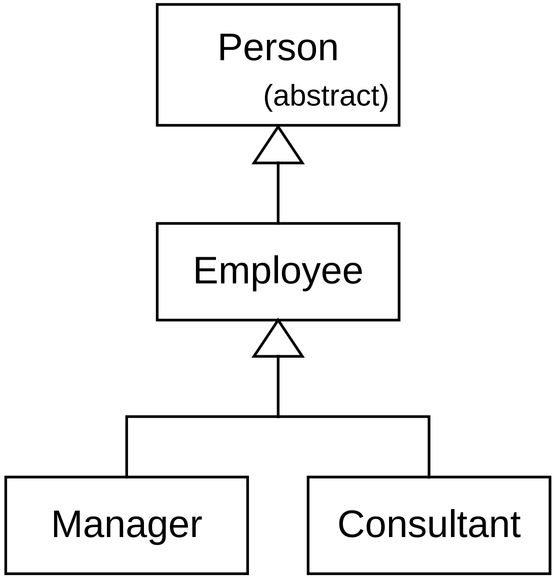
Derived classes

C++ has a relatively complete and complicated derivation mechanism.

- supports several *inheritance models*
  - *single* inheritance – only one direct base class
  - *multiple* inheritance – two or more direct base classes
  - *repeated* inheritance – an indirect base class is inherited several times through multiple inheritance
  - multiple and repeated inheritance can lead to ambiguities and other problems – need for ways to solve such – *virtual inheritance*
  - *static members*, *nested types* and *enumerators* are *class members* – can always be found unambiguously
- several ways to specify *access to base class members* in a derived class
  - **public** – **public** members of the base class are accessible as **public** in the derived class, **protected** members as **protected**
  - **protected** – **public** members of the base class are accessible as **protected** in the derived class, **protected** members as **protected**
  - **private** – **public** and **protected** members of the base class are accessible as **private** in the derived class
  - default access is **public** if a base class is a **struct**, **private** if it is a **class**
- in case of repeated inheritance the *number of subobjects* of a repeatedly inherited base class can (must) be controlled
  - **virtual** base class – in combination with one of the three above, e.g. **virtual public**
- polymorphic behaviour is controlled by the programmer
  - only *virtual functions* can be bound dynamically and have polymorphic behaviour
  - objects must be referred to by *pointers* or *references,* if virtual function calls are to be bound dynamically
  - the overhead of polymorphism can be avoided if not desired – don’t declare any virtual functions unless required

Person-Employee-Manager-Consultant – a polymorphic class hierarchy



Design of a simple polymorphic class hierarchy for different categories of employees

- class representing *persons* in general – **Person**
  - have name and civic registration number
  - all employees shall share the properties of this class
  - no Persons are to be created – shall be an *abstract class*
- class for *employees* in general – **Employee**
  - have employment date, employment number and salary, works at a department
  - Employees are to be created – shall be a concrete class
- class for employees that also are department *managers* – **Manager**
  - manages a department and its employees
- class for (temporary) employees that are *consultants* – **Consultant**
  - no actual difference to employees in general but need to be distinguishable
- objects are supposed to be referred to by pointers and created dynamically
  - otherwise no polymorphic behaviour, and the way objects are copied require dynamic allocation
  - other polymorphic type objects may be declared statically and polymorhism obtained to by reference passing

Class Person


```
class Person
{
public:
   virtual ~Person() = default;

   virtual Person* clone() const = 0;

   virtual std::string str() const;

   std::string get_name() const;
   void        set_name(const std::string&);

   CRN  get_crn() const;
   void set_crn(const CRN&);

protected:
   Person(const std::string& name, const CRN& crn);
   Person(const Person&) = default;

private:
   Person& operator=(const Person&) = delete;

   std::string name_;
   CRN         crn_;
};
```

Comments on Person

- basically a trivial class
  - well-behaved data members regarding initialization, copying/moving, and destruction
  - generated *copy constructor and move constructor* are fine, but allowed only for internal use – **protected** and **default**
  - no obvious use of move constructor, but since the copy constructor is allowed…
- *defaulted* and *deleted* member functions
  - the *default constructor* is not generated when another constructor is declared – could be *defaulted* if required
  - *copy assignment* shall not be allowed – *deleted* – access specification does not matter, it will be **private** regardless
  - *move assignment* is *not* generated because of other declared special member functions (more about this later)
  - only special member functions can be *defaulted* – any function can be *deleted*
- *virtual functions* – **virtual**
  - virtual functions can be *overridden* by subclasses
  - happens if a function with the same signature is declared in a subclass – **virtual** is then optional
  - a function in a subclass with the *same name* but with *different signature* will instead *hide*
  - makes the class *polymorphic*
- *pure virtual function*
  - a *pure specifier* = 0 makes a virtual function publicly non-callable
  - *can* have a separate definition – *must*, if a destructor – callable by other member functions, and from subclass member functions
  - pure virtual functions are inherited – a subclass becomes abstract unless all inherited pure virtual functions are overridden
  - makes the class *abstract*

Comments on Person, cont.

- *polymorphic class*
  - have virtual functions, own or inherited
  - must have a *virtual destructor* to ensure correct destruction of subobjects
  - objects will contain *type information* – used when calling virtual functions and by **dynamic_cast**
  - objects will contain a *virtual table* (e.g. `__vtable`) – implementation technique for calling virtual functions (generated by compiler)
- *abstract class*
  - no free-standing objects can be created
- *protected constructors*
  - since Person is abstract there is no need for any public constructor
  - **protected** constructors can be used to emphasizes abstractness
- *static type* and *dynamic type*

```
Person* p{ new Employee{ … } };   // p has static type "pointer to Person"

p->clone();                       // the dynamic type of the expression *p is Employee
```

  - the static type is used during compilation to check if clone() is valid for the kind og object that p kan point to
  - the dynamic type is used during execution to bind the overriding of clone() corresponding to the object p actually points to

Constructor taking name and civic registration number

```
Person::Person(const std::string& name, const CRN& crn)
   : name_{ name }, crn_{ crn }
{}
```

Ensures that a new Person always have a name an a civic registration number.

- default constructor is eliminated
- no other constructor is available that can initialize an object in some other way, except the copy and move constructor
- only to be used by corresponding direct subclass constructors – declared **protected**

Member function str()

```
virtual string Person::str() const;
```

Definition:

```
string Person::str() const
{
   return name_ + ’ ’ + crn_.str();
}
```

A call will be bound *dynamically*, if the object in question is referred to by a *pointer* or a *reference*

- the dynamic type decides which overriding to be called

```
Person* p = new Manager{ name, crn, date, employment_number, salary, dept };

cout << p->str() << endl;
```

- pointer **p** has *static type* `Person*`
- expression `*p` has *dynamic type* `Manager`

```
(*p).str()
```

- `Manager::str()` is called – we prefer the arrow operator in this case

```
p->str()
```

Member function clone()

```
virtual Person* clone() const = 0;
```

A polymorphic class needs a *polymorphic copy function.*

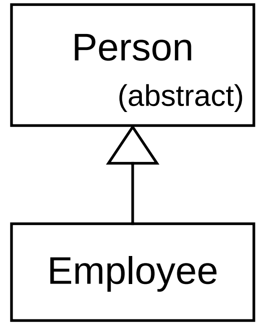
- the copy constructor could be used, but it would be very cumbersome
- polymorphic class objects are often allocated dynamically and handled with polymorphic pointers

```
Person* p = new Manager{ name, crn, date, employment_number, salary, dept };

Person* copy = p->clone();
```

- every concrete subclass must have its own, specific, overriding of clone()
  - it’s the programmer’s responsibility to ensure this
  - it’s possible to code so the compiler can check this
- suitable candidate for making Person abstract
  - we have decided to not allow Person objects a such
  - made pure virtual by the *pure specifier* (= 0)

Subclass Employee

```
class Employee : public Person
{
public:
   Employee(const std::string& name,
            const CRN&         crn,
            const Date&        e_date,
            const int          e_number,
            const double       salary,
            const int          dept = 0);

   ~Employee() = default;

   Employee* clone() const override;              // note return type!

   std::string str() const override;

   int  get_department() const;
   Date get_employment_date() const;

   int    get_employment_number() const;
   double get_salary() const;

protected:
   Employee(const Employee&) = default;
```




```
private:
   Employee& operator=(const Employee&) = delete;

   friend class Manager;
   void set_department(const int dept);
   void set_salary(const double salary);

   Date   e_date_;
   int    e_number_;
   double salary_;
   int    dept_;
};
```

Comments on Employee

- trivial class
  - same considerations as for Person
- an Employee object consists of a subobject of type Person and the specific Employee data members
  - the Person subobject is by definition initialized before the other members of Employee
  - the Person subobject is by definition destroyed after the other members of Employee
  - the only way to pass arguments to a base class constructor is by a *member initializer*
- both virtual functions are overridden
  - Employee is to be a concrete class
  - require a specific version of str()
  - clone() must be overridden for every concrete class
  - marking a virtual function with `override` makes the compiler check there is such a virtual function to be overridden

```
Employee* clone() const override;
```

  - recommended style is to *not* declare **virtual** when using `override`
- Manager is declared **friend**
  - all member functions of Manager is given unrestricted access to all Employee members, including **private** members
  - *friendship* creates stronger coupling than derivation – derivation does not give access to **private** members
  - why Employee declares Manager a friend we leave to later …

Employee's public constructor

```
Employee::Employee(const string& name,
                   const CRN&    crn,
                   const Date&   e_date,
                   const int     e_nbr,
                   const double  salary,
                   const int     dept)
   : Person{name, crn}, e_date_{e_date}, e_number_{e_nbr}, salary_{salary}, dept_{dept}
{}
```

- Person subobject is by definition initialized first
  - write the Person initializer first in the *member initializer list*
  - avoids unnecessary warnings
- Employee data members are then initialized in declaration order
  - write the member initializers in that order
  - avoids unnecessary warnings

Member function str() overridden

```
string Employee::str() const override
{
   return Person::str() + " (Employee) " + e_date_.str() + ' ' + std::to_string(dept_);
}
```

- calls `str()` for the Person subobject to produce part of the string to return
- `std::to_string()` is overloaded for all fundamental types

Member function clone() overridden

```
Employee* clone() const override
{
   return new Employee{ *this };
}
```

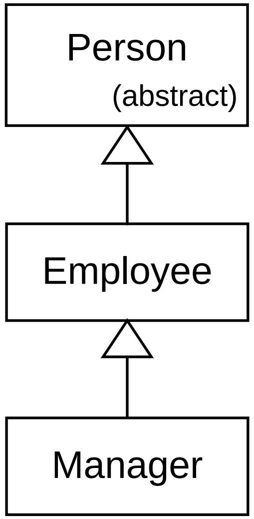
- shall create a dynamically allocated copy of the object for which clone() is called, and return a pointer to that object
- the copy constructor is the natural choice to make a copy
- since the return type belongs to a polymorphic class hierarchy we are allowed to adapt it

```
Employee* p1= new Employee{ name, crn, date, employment_nbr, salary };

Employee* p2 = p1->clone();          // no cast needed if clone() returns Employee*

Person*   p3 = p1->clone();          // implicit upcast – Employee* to Person*
```

- the types are said to be *covariant*
  - adapting the return type for clone() is allowed because of their close relation to each other

Subclass Manager

```
class Manager : public Employee
{
public:
   Manager(const std::string& name,
           const CRN&         crn,
           const Date&        e_date,
           const int          e_number,
           const double       salary,
           const int          dept);

   ~Manager() = default;

   Manager* clone() const override;

   std::string str() const override;

   void add_department_member(Employee* ep) const;
   void remove_department_member(const int e_number) const;
   void print_department_list(std::ostream&) const;

   void raise_salary(const double percent) const;

protected:
   Manager(const Manager&) = default;
```



```
private:
   Manager& operator=(const Manager&) = delete;
   // Manager& operator=(Manager&&); is not generated

   // Manager does not own the group members objects, no clean-up required
   // Employment number is key
   mutable std::map<int, Employee*> dept_members_;
};
```


Comments on Manager

- trivial class
  - well-behaved std::map member dept_members_ – default initialized to an empty map
  - otherwise same considerations and measures taken as for Employee and Person
- a Manager object consists of a subobject of type Employee, which in turn consists of a subobject of type Person, and a std::map object
  - base members are initialized top-down – Person subobject first, then Employee subobject, thereafter dept_members_
  - destruction is performed in the opposite order – dept_members_ – Employee members – Person members
- dept_members_ is declared **mutable**
  - add_department_member() och remove_department_member() modifies the object
  - we prefer to regard them as *non-modifying* operations from a public point of view – **const**
  - **mutable** allow dept_members_ to be modified by **const** member functions
- Manager was declared **friend** by Employee
  - Manager does *not* access any private members of Employee – so why?
  - we will soon find out…

Manager’s public constructor

```
Manager(const std::string& name,
        const CRN&         crn,
        const Date&        e_date,
        const int          e_number,
        const double       salary,
        const int          dept)
   : Employee{ name, crn, e_date, e_number, salary, dept }
{}
```

- all parameters are passed as arguments to direct base class Employee’s constructor
- dept_members_ have default construction – an empty employee list is created for a new Manager

Member function clone() overridden

```
Manager* clone() const override
{
   return new Manager{ *this };
}
```

Suppose we forget to override clone() in Manager.

- the *final overrider* is then Employee::clone()
- instead of a Manager clone() would return an Employee
  - copied from the Employee subobject of the Manager which was to be copied
  - Employee copy constructor creates the copy – member by member copy of the Employee data members

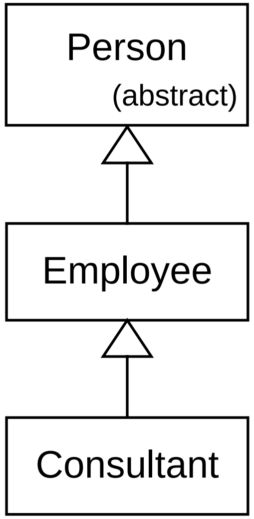
Adding and removing department employees is done by Manager

```
void Manager::add_department_member(Employee* ep) const
{
    // Set employee's department to same as manager's department
    ep->set_department(get_department());                          // require friendship

    // Add employee to department members
    dept_members_.insert(make_pair(ep->get_employment_number(), ep));
}
```

- Manager must be **friend** of Employee, to be allowed to call Employee::set_department() in this context
  - function parameter `ep` is a pointer to an Employee
  - only **public** operations are then allowed (unless Manager is a friend of Employee)
- it makes no difference if set_department() had been **protected**, Manager must still be **friend**
  - *protected access* is only allowed when the accessed object is a *subobject* of the accessing object

Consultant

```
class Consultant final : public Employee            // no subclassing
{
public:
   using Employee::Employee;                        // inheriting constructors

   ~Consultant() = default;

   Consultant* clone() const override;

   std::string str() const override;

protected:
   Consultant(const Consultant&) = default;

private:
   Consultant& operator=(const Consultant&) = delete;
};
```



Comments on Consultant

- no real difference compared to Employee
  - same data members, same set of operations
- we want to be able distinguish consultants from ordinary employees
  - subtyping is a way to allow for that by dynamic type checks
- marking a class `final`

```
class Consultant final : public Employee
```

  - not allowed to derive from Consultant
- marking a virtual function `final`

```
string str() const override final;
```

  - such a function is not allowed to override in subclasses
- *inheriting constructors*

```
using Employee::Employee;
```

  - naming a constructor in a nested using declaration opens for inheriting constructors from a base class
  - the public constructor for Manager and Consultant is inheried from Employee
  - our special member functions must still be declared

The using declaration in class scope

- a *using-declaration* introduces a name in the declarative region in which it appears

```
using name;        // an alias for the name of some entity declared elsewhere
```

- can be used in *class scope* to introduce names from a base class

```
using Base::hidden;
```

  – inheriting constructors

```
using Base::Base;
```

- the alias created has the usual accessibility for a member declaration
  – an alias can be a *public* alias for a *protected* member
  – a *using-declaration* can *not* make a *private* member of a base class (more) accessible
- member functions in the derived class *override* and/or *hide* member functions with the same name and parameter types in the base class
- can *not* be used to resolve inherited member ambiguities

Note: A *using-directive* can not appear in class scope, so the following is *not* allowed in class scope:

```
using namespace std;
```


Using the using declaration in class scope

```
class A
{
   public:
      void f();
      void h();          // h can be named in a using declaration, since no private h
   protected:
      void h(int);
      void g();          // g can be named in a using declaration, since no private g
      void g(int);
      void p();
   private:
      void f(int);       // using A::f not possible (access control)
      void p(int);       // using A::p not possible (access control)
};

class B : public A
{
   public:
      using A::h;        // OK
      using A::g;        // OK
      using A::p;        // error: void A::p(int) is private within this context
   private:
      using A::f;        // error: void A::f(int) is private within this context
};
```

Inheritance and special member functions

- The *default constructor* and the *copy/move constructors* are *not* inherited,
  - implicitly-declared in a derived class, if not user declared
- The *destructor* is *not* inherited,
  - implicitly-declared, if not user declared
- *Operator functions are* inherited, *but*
  - inherited *copy/move assignment operators* are always *hidden*, either by implicitly-declared or user declared copy/move assignment operators of the derived class
- A *using-declaration* that brings in a *copy/move constructor* or a *copy/move assignment operator* from the base class is *not* concidered an explicit declaration and does *not* supress the implicit declaration of these functions in the derived class. Such special member functions introduced by a using-declaration is therefore hidden by the implicit declaration.
- A using-declaration cannot refer to a destructor for a base class.
- Other constructors brought in from a base class by a using-declaration *are* inherited.

For Consultant this gives that

- the *default constructor* is not inherited (never is), and is not implicitly-declared since other constructors are declared
- the *copy/move constructors* are not inherited (never is) – are user declared (defaulted)
- the destructor is not inherited (never is) – is user declared (defaulted)
- the *copy assignment operator* is inherited but hidden, as always, in this case by a user-declared (deleted) copy assignment operator
- the move assignment operator is not implicitly-declared since, e.g., the copy assignment operator is explicitly declared
- the user-declared public contructor of Employee is inherited, because of the using-declaration **using** Employee::Employee;

The NVI pattern – Non-Virtual Interface

```
class Person
{
public:
    …
    std::string str() const;
    Person* clone() const;
    …
private:
    …
    virtual std::string to_str() const;
    virtual Person* make_clone() const = 0;
};
```

- public str() and clone() are declared *non-virtua*l
- implemented by call-through to a corresponding private virtual function to_str() and make_clone(), respectively
- subclasses override to_str() and make_clone()

The *Non-Virtual Interface* pattern eliminates the problem that a public virtual function really do two things:

- specifies interface
- specifies implementation – namely internal customizable behaviour

NVI keeps public interface apart from implementation – makes it easier to modify implementation without affecting clients.

NVI str() and to_str()

- the non-virtual public member str() is inherited by subclasses

```
std::string Person::str() const
{
   return to_str();              // explicitly this->to_str()
}
```

  - a call to str() will be bound *statically*
  - a call to to_str() will be bound *dynamically*
  - the type of the **this** pointer reflects the type of the object that have called the function
- definition of to_str() for Person

```
string Person::to_str() const
{
   return name_ + ' ' + crn_.str();
}
```

  - to be overridden by subclasses, might be used by subclass implementation

Note: It *is* allowed to override a virtual function, even if it is declared **private** in the base class.

NVI clone() and make_clone()

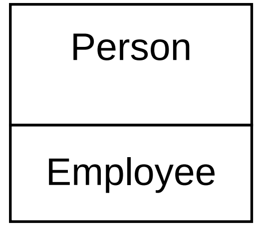
```
Person* Person::clone() const
{
   Person* p = make_clone();      // explicitly: this->make_clone()

   // assert will fail if the copy *p doesn't have same type as the original, *this
   assert(typeid(*p) == typeid(*this) && "make_clone() incorrectly overridden");

   return p;
}
```

- the non-virtual public member clone() is inherited by subclasses
  - the string literal "make_clone() incorrectly overridden" is converted to **true** in this context
  - assert will not fail if the copied object (*p) and the original (***this**) have same type – **true** && **true** is **true**
  - the purpose of the right hand side of `&&` is that this text is to appear in the error message when the actual assertion fails
- make_clone() for Person is pure virtual

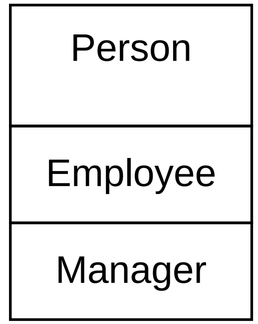
```
virtual Person* make_clone() const = 0;
```

  - every concrete subclass must override make_clone()
  - the assert will check this, but unfortunately not until runtime

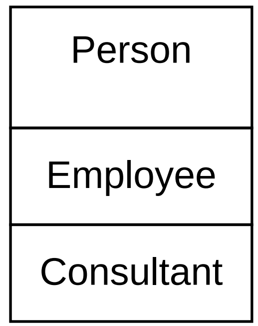
Initialization and destruction of objects of derived type



- an object of derived type is made up of parts, subobjects.
  - base class subobjects and the data members of the class in question
  - the *“most derived type”*
- initialization order is top-down (and left-to-right if multiple inheritance)
  - base class subobjects are initialized before subclass subobjects
  - the first constructor to be called is that of *the most derived class*
  - direct base class constructors are called recursively
  - data members are initialized in the same order as they are declared within the class



- destruction order is reverse to initialization order – bottom-up (and right-to-left if multiple inheritance)
  - data members are destroyed in the reverse order to how they are declared within the class
  - direct base class destructors are then called, recursively
- if virtual inheritance
  - all virtual base class subobjects are initialized first of all – top-down, left-to-right
  - the non-virtual subobjects are initialized/destroyed as described above
  - all virtual base class objects are destroyed last of all – bottom-up, right-to-left
- important that the top-most polymorphic base class have a virtual destructor




```
Person* p = new Consultant{ … };
…
delete p;        // ~Person() or ~Consultant() ?
```

Using Person-Employee-Manager-Consultant

```
Person*     pp;  // can point to an Employee or a Manager or a Consultant object
Employee*   pe;  // can point to an Employee or a Manager or a Consultant object
Manager*    pm;  //  can only point to an Manager object (since no subclasses to Manager)
Consultant* pc;  // can only point to a Consultant object (ditto)


pm = new Manager{ name, crn, date, employment_nbr, salary, 17 };

pp = pm;                                  // upcast is automatic – Manager* -> Person*

pm = dynamic_cast<Manager*>(pp);          // downcast must be explicit – Person* -> Manager*

if (pm != nullptr)                        // do we have a Manager?
{
   pm->print_department_list(cout);
}
```

- polymorphic pointers – can point to objects of the pointee type and subtypes
- *upcast* is an automatic and safe type conversion
- *downcast* must be explicit and possibly checked before use
  - print_department_list() is specific for Manager and can only be invoked using a pointer of type Manager* (or reference Manager&)
- the object is not affected – once a Manager is always a Manager

Dynamic type check and dynamic type conversion

- sometimes you need to find out type information for a polymorphic object during execution
  - what kind of object does a polymorphic pointer point to?
  - what kind of object does a polymorphic reference refer to?
- sometimes you need to do *dynamic type conversion*, e.g.
  - if a subclass have a member function which is not inherited, as e.g. Manager::print_department_list()
  - *and* an object of the subclass is referred by a base class pointer or a base class reference
  - *and* you want to call the member function
- **typeid** expressions can be used to
  - check if an object have a *specific type*
  - check the type of an expression
  - can be applied to all types
  - include <typeinfo>
- **dynamic_cast** can be used
  - only for *polymorphic types* – require the type information such objects keep
  - to check if an object have a specific *type* or is a *subtype* to some type
  - to convert a polymorphic pointer or a polymorphic reference
- **static_cast** is possible to use also when converting polymorphic pointers and references, but without dynamic type checks
  - you need to be absolutely sure it’s correct to do

Dynamic type control using typeid expressions

One way to find out the type of an object is to use **`typeid`**

```
if (typeid(*p) == typeid(Manager)) …
```

- can be used for *type names*, all kind of *objects*, and all kind of *expressions*
- a **`typeid`** expression returns a `type_info` object (a class type)
- type checking is done by comparing two `type_info` objects

`typeid` expressions:

typeid(*p)	// `p` *is a pointer to an object of some type*
typeid(r)	// `r` *is a reference to an object of some type*
typeid(T)	// `T` *is a type*
typeid(p)	// *is usually a mistake if* `p` *is a pointer*

`type_info` operations:

`==`	*check if two* `type_info` *objects are equal* – **typeid**(*p) == **typeid**(T)
`!=`	*check if two* `type_info` *objects are not equal* – **typeid**(*p) != **typeid**(T)
`name()`	*returns the type name as a string – may be an internal name used by the compiler, a "mangled name"*

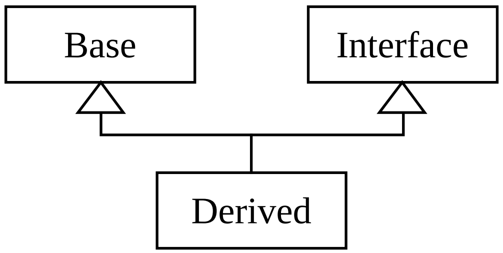
Dynamic type conversion using dynamic_cast

`dynamic_cast` can be used to type convert polymorphic pointers and references.

```
dynamic_cast<T*>(p)        // convert p to "pointer to T"
dynamic_cast<T&>(r)        // convert r till "reference to T"
```

- typically used to *"downcast"*
  - from a *base type pointer* to *subtype pointer*
  - from *base type reference* to *subtype reference*
- if conversion fails
  - in the pointer case, 0 is returned
  - in the reference case, exception `bad_cast` is thrown
- *"upcast"* is automatic and safe
- in case of multiple inheritance *"crosscast"* can be of interest

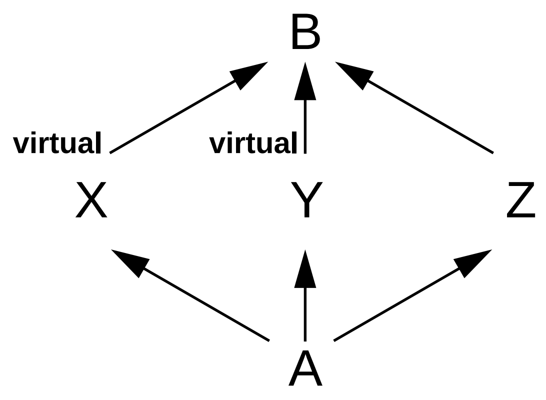
```
class Derived : public Base, public Interface { … };

Base* pb{ new Derived };

Interface* pi = dynamic_cast<Interface*>(pb);

if (pi != nullptr) ...
```



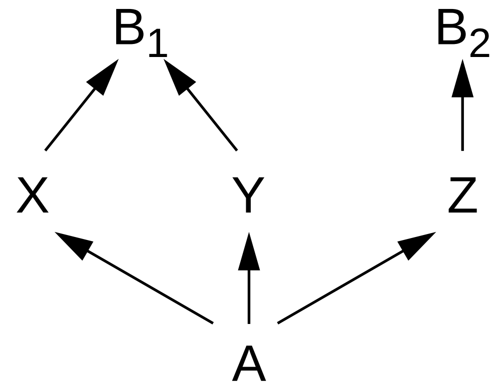
Virtual base classes – class lattice – subobject lattice

```
class B {
public:
   int i;
   static int s;
   enum { e };
};

class X : virtual public B { … };

class Y : virtual public B { … };

class Z : public B { … };

class A : public X, public Y, public Z { … };
```

class lattice:



subobject lattice:

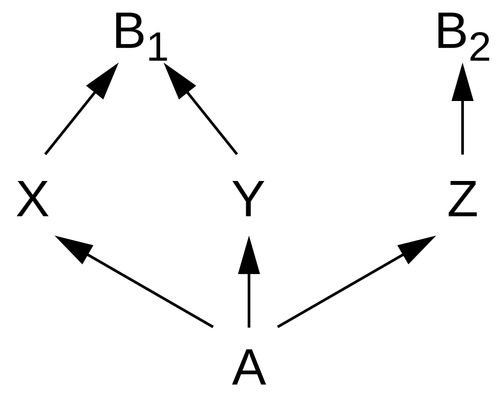



```
void F(A* p)
{
   p->i++;              // Ambiguous: two i in A
   p->X::i++            // OK: qualification specifies
   p->s++;              // OK: only one s (static)
   p->s = p->e          // OK: only one e (enumerator)
}
```

Initialization and destruction of derived class objects with virtual base classes

When an object of type **A** is created the following takes place.

subobject lattice:



- first the constructor for the most derived class – **A** – is called
  - constructors for all virtual subobjects are called top-down, left-to-right ($\mathbf{B_1}$)
  - a member initializer for **B** is required in **A**, unless default initialized
- then the direct non-virtual base subobjects to **A** are constructed in declaration order – **X**, **Y**, **Z** – recursively
  - any non-virtual base subobjects are constructed
  - any non-static data member subobjects are constructed in declaration order
  - the constructor body is executed

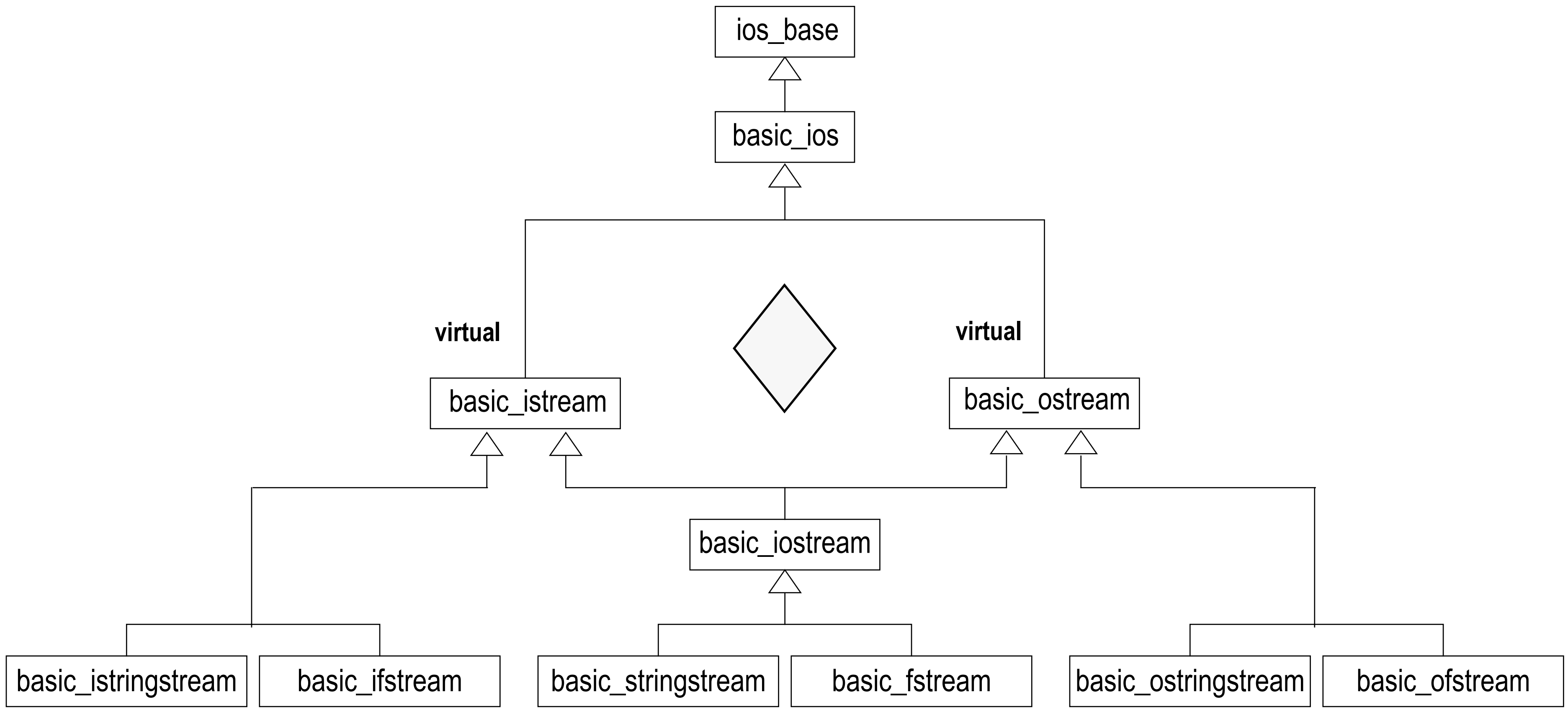
Initialization order: `B`$_1$ `-> X -> Y -> B`$_2$ `-> Z -> A`

- an object comes into existence first after its constructor body has succeded
  - if construction fails, already constructed subobjects are destroyed in reverse construction order
  - the subobject that construction failed for never existed
- all non-abstract classes in a class lattice must have initializers for virtual base classes, unless virtual bases have default construction
  - initializers for virtual bases are ignored in all constructors except in the constructor for the most derived class

  X, **Y** and **A** (if non-abstract) must all have an member initializer for **B**, or rely on a default constructor.

An example of real use of virtual base classes

The standard stream classes:



Comments on derived class design

- prevent subclassing by marking a class `final`
- make compiler check overridings by marking virtual function declarations `override`
- prevent further overriding by marking a virtual function `final`
- make a base class destructor
  - **public** and **virtual**, if deletion through a base class pointer should be allowed (typically for polymorphic types)
  - **protected** and *non-virtual* otherwise, if class is abstract
  - a compiler generated destructor is **public** and *non-virtual (*unless there is a base class having a virtual destructor)
  - if a base class has a virtual destructor, generated subclass destructors will also be virtual
- *default* and *delete* special member functions properly
  - if the copy/move constructors are desired and can be generated – *default* with proper access (**public**, **protected**)
  - if copying is *not* allowed, *delete* the copy constructor – the move constructor is not generated
  - if the copy constructor is declared but the move constructor is not desired – *do not* delete the move constructor!
  - an explicitly deleted member function will implicitly be **private** – compile error if chosen in overload resolution
  - analogous for copy assignment and move assignment operators
- avoid calling **virtual** functions in constructors and destructors
  - virtual functions don’t behave virtual in constructors and destructors
  - a Manager object have dynamic type Person when the Person constructor/destructor is executing
  - use som “post-construction” technique if virtual dispatch into a derived class is needed from a base constructor

Comments on derived class design, cont.

- avoid *slicing*
  - slicing is automatic, invisible and likely to create problems
  - typically occur when a function has a value parameter of base type (& forgotten?)
  - to allow for *explicit* slicing the copy constructor can be declared **explicit**, to avoid unexpected use

```
explicit C::C(const C);
```

- consider a virtual copy function for copying polymorphic types – clone()
  - prevents slicing
  - other ways to copy should be restricted to internal use only – make copy/move constructors protected, e.g.
- consider making **virtual** functions *non-public* and **public** functions *non-virtual*
  - separates the public interface from the customization interface – the NVI pattern
  - especially interesting for base classes with a high cost of change (libraries, frameworks, etc.)
- consider containment instead of inheritance
  - prefer containment if no real gain using derivation

