dependent base

```
template<typename T>
class stack : private list<T> {
public:
   void push(const T& val)
   {
      push_back(val);    // dependent name
   }                     // not found at instantiation time
   void pop()
   {
      pop_back();        // non-dependent name
   }                     // not found at definition time
   bool empty() const
   {
      return list<T>::empty();   // qualified name
   }
   // other members omitted
};
int main() { stack<int> s; s.push(3); }
```

There are three ways to get over this problem.
Taking **push_back** as an example:

1 **this->push_back(val)**
2 **list<T>::push_back(val)**
3 **using list<T>::push_back;** within the **stack** class

- Dependent name resolution

  In resolving dependent names at the point of template instantiation, names from the following sources are considered:
  1 Declarations that are visible at the point of definition
  2 Declarations from namespaces determined by ADL (from both the instantiation and the definition contexts).

- Example

```
void pop_back() {}

// definition of the stack class template
// push calls  push_back(val)    //* error, not found
// pop calls  pop_back()         // ok, call ::pop_back

void push_back(int) {}
int main()
{
   stack<int> s; s.push(3);
}
```

At the instantiation time of **stack<int>::push**, the unquailfied name **push_back** doesn't resolve to **::push_back**, since
1 it's invisible at the point of definition of **stack<int>::push**
2 the type of **val** is **int**, and so ADL doesn't applies

Compare with this:

```
void pop_back() {}

// definition of the stack class template
// push calls  push_back(val)    // ok, call ::push_back(B)
// pop calls  pop_back()         // ok, call ::pop_back

struct B {};
void push_back(B) {}             // it is selected, thanks to ADL
int main()
{
   stack<B> s; s.push(B());
}
```

- Example – Printable stack: Combination generation

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

int c(int n,int k)
{
   return c(n,k,stackP<int>());
}
```

Method 1: Protected inheritance + Private data members

`list` (private data in method 1 → protected data in method 2)

↑ protected

`stack`   `stackP` and its friends may access `list`'s public/protected members

↑ public

`stackP` (friend `operator<<` for `stackP`)

```
template<typename T>
class stack : protected list<T>  {
// all members remain the same
};

template<typename T> class stackP;

template<typename T>
ostream& operator<<(ostream&,const stackP<T>&);
```

- Example (Cont'd)

```
template<typename T>
class stackP : public stack<T> {
friend
ostream& operator<< <T>(ostream&,const stackP<T>&);
};

template<typename T>
ostream& operator<<(ostream& os,const stackP<T>& s)
{
   for (auto& e : s) os << e << ' ';
   return os;
}
```

or

```
template<typename T>
ostream& operator<<(ostream& os,const stackP<T>& s)
{
   typename list<T>::const_iterator it=s.begin();
   while (it!=s.end()) {
      os << *it << ' ';
      ++it;                    // upcast to grandparent
   }
}
```

Method 2: Protected inheritance + Protected data members

```
template<typename T>
class list {
// other members remain the same
protected:
   struct node;
   node* head;
   size_type sz;
};
```

// **stack** and **stackP** remain unchanged

- Example (Cont'd)

```
template<typename T>
ostream& operator<<(ostream& os,const stackP<T>& s)
{
   typename list<T>::node* p=s.head->succ;
   while (p!=s.head) {
      os << p->datum << ' ';
      p=p->succ;                    // no upcast
   }
   return os;
}
```

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

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.

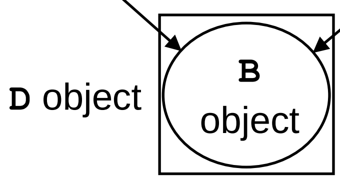
If class `D` publicly inherits from class `B`, then

1) `B` is more general than `D` (or, `D` is more specialized than `B`).
2) every `D` object is a `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&);   // reference to subobject B contained in D
void p(B&&);
void p(B);    // copy/move subobject B contained in D
```

can be invoked by `B::B(const B&)` or `B::B(B&&)`

```
p(D());
```



- Example (Cont'd)

  Special member functions

  It should now be clear that implicitly generated special member functions suffice for the **stackP** class.

  For examples,

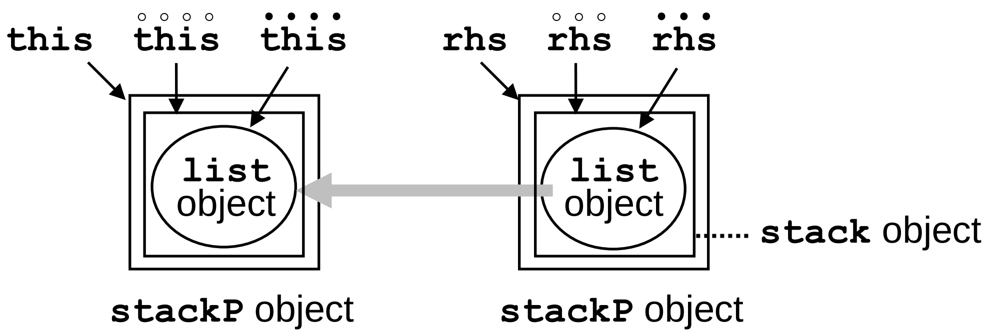
```
// move ctor
stackP<T>::stackP(stackP<T>&& rhs)
: stack<T>(std::move(rhs))
{}
```

implicit upcast

```
stack<T>::stack(stack<T>&& rhs)
: list<T>(std::move(rhs))
{}
```

implicit upcast

```
list<T>::list(list<T>&& rhs)
: … { … }
```




```
// dtor
stackP<T>::~stackP() { }
stack<T>::~stack() { }
list<T>::~list() { … }
```

// call it and then return

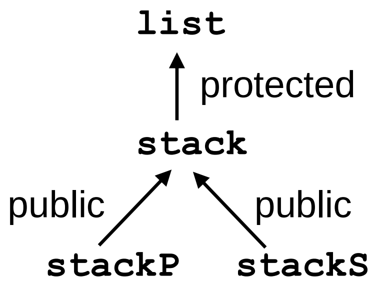
// call it and then return

// destroy the list

- Example – Summable stack: 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.

```
int dgs(int* a,int n,int k,int d,stackS<int>&& s)
{
   if (k==0||n==k) {
      int sum=s.sum();       // summable stack
      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,std::move(s));
      s.pop();
      return r+dgs(a,n-1,k,d,std::move(s));
   }
}

int dgs(int* a,int n,int k,int d)
{
   return dgs(a,n,k,d,stackS<int>());
}
```



// A summable stack is a stack.

```
template<typename T>
class stackS : public stack<T> {
public:
   T sum();
};
```

- Example (Cont'd)

Method 1: Protected inheritance + Private data members

```
template<typename T>
T stackS<T>::sum()
{
    T s{};
    for (auto& e : *this) s+=e;
    return s;
}
```

Method 2: Protected inheritance + Protected data members

```
template<typename T>
T stackS<T>::sum()
{
   T s{};           // value-initialized
   typename list<T>::node* p=this->head->succ;
   while (p!=this->head) {
      s+=p->datum;
      p=p->succ;
   }
   return s;
}
```

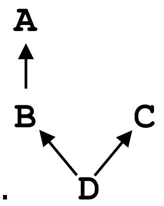
Comment

Due to dependent base, we cannot simply use the unqualified names **`node`** and **`head`** within **`stackS<T>::sum`**.

Layering vs private/protected inheritance

Whenever there are protected members and/or virtual functions, use private/protected inheritance to model "is-implemented-by", since only inheritance gives access to protected members.

Multiple inheritance

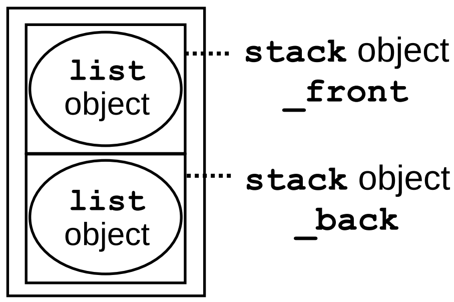


- A class can have one or more direct base class.
  Single inheritance: single direct base class
  Multiple inheritance: multiple direct base classes
- A class cannot be a direct base class more than once.
  A class can be an indirect base class more than once.
  A class can be a direct and an indirect 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)

```
template<typename T>
class queue {
public:
   void push(const T& val)
   {
      _back.push(val); _check();
   }
   void push(T&&);
   void pop() { _front.pop(); _check(); }
   T& front();
   const T& front() const;
   bool empty() const;
private:
   void _check();
   stack<T> _front,_back;
};
```

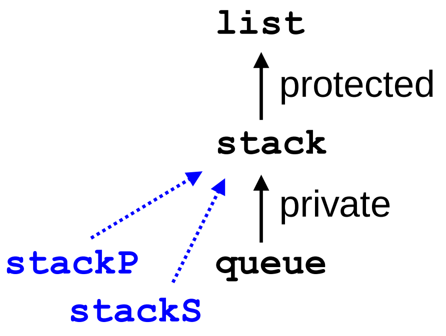
// Definitions of other members
// omitted.

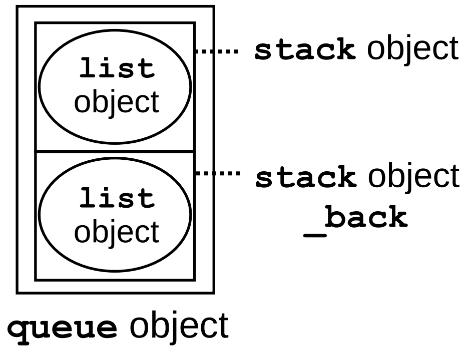


queue object

- Example (Cont'd)

Method 2 – Layering + Single inheritance





Comment
Private inheritance is sufficient, if we consider only the solid-line class lattice (or hierarchy)

```
template<typename T>
class queue : private stack<T> {
public:
   void push(const T&);
   void push(T&&);
   void pop() { stack<T>::pop(); _check(); }
   T& front() { return this->top(); }
   const T& front() const { return this->top(); }
   bool empty() const { return stack<T>::empty(); }
private:
   void _check();
   stack<T> _back;
};
template<typename T>
void queue::push(const T& val)
{
   _back.push(val); _check();
}
template<typename T>
void queue::push(T&& val)
{
   _back.push(std::move(val)); _check();
}
```

- Example (Cont'd)

```
template<typename T>
void queue<T>::_check()
{
   if (empty())
      while (!_back.empty()) {
         stack<T>::push(_back.top());
         _back.pop();
      }
}
```

Selected special member functions

// default ctor
```
template<typename T>
queue<T>::queue()
: stack<T>(), _back() {}
```

Direct base is default-constructed first and then nonstatic data member.

// copy ctor
```
template<typename T>
queue<T>::queue(const queue<T>& rhs)
: stack<T>(rhs), _back(rhs._back) {}
```

Direct base is copy-constructed first and then nonstatic data member.

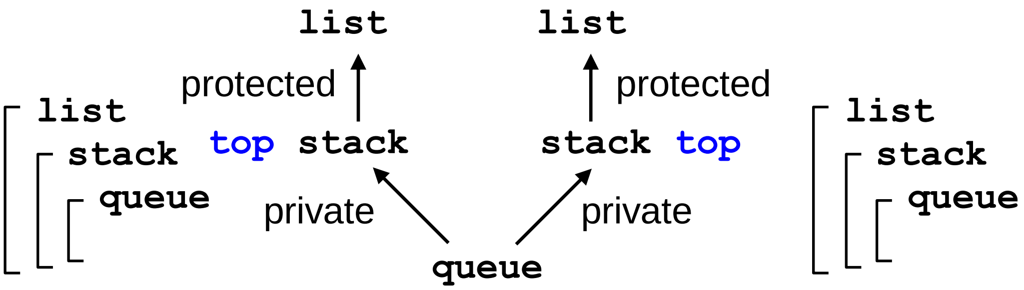
// dtor
```
template<typename T>
queue<T>::~queue()
{}
```

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

- Example (Cont'd)

  Method 3 – Multiple inheritance

  Attempt 1

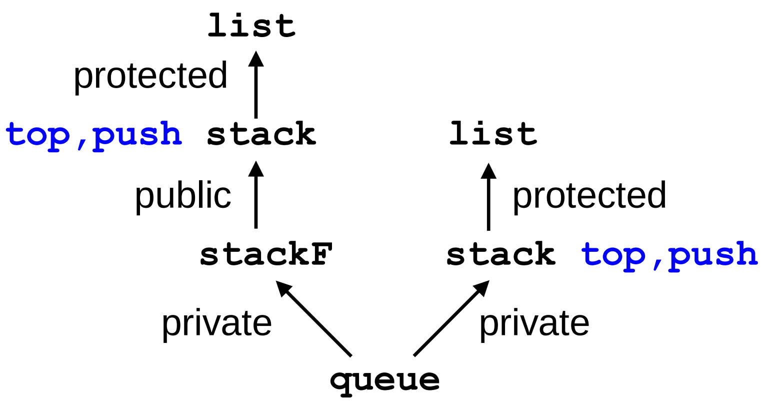



```
template<typename T>
T& queue<T>::front() { return this->top(); }
```

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



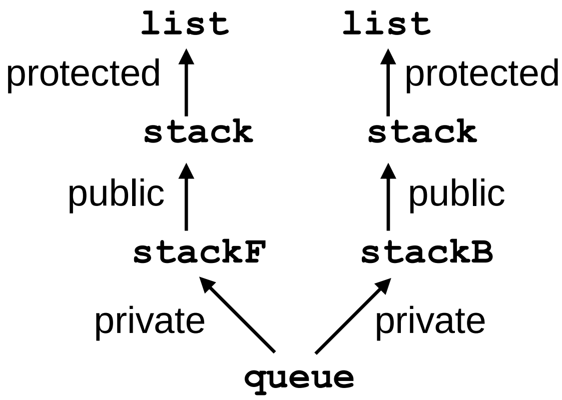

```
template<typename T>
T& queue<T>::front() { return stackF::top(); }
template<typename T>
void queue<T>::push(const T& val)
{
   this->push(val); _check();
};
```

Ambiguous! How to access **stack<T>::push** of direct base?

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

- Example (Cont'd)

  Solution



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

Try 1

```
template<typename T>
class stackF : protected stack<T> {};
stackF<int> 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

```
template<typename T>
class stackF : public stack<T> {
protected:
   stackF() = default;
};
stackF<int> s;       // no, cannot create a stackF object
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<int> q;
stackF<int> s(reinterpret_cast<stackF<int>&>(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<int>&>(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**.

  Solution

  Protect all special member functions, taking **stackF** as an example

```
template<typename T>
class stackF : public stack<T> {      // isa
protected:
   stackF() = default;
   stackF(const stackF<T>&) = default;
   stackF(stackF<T>&&) = default;
   ~stackF() = default;
   stackF& operator=(const stackF<T>&) = default;
   stackF& operator=(stackF<T>&&) = default;
};
```

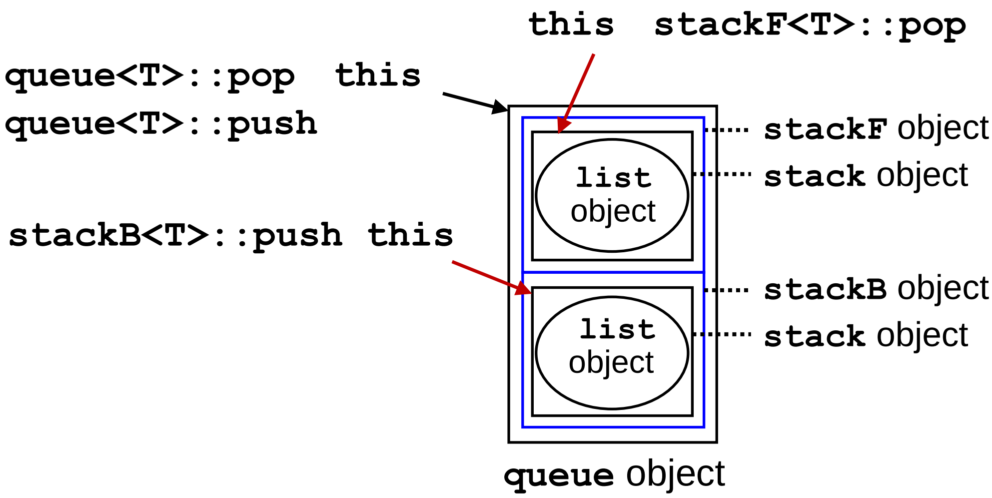
- Example (Cont'd)

  Finally, define // ctors are executed from left to right

```
template<typename T>
class queue
:  private stackF<T>, private stackB<T> {
public:
   void push(const T&);
   void push(T&&);
   void pop() { stackF<T>::pop(); _check(); }
   T& front() { return stackF<T>::top(); }
   const T&
   front() const { return stackF<T>::top(); }
   bool
   empty() const { return stackF<T>::empty(); }
private:
   void _check();
};

template<typename T>
void queue<T>::push(const T& val)
{
   stackB<T>::push(val); _check();
};

template<typename T>
void queue<T>::push(T&& val)
{
   stackB<T>::push(std::move(val)); _check();
};

template<typename T>
void queue<T>::_check()
{
   if (empty())
      while (!stackB<T>::empty()) {
         stackF<T>::push(stackB<T>::top());
         stackB<T>::pop();
      }
}
```

- Example (Cont'd)

Upcast in multiple inheritance



In multiple-inheritance, an upcast may have to adjust the pointer value.
To see why, notice that a **`queue<T>`** object and the embedded **`stackF<T>::stack`** object have the same beginning address, but the embedded **`stackB<T>::stack`** object has a different beginning address. Thus,

`queue<T>*` → **`stackF<T>::stack*`**		no adjustment
`queue<T>*` → **`stackB<T>::stack*`**		adjustment needed

Selected special member functions

```
// default ctor
template<typename T>
queue<T>::queue()
: stackF<T>(), stackB<T>() {}
```

Direct base classes are initialized in declaration order.

```
// dtor
template<typename T>
queue<T>::~queue() {}
```

Before it returns,
`stackB<T>::~stackB()` and **`stackF<T>::~stackF()`** are called in order.

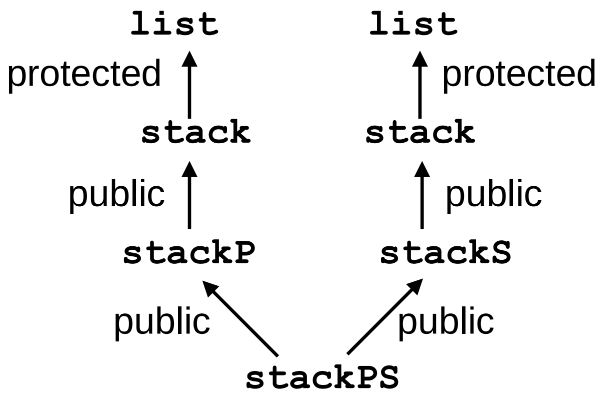
Virtual inheritance

- 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 – Printable and summable stack

  Enumerated divisible group sums

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

  Method 1

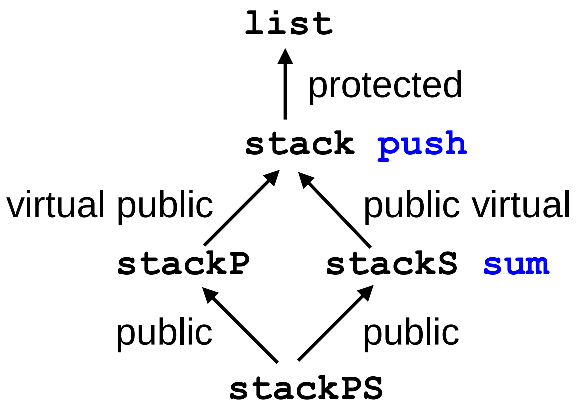



```
template<typename T>
class stackPS
:  public stackP<T>, public stackS<T>
{};
```

This class hierarchy is inefficient, as the most derived **`stackPS`** object *undesirably* contains two **`stack`** subobjects. Each time an element is pushed onto or popped off a **`stackPS`** object, it must be pushed onto or popped off both **`stack`** subobjects.

- Example (Cont'd)

Method 2

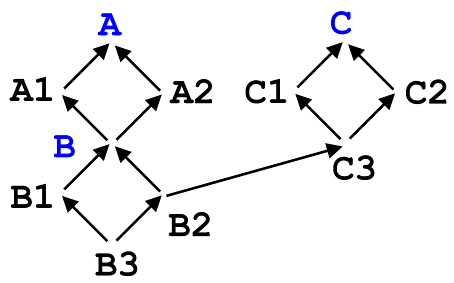



```
template<typename T>
class stackP : virtual public stack<T> {
    // friend operator<< declaration
};

template<typename T>
class stackS : virtual public stack<T> {
    // sum  declaration
};

template<typename T>
class stackPS
:   public stackP<T>, public stackS<T> {
};

int edgs(int* a,int n,int k,int d,stackPS<int>&& s)
{
    if (k==0||n==k) {
        int sum=s.sum();      // printable and summable stack
        for (int i=0;i<k;i++) sum+=a[i];
        if (sum%d==0) {
            for (int i=k;i>=1;--i) cout << i << ' ';
            cout << s << endl;
            return 1;
        } else return 0;
    } else
        // similar to the previous function dgs
}
```

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



In which **`A`**, **`B`**, and **`C`** are virtual bases. The ctors are executed in the following order:

`B3`, **`A`**, **`B`**, **`A1`**, **`A2`**, **`C`**, **`B1`**, **`B2`**, **`C3`**, **`C1`**, **`C2`**

- Example (Cont'd)

Here are the implicitly generated ctors for the stack hierarchy:

```
template<typename T>
stack<T>::stack() : list<T>() {}

template<typename T>
stackP<T>::stackP() : stack<T>() {}

template<typename T>
stackS<T>::stackS() : stack<T>() {}

template<typename T>
stackPS<T>::stackPS()
: stack<T>(),stackP<T>(),stackS<T>() {}
```

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.

- Example (Cont'd)

  Consider the following declarations:

  `stackP<int> s;`

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

  `stackPS<int> s;`

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

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

Copy/move assignment operator

- As usual, the copy assignment operator is implicitly defined if it is needed, except that it is unspecified whether subobjects representing virtual base classes are assigned more than once. (This is an efficiency issue, rather than a semantics issue.)

- However, the move assignment operator is defined as deleted if a class has any direct or indirect virtual base class.

  To see why, observe that if one object is moved to another more than once, both objects become undefined, e.g.

  ```
  list<int> a{1,2,3},b{4,5};
  a=std::move(b);        // b undefined
  a=std::move(b);        // a undefined, too
  ```

  Thus, if it isn't defined as deleted, the implicitly-defined move assignment operator must behavior like ctors and unlike the copy assignment operator – this is a disaster in semantics.

- Principle

For classes with virtual bases, define
1 copy assignment operators for compiler-independent guaranteed efficiency, and
2 move assignment operators, if move semantics is desired

- Example (Cont'd)

Class **stackPS**

```
// copy assignment
template<typename T>
stackPS<T>&
stackPS<T>::operator=(const stackPS<T>& rhs)
{
   if (this!=&rhs) {
      stackS<T>::operator=(rhs);
//    stackP<T>::operator=(rhs);          // don't call
   }
   return *this;
}
```

```
// move assignment
template<typename T>
stackPS<T>& stackPS<T>::operator=(stackPS<T>&& rhs)
{
   if (this!=&rhs) {
      stackS<T>::operator=(std::move(rhs));
//    stackP<T>::operator=(std::move(rhs));
   }                                      // don't call
   return *this;
}
```

Having defined them, other special member functions, when needed, have to be explicitly declared as defaulted.

- Example (Cont'd)

```
template<typename T>
class stackPS : public stackP<T>,public stackS<T> {
public:
   stackPS() = default;
   stackPS(const stackPS<T>&) = default;
   stackPS(stackPS<T>&&) = default;
   stackPS<T>& operator=(const stackPS<T>&);
   stackPS<T>& operator=(stackPS<T>&&);
};
```

Class **stackS** (**stackP** is similar)

```
template<typename T>
class stackS : virtual public stack<T> {
public:
   T sum();
   stackS() = default;
   stackS(const stackS<T>&) = default;
   stackS(stackS<T>&&) = default;
   stackS<T>& operator=(const stackS<T>&) = default;
   stackS<T>& operator=(stackS<T>&&);
};
```

```
// move assignment
template<typename T>
stackP<T>& stackS<T>::operator=(stackS<T>&& rhs)
{
   if (this!=&rhs)
      stack<T>::operator=(std::move(rhs));
   return *this;
}
```

Dtor

- There is nothing special for the dtor. The order of (virtual and nonvirtual) base class dtor invocations is guaranteed to be the reverse order of ctor invocations.

Virtual function

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

```
template<typename T>
class stackS : virtual public stack<T> {
public:
   stackS() : stack<T>(), _sum() {}
   void push(const T&);
   void push(T&&);
   void pop();
   T sum() { return _sum; }
private:
   T _sum;
};

template<typename T>
void stackS<T>::push(const T& val)
{
   _sum+=val; stack<T>::push(val);
}

template<typename T>
void stackS<T>::push(T&& val)
{
   _sum+=val; stack<T>::push(std::move(val));
}

template<typename T>
void stackS<T>::pop()
{
   _sum-=this->top(); stack<T>::pop();
}
```

Notice that the **stackS** class redefines the member functions **push** and **pop** publicly inherited form the **stack** class.

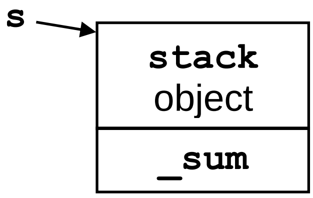
- Example (Cont'd)

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

```
stack<int>* s=new stackS<int>();
```

and consider

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



stackS object

Shall it call `stack<int>::push` or `stackS<int>::push`?

Clearly, it should invoke `stackS<int>::push`.
Why?
∵ The object in existence is indeed a `stackS<int>` object. Invoking `stack<int>::push` will make it inconsistent.

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

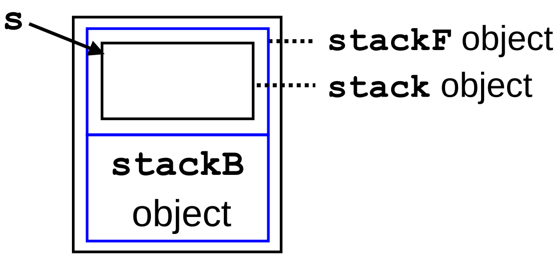
However, as written, it will invoke `stack<int>::push`, for
1) `push` is a non-virtual function, and
2) the static (i.e. declared) type of `s` is `stack*`

As a contrast example, let

```
queue<int> q;
stack<int>* s=reinterpret_cast<stackF<int>*>(&q);
```

and consider

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



queue object

Shall it call `stackF<int>::push` or `queue<int>::push`?

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

- Example (Cont'd)

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

  Comment

  We won't say that **`queue<T>::push`** redefines **`stack<T>::push`**, since 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.

```
template<typename T>
class stack : protected list<T> {
public:
// virtual function
    virtual void push(const T&);
    virtual void push(T&&);
    virtual void pop();
// nonvirtual function
    T& top();
    const T& top() const;
    size_type size() const;
    bool empty() const;
};
```

  The derived class **`stackP`** doesn't redefine the inherited virtual functions. But, **`stackS`** does.

- Example (Cont'd)

```
template<typename T>
class stackS : virtual public stack<T> {
public:
   stackS();
   virtual void push(const T&);     // optional
   virtual void push(T&&);          // optional
   virtual void pop();              // optional
   T sum();
private:
   T _sum;
};
```

Comment

1 The **`virtual`** specifier can only appear in the declaration of nonstatic member functions.
2 The **`virtual`** specifier is optional when redefining **`push`** and **`pop`** in the derived class.

```
stack<int>* s=new stackS<int>();
s->push(2);              // call stackS<int>::push
stack<int>* s=new stack<int>();
s->push(2);              // call stack<int>::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 function

- Let **vf** be a virtual function declared in **B**

```
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).

- Example

```
stackS<int> s;
stack<int>& t=s;
t.push(2);              // call stackS<int>::push
stack<int> u=s;         // copy the stack subobject
u.push(2);              // call stack<int>::push
```

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

```
template<typename T>
void stackS<T>::push(const T& val)
{
   _sum+=val; stack<T>::push(val);
}
```

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 otherwise, 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.

- Comment

  ⌖ 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

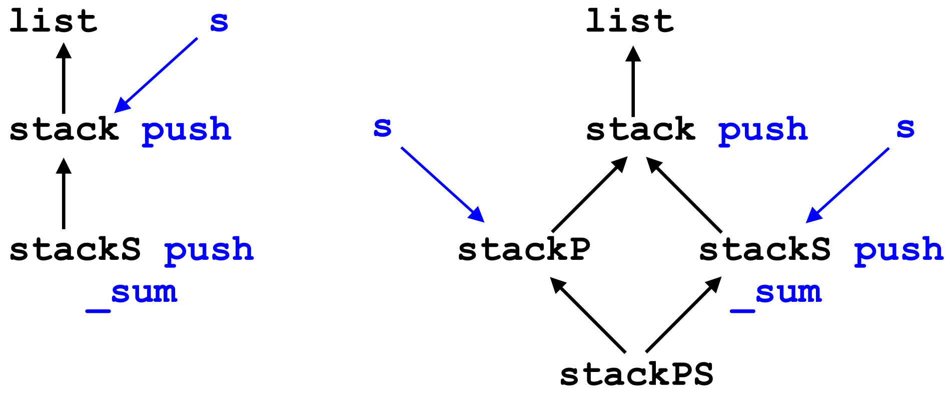
Where to start	the dynamic type
Where to search	only the path(s) from the dynamic type to the class containing the resolved function
Condition	overriding (parameters are considered)

- Example

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

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

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

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

- Example



`stack<int>* s=new stackS<int>();`
`s->push(n);`

1 resolve to **stack::push**

2.2 call **stackS::push,** the final overrider of **stack::push**

`stackS<int>* s=new stackPS<int>();`
`stackPS<int>* s=new stackPS<int>();` ♥
`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<int>* s=new stackPS<int>();`
`stackP<int>* s=new stackPS<int>();` ♦
`s->push(n);`

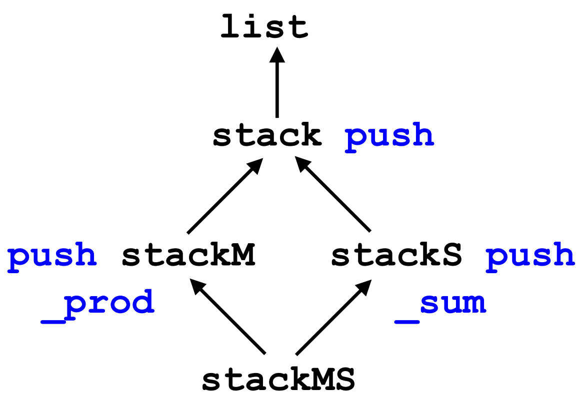
1 resolve to **stack::push**

2.2 call **stackS::push,** the final overrider of **stack::push** ∵ Being the overrider along the r.h.s path, it 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.

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

- Example (unique final overrider)



where the `stackM` class redefines `stackM::push` as:

```
template<typename T>
void stackM<T>::push(const T& val)
{
   _prod*=val; stack<T>::push(val);
}
```

This class lattice is illegal, since `stack::push` doesn't have a unique final overridder. (`stackM::push` and `stackS::push` are two overriders. But, neither is final.)

To correct it, `stackMS::push` must also be redefined (where `_sum` and `_prod` are protected):

```
template<typename T>
void stackMS<T>::push(const T& val)
{
   _prod*=val; _sum+=val; stack<T>::push(val);
}
```

Virtual destructor

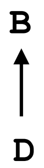
- If the base class's dtor is nonvirtual, the result of deleting a derived class object through a base class pointer is undefined.
- Example

  ```
  stack<int>* s=new stackPS<int>();
  delete s;        // undefined
  ```

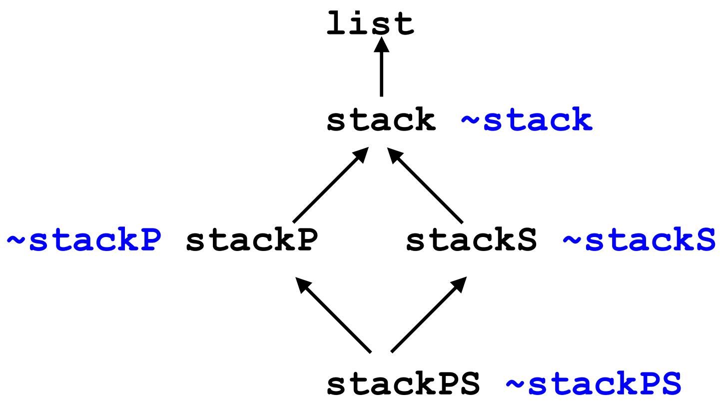
  The behaviour is undefined, because the dtor of the base class **`stack`** isn't virtual. Usually, the dtor of **`stack`** is called, but the dtor of **`stackPS`** is expected.

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

- If **`B::~B()`** is virtual, so is **`D::~D()`**.
- Even if the dtor isn't inherited, **`D::~D()`** overrides **`B::~B()`**.
- Example (Cont'd)




  ```
  template<typename T>
  class stack : protected list<T> {
  public:
     virtual ~stack() = default;
     // other members omitted
  };
  ```




  ```
  stack<int>* s=new stackPS<int>();
  delete s;
  ⇒  s->~stack()
  ```

- Example (Cont'd)

  `s->~stack()`

  1 resolve to **`stack::~stack()`**

  2.2 call **`stackPS::~stackPS()`**, the final overrider of **`stack::~stack()`**

  Comment
  As far as this class lattice is concerned, **`list`**'s dtor needn't be virtual under normal usage, due to protected inheritance.

  Having declared the dtor, other special member functions, when needed, have to be explicitly declared as defaulted:

```
template<typename T>
class stack : protected list<T> {
public:
   virtual ~stack() = default;
   stack() = default;
   stack(const stack<T>&) = default;
   stack(stack<T>&&) = default;
   stack<T>& operator=(const stack<T>&) = default;
   stack<T>& operator=(stack<T>&&) = default;
// other members omitted
};
```

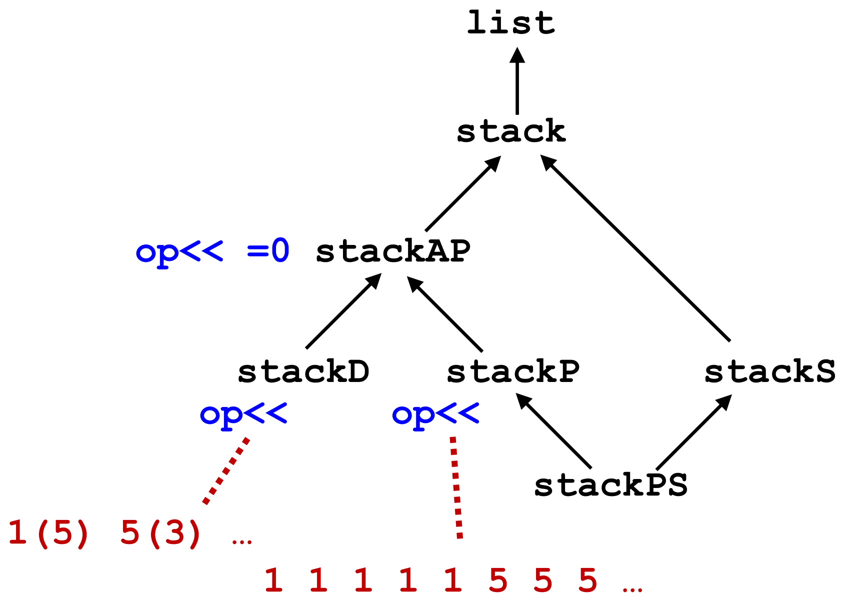
Pure virtual function

- A pure virtual function is a virtual function whose declaration ends with the pure specifier =0.
- An *abstract* class is one that 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 then specified the details by implementing 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 will invoke 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: Abstract printable stack: Coin change

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



Abstract class **stackAP**

```
using OS=ostream;

template<typename T> class stackAP;

template<typename T>
OS& operator<<(OS&,const stackAP<T>&);

template<typename T>
class stackAP : virtual public stack<T> {
friend OS& operator<< <T>(OS&,const stackAP<T>&);
private:
   virtual ostream& operator<<(ostream&) const=0;
};

template<typename T>
OS& operator<<(OS& os,const stackAP<T>& s)
{
   return s << os;
}
```

- Example (Cont'd)

Concrete class **stackP**

```
template<typename T> class stackP;

template<typename T>
OS& operator<<(OS&,const stackP<T>&);

template<typename T>
class stackP : public stackAP<T> {
friend OS& operator<< <T>(OS&,const stackP<T>&);
private:
   ostream& operator<<(ostream&) const;
};

template<typename T>
ostream& stackP<T>::operator<<(ostream& os) const
{
   for (auto& e : *this) os << e << ' ';
   return os;
}

template<typename T>
OS& operator<<(OS& os,const stackP<T>& s)
{
   return s << os;
}
```

Concrete class **stackD**

```
template<typename T> class stackD;

template<typename T>
OS& operator<<(OS&,const stackD<T>&);

template<typename T>
class stackD : public stackAP<T> {
friend OS& operator<< <T>(OS&,const stackD<T>&);
private:
   ostream& operator<<(ostream&) const;
};
```

- Example (Cont'd)

```
template<typename T>
ostream& stackD<T>::operator<<(ostream& os) const
{
   if (this->empty()) return os;
   auto d=*this->begin();
   int c=1;
   auto& it=++this->begin()
   for (;it!=this->end();++it)
      if (*it==d) c++;
      else {
         os << d << '(' << c << ") ";
         d=*it; c=1;
      }
   os << d << '(' << c << ")";
   return os;
}

template<typename T>
OS& operator<<(OS& os,const stackD<T>& s)
{
   return s << os;
}
```

Coin-change app

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

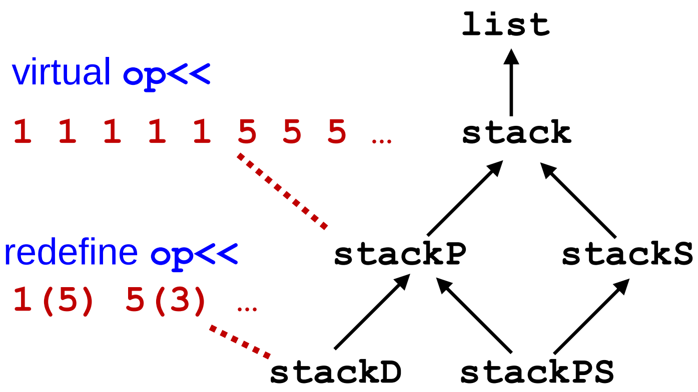
- Example (Cont'd)

```
int main()
{
    vector<int> d{1,5,10,50};
    cout << cc(30,d.size(),d,stackD<int>());
    cout << cc(30,d.size(),d,stackP<int>());
}
```

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.

For example, consider the following class lattice



This hierarchy is problematic in two aspects:

1. Every **stackD** object is a **stackP** object, meaning that wherever a **stackP** object whose elements can be printed in ***one* way** is needed, a **stackD** object whose elements can be printed in ***another* way** may be supplied. This is unreasonable.

2. If **stackD** forgets to redefine **op<<**, the compiler is unable to detect it.

- Example (Cont'd)

A safer design is to inherit the default implementation *manually*. To this end, modify the abstract class **stackAP** and the concrete class **stackP** as follows:

Abstract class **stackAP**

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

```
// default implementation of pure virtual function
template<typename T>
ostream& stackAP<T>::operator<<(ostream& os) const
{
   for (auto& e : *this) os << e << ' ';
   return os;
}
```

Concrete class **stackP**

```
// manually request the default implementation
template<typename T>
ostream& stackP<T>::operator<<(ostream& os) const
{
   return stackAP<T>::operator<<(os);
}
```

All the others remain unchanged.

Now, If **stackD** forgets to redefine **op<<**, it is an abstract class, too. Any attempt to create a **stackD** object will be denied by the compiler.

- 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

Run-time type information (RTTI)

- RTTI has three components

1	**`type_info`** class	describe type information
2	**`typeid`** operator	obtain a **`type_info`** object
3	**`dynamic_cast`** operator	browse the class hierarchy upcast, downcast, crosscast for polymorphic class only

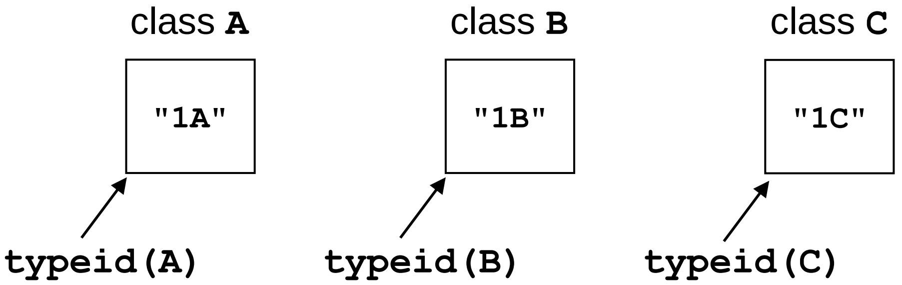
- Here is an abridged class **`type_info`**

```
class type_info {
public:
   const char* name() const { return _name.data(); }
   bool operator==(const type_info& rhs) const
   {
      return _name==rhs._name;
   }
   bool operator!=(const type_info& rhs) const;
   // other members omitted
private:
   string _name;      // name is implementation-dependent
};
```

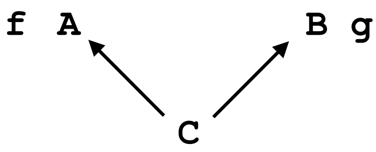
- Example

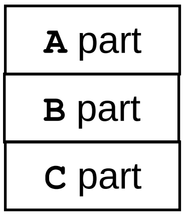
```
class A { public: virtual void f() {} };
class B { public: virtual void g() {} };
class C: public A, public B {};
```

`type_info` objects (**`g++48`**, **`clang++`**)



- Example (Cont'd)





C object

```
#include <typeinfo>
```

Upcast

```
C* pc=new C;
typeid(*pc).name()              // 1C
dynamic_cast<B*>(pc)->g();      // compile-time check
static_cast<B*>(pc)->g();       // compile-time check
pc->g();                        // compile-time check
```

Downcast

```
A* pa=new C;
typeid(*pa).name()              // 1C
dynamic_cast<C*>(pa)->g();      // runtime check, ok
static_cast<C*>(pa)->g();       // unchecked, but ok

A* pa=new A;
typeid(*pa).name()              // 1A
dynamic_cast<C*>(pa)            // fail; yield a null pointer
static_cast<C*>(pa)             // undefined
```

Crosscast

```
A* pa=new C;
typeid(*pa).name()              // 1C
dynamic_cast<B*>(pa)->g();      // runtime check, ok
static_cast<B*>(pa)->g();       // unchecked, but ok

A* pa=new A;
typeid(*pa).name()              // 1A
dynamic_cast<B*>(pa)            // fail; yield a null pointer
static_cast<B*>(pa)             // undefined
```

Single and double dispatchings

- Single dispatching
  A virtual function call that depends on the dynamic type of one object is called single dispatching.
- Double dispatching
  A virtual function call that depends on the dynamic types of two objects is called double dispatching.
- Example

```
                    virtual sleepPosture()
                    virtual meet(A&)=0
                             A
meet(A&)     B               C              D   meet(A&)
chirp()                   meet(A&)              bark()
sleepPosture()            meow()
```

Single dispatching

```
A& a=*new B;
a.sleepPosture()       // dispatch B::sleepPosture()
```

Double dispatching

```
A& a1=*new C;
A& a2=*new D;
a1.meet(a2);           // dispatch C::meet(A&), where
                       // A=D ⇒ dispatch D::bark()
```

Method 1: Use **typeid**

```
class A {
public:
   virtual void sleepPosture()
   {
      cout << "lying prone\n";
   }
   virtual void meet(A&)=0;
};
```

- Example (Cont'd)

```
class B: public A {
public:
    void chirp() { cout << "chirp\n"; }
    void sleepPosture() { cout << "standing\n"; }
    void meet(A&);
};

class C: public A {
public:
    void meow() { cout << "meow\n"; }
    void meet(A&);
};

class D: public A {
public:
    void bark() { cout << "bark\n"; }
    void meet(A&);
};

void D::meet(A& a)
{
   bark();
   if (typeid(a)==typeid(B))
      static_cast<B&>(a).chirp();
   else if (typeid(a)==typeid(C))
      static_cast<C&>(a).meow();
   else if (typeid(a)==typeid(D))
      static_cast<D&>(a).bark();
}
```

Comment

1 `static_cast` is better than `dynamic_cast`, since the downcasts here are guaranteed safe.

2 `B::meet` and `C::meet` can be defined in a like manner.

- Example (Cont'd)

  Method 2: Use **dynamic_cast**

  Pointer version

```
void D::meet(A& a)
{
   bark();
   if (B* pb=dynamic_cast<B*>(&a))
      pb->chirp();
   else if (C* pc=dynamic_cast<C*>(&a))
      pc->meow();
   else if (D* pd=dynamic_cast<D*>(&a))
      pd->bark();
}
```

  Comment

  1 Note that, in C++, variables can be declared in the condition part of an **if** statement. The scope of such a variable is the **if** statement in which it is declared.

  2 A failing **dynamic_cast** to a pointer returns a null pointer. But, a failing **dynamic_cast** to a reference throws a **bad_cast** object.

  Reference version

```
void D::meet(A& a)
{
   bark();
   try { dynamic_cast<B&>(a).chirp(); }
   catch (bad_cast&) {
      try { dynamic_cast<C&>(a).meow(); }
      catch (bad_cast&) {
         dynamic_cast<D&>(a).bark();
      }
    }
}
```

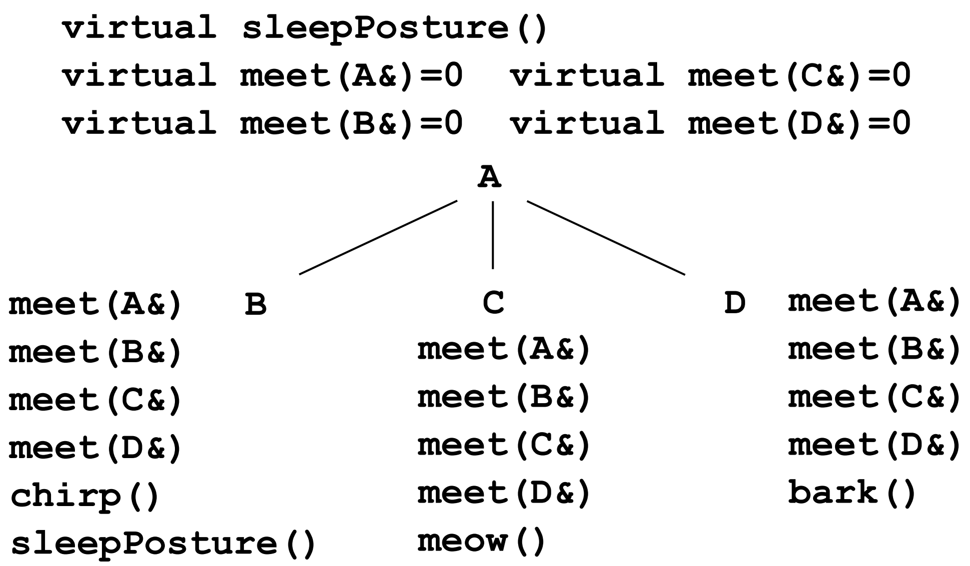
- Example (Cont'd)

Method 3: Use two single dispatchings

```
class B; class C; class D;

class A {
public:
   virtual void sleepPosture()
   {
      cout << "lying prone\n";
   }
   virtual void meet(A&)=0;
   virtual void meet(B&)=0;
   virtual void meet(C&)=0;
   virtual void meet(D&)=0;
};

class B: public A {
public:
   void chirp() { cout << "chirp\n"; }
   void sleepPosture() { cout << "standing\n"; }
   void meet(A& a) { a.meet(*this); }
   void meet(B& b) { chirp(); b.chirp(); }
   void meet(C& c); // { chirp(); c.meow(); }
   void meet(D& d); // { chirp(); d.bark(); }
};

class C: public A {
public:
   void meow() { cout << "meow\n"; }
   void meet(A& a) { a.meet(*this); }
   void meet(B& b) { meow(); b.chirp(); }
   void meet(C& c) { meow(); c.meow();  }
   void meet(D& d); // { meow(); d.bark();  }
};
```

- Example (Cont'd)

```
class D: public A {
public:
   void bark() { cout << "bark\n"; }
   void meet(A& a) { a.meet(*this); }
   void meet(B& b) { bark(); b.chirp(); }
   void meet(C& c) { bark(); c.meow(); }
   void meet(D& d) { bark(); d.bark(); }
};

void B::meet(C& c) { chirp(); c.meow(); }
void B::meet(D& d) { chirp(); d.bark(); }
void C::meet(D& d) { meow(); d.bark(); }
```



Let' see how it work

```
A& a1=*new C;
A& a2=*new D;
a1.meet(a2);
```

First, resolve to `A::meet(A&)` and dispatch `C::meet(A&)`

```
void C::meet(A& a) { a.meet(*this); }
```

Next, resolve to `A::meet(C&)` and dispatch `D::meet(C&)`

```
void D::meet(C& c) { bark(); c.meow(); }
```