Class

Class

```
class X {
  /*...*/
};
```

- Class in C++ is an extremely powerful construct
  - Other languages often have several less powerful constructs (class+interface)
  - Requires caution and conventions
- Three degrees of usage
  - Non-instantiated class - a pack of declarations (used in generic programming)
  - Class with data members
  - Class with inheritance and virtual functions (object-oriented programming)
    - Only these classes carry their type information at runtime
- class = struct
  - struct members are by default public
    - by convention used for simple or non-instantiated classes
  - class members are by default private
    - by convention used for large classes and OOP

Three degrees of classes

Non-instantiated class

```
class X {
public:
  typedef int t;
  static constexpr
    int c = 1;
  static int f( int p)
  { return p + 1; }
};
```

Class with data members

```
class Y {
public:
  Y()
    : m_( 0)
  {}
  int get_m() const
  { return m_; }
  void set_m( int m)
  { m_ = m; }
private:
  int m_;
};
```

Classes with inheritance

```
class U {
public:
  virtual ~U() {}
  void f()
  { f_(); }
private:
  virtual void f_() = 0;
};

class V : public U {
public:
  V() : m_( 0) {}
private:
  int m_;
  virtual void f_()
  { ++ m_; }
};
```

Type and static members of classes

```
class X {
public:
  class N { /*...*/ };
  typedef unsigned long t;
  using t2 = unsigned long;
  static constexpr t c = 1;
  static t f( t p)
  { return p + v_; }
private:
  static t v_;  // decl. of X::v_
};

X::t X::v_ = X::c;  // def. of X::v_

void f2()
{
  X::t a = 1;
  a = X::f( a);
}
```

- Type and static members...
  - Nested class definitions
  - typedef/using definitions
  - static member constants
  - static member functions
  - static member variables
- ... are not bound to any class instance (object)
- Equivalent to global types/variables/functions
  - But referenced using qualified names (prefix X::)
  - Encapsulation in a class avoids name clashes
    - But namespaces do it better
  - Some members may be private
  - Class may be passed to a template

Uninstantiated class

- Class definitions are intended for objects
  - Static members must be explicitly marked
- Class members may be public/protected/private

```
class X {
public:
  class N { /*...*/ };
  typedef unsigned long t;
  static constexpr t c = 1;
  static t f( N p);
private:
  static t v;  // decl. of X::v
};
```

- Class must be defined in one piece
  - Except of definitions placed outside

```
X::t X::v = X::c;      // def. of X::v
X::t X::f( N p) { return p.m + v; }
```

- Access to members requires qualified names

```
void f2()
{
  X::N a;
  auto b = X::f( a);
}
```

- A class may become a template argument
  - This is the (only) reason for uninstantiated classes

```
using my_class = some_generic_class< X>;
```

Namespace

- Namespace members are always static
  - No objects can be made from namespaces
  - Functions/variables are not automatically inline/extern

```
namespace X {
  class N { /*...*/ };
  typedef unsigned long t;
  constexpr t c = 1;
  extern t v;    // decl. of X::v
};
```

- Namespace may be reopened and member declarations added
  - Namespace may be split into several header files

```
namespace X {
  inline t f( N p) { return p.m + v; }
};
```

  - Definitions of previously declared namespace members may be outside

```
X::t X::v = X::c;   // def. of X::v
```

Namespaces

- Namespace members are always static
  - No objects can be made from namespaces
  - Functions/variables are not automatically inline/extern

```
namespace X {
  class N { /*...*/ };
  typedef unsigned long t;
  constexpr t c = 1;
  extern t v;   // decl. of X::v
};
```

- Namespace may be reopened and member declarations added
  - Namespace may be split into several header files

```
namespace X {
  inline t f( N p) { return p.m + v; }
};
```

  - Definitions of previously declared namespace members may be outside

```
X::t X::v = X::c;   // def. of X::v
```

```
void f2()
{
  X::N a;
```

  - Functions in namespaces are visible by *argument-dependent lookup*

```
  auto b = f( a);
```

    - calls X::f because the class type of a is a member of X
  - Namespace members can be made directly visible

```
  using X::t;
  t b = 2;
  using namespace X;
  b = c;
}
```

Class with data members

```
class Y {
public:
  Y()
    : m_( 0)
  {}
  int get_m() const
  { return m_; }
  void set_m( int m)
  { m_ = m; }
private:
  int m_;
};
```

- Class (i.e. type) may be instantiated (into objects)
  - Using a variable of class type

```
Y v1;
```

    - This is NOT a reference!
  - Dynamically allocated
    - Held by a (smart) pointer

```
auto p = std::make_unique< Y>();
auto q = std::make_shared< Y>();
```

  - Element of a larger type

```
typedef std::array< Y, 5> A;
class C1 { public: Y v; };
class C2 : public Y {};
```

    - Embedded into the larger type
    - NO explicit instantiation by new!

  - Membership (C1::v) and inheritance (C2) are fairly similar
    - The same physical layout
    - Inheritance may have unintended consequences
    - Use inheritance only when you need to override virtual functions

Class with data members

```
class Y {
public:
  Y()
    : m_( 0)
  {}
  int get_m() const
  { return m_; }
  void set_m( int m)
  { m_ = m; }
private:
  int m_;
};
```

- Class (i.e. type) may be instantiated (into objects)

```
Y v1;
auto p = std::make_unique< Y>();
```

  - Non-static data members constitute the object
  - Non-static member functions are invoked on the object
  - Object must be specified when referring to non-static members

```
v1.get_m()
p->set_m(0)
```

    - References from outside may be prohibited by "private"/"protected"

```
v1.m_  // error
```

    - Only "const" methods may be called on const objects

```
const Y * pp = p.get();
pp->set_m(0) // error
```

Inheritance and virtual functions

Inheritance

```
class Base { public:
  virtual ~Base() noexcept {}
  virtual void f() { /* ... */ }
};
class Derived final : public Base {
  virtual void f() override { /* ... */ }
};
```

- Derived class
  - Contains all types, data elements and functions of Base
    - Because of this, a pointer/reference to Derived may be silently converted to a pointer/reference to Base
    - The opposite conversion is available as explicit cast
  - New types/data/functions may be added
    - Hiding old names by new names is not wise, except for virtual functions
  - Functions declared as **virtual** in Base may change their behavior by reimplementation in Derived
    - private virtual functions may be overridden too
    - **override** – verify existence of this virtual function in (some of) the base classes
  - Virtual destructor needed in Base to ensure proper delete
  - **final** – disable derivation from this class

Classes in inheritance

- Abstract class
  - Definition in C++: A class that contains some pure virtual functions

```
virtual void f() = 0;
```

    - Such class is incomplete and cannot be instantiated alone
  - General definition: A class that will not be instantiated alone (even if it could)
  - Defines the interface which will be implemented by the derived classes

- Concrete class
  - A class that will be instantiated as an object
  - Implements the interface required by its base class

Virtual functions

```
class Base { public:
  virtual ~Base() noexcept {}
  virtual void f() { /* ... */ }
};
class Derived : public Base { public:
  virtual void f() { /* ... */ }
};
```

- Virtual function call works only in the presence of pointers or references

```
std::unique_ptr<Base> p = std::make_unique< Derived>();// automatic conversion
p->f();   // calls Derived::f although p is pointer to Base
```

- Without pointers/references, having functions virtual has no sense

```
Derived d;
d.f();       // calls Derived::f even for non-virtual f

Base b = d;  // slicing = copying a part of an object
b.f();       // calls Base::f even for virtual f
```

- Slicing is specific to C++
  - Often prohibited due to Base being abstract

Dynamic cast

`dynamic_cast<T>(e)`

- Base-to-derived pointer/reference conversions
  - Runtime checks included – requires type information in the **e** object
    - At least one virtual function required in the type of **e**
  - If the dynamic type of **e** is not **T** (or derived from T)...
    - Pointers: Returns **nullptr**
    - References: Throws **std::bad_cast**

```
class Base { public:
  virtual ~Base(); /* base class must have at least one virtual function */
};
class X : public Base { /* ... */
};
class Y : public Base { /* ... */
};

Base * p = /* ... */;
X * xp = dynamic_cast< X *>( p);
if ( xp ) { /* ... */ }
Y * yp = dynamic_cast< Y *>( p);
if ( yp ) { /* ... */ }
```

Inheritance and the destructor

```
class Base {
public:
  virtual ~Base() noexcept {}
};

class Derived : public Base {
public:
  virtual ~Derived() noexcept {/**/}
};
```

- Old-style

```
Base * p = new Derived;
delete p;
```

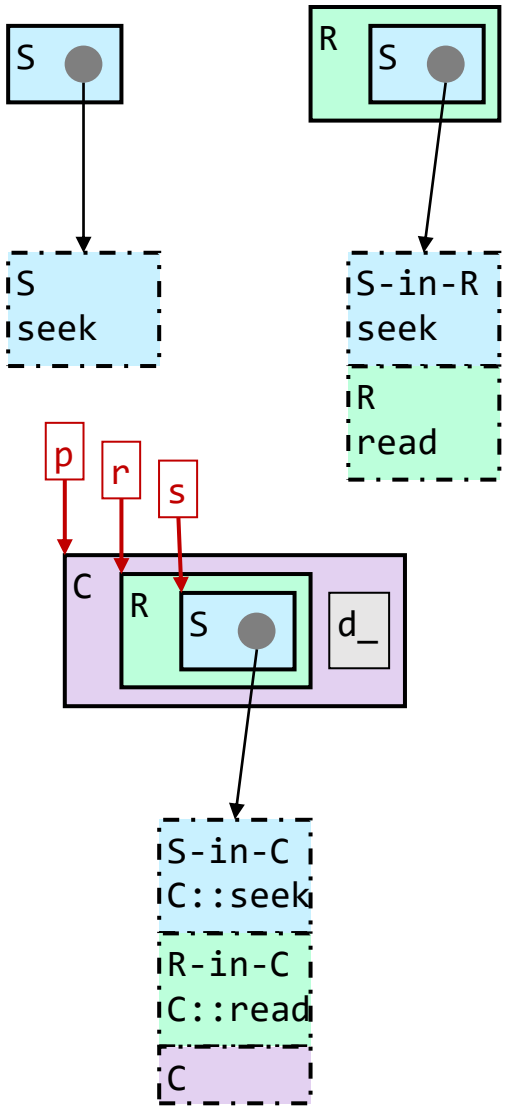
- Modern-style

```
{
  std::unique_ptr<Base> p =
  std::make_unique< Derived>();
}
```

- Language rule:
  - If an object is destroyed using a pointer to a base class, the base class must have a *virtual* destructor
  - This triggers the more complex implementation of delete:
    - Correctly destruct the complete object
    - Correctly determine the memory block
- Recommendation:
  - Every abstract class shall have a virtual destructor
    - Cost is negligible because other virtual functions are present
    - A pointer to the abstract class will likely be used for destruction

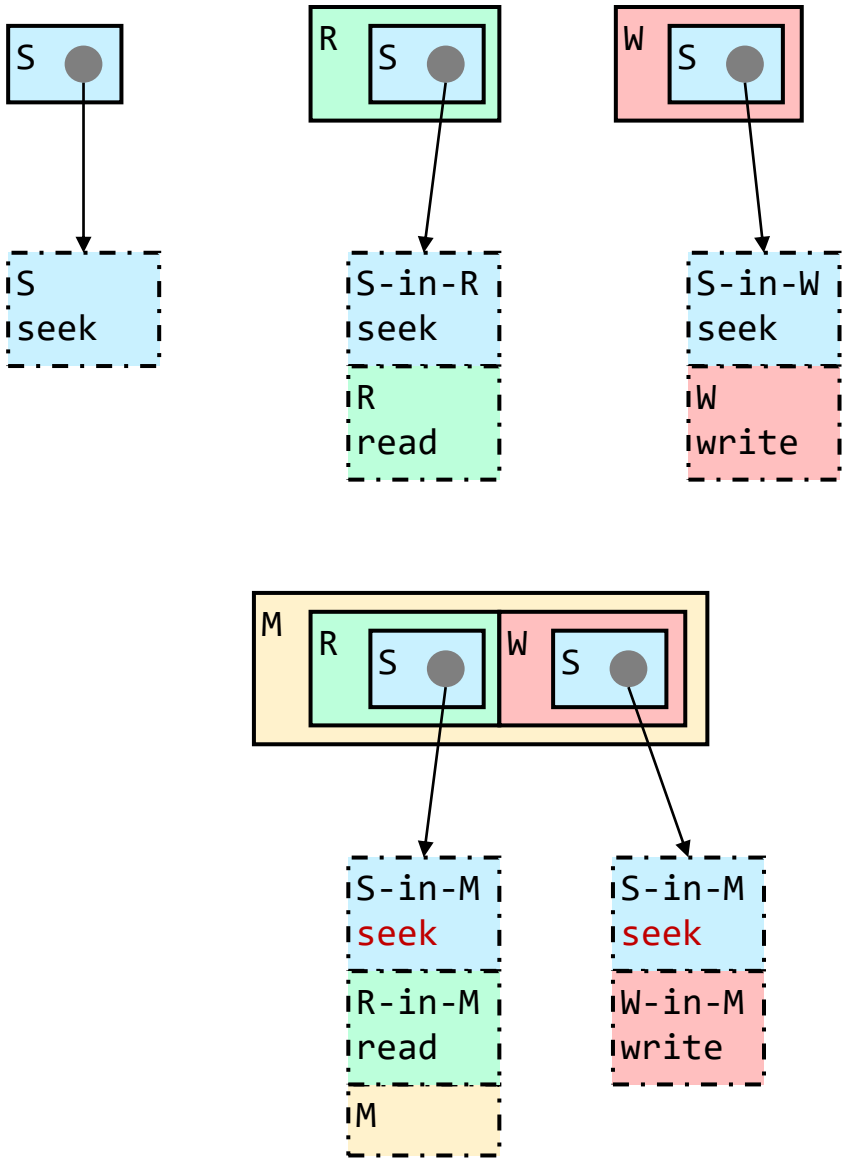
Single non-virtual inheritance - example

```
class S
{ public:
    virtual ~S() = default;
    virtual void seek(int) = 0;
};

class R
: public S
{ public:
    virtual int read() = 0;
};

class C
: public R
{
    virtual void seek(int) {/**/}
    virtual int read() {/**/}
    std::istream d_;
};

auto p = std::make_unique<C>();
std::unique_ptr<R> r = move(p);
S* s = &*r;
r.reset();   // C::~C()
```



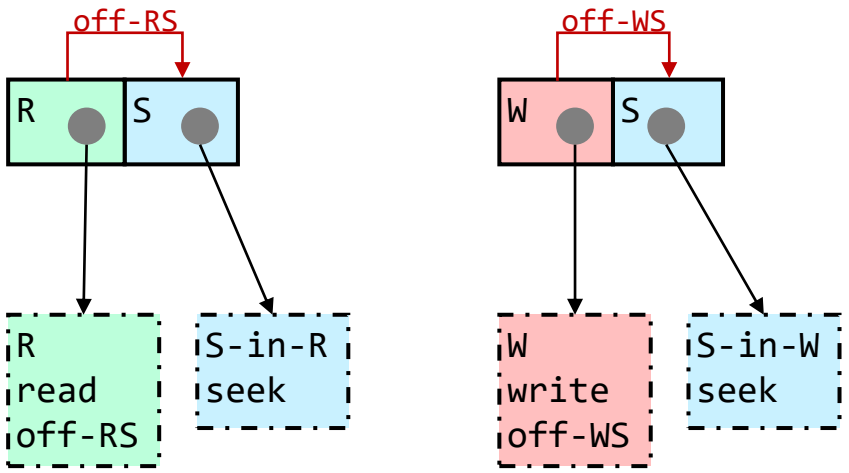
Multiple non-virtual inheritance - example

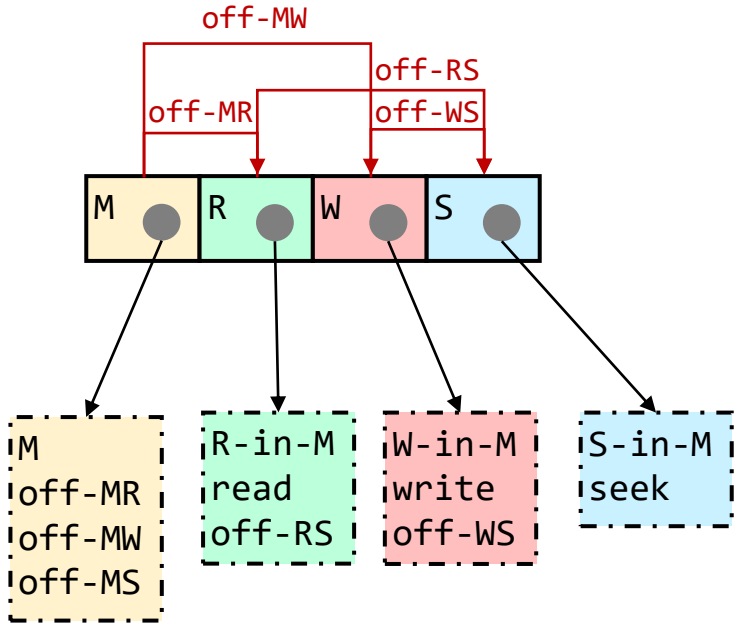
```
class S
{ public:
    virtual ~S() = default;
    virtual void seek(int) = 0;
};

class R
: public S
{ public:
    virtual int read() = 0;
};

class W
: public S
{ public:
    virtual void write(int) = 0;
};

class M
: public R,
  public W
{
    virtual void seek(int) {/**/}
    // ERROR: which seek?
};
```



Virtual inheritance - example

```
class S
{ public:
    virtual ~S() = default;
    virtual void seek(int) = 0;
};

class R
: public virtual S
{ public:
    virtual int read() = 0;
};

class W
: public virtual S
{ public:
    virtual void write(int) = 0;
};

class M
: public virtual R,
  public virtual W
{};

void copy(W &, R &);
void sort(M &);
```





Inheritance

- Inheritance mechanisms in C++ are very strong
  - Often misused
- Inheritance shall be used only in these cases
  - IS-A hiearachy
    - Eagle IS A Bird
    - Square-Rectangle-Polygon-Drawable-Object
  - Interface-implementation
    - Readable-InputFile
    - Writable-OutputFile
    - (Readable+Writable)-IOFile

Inheritance

- ISA hierarchy
  - C++: Single non-virtual public inheritance

```
class Derived : public Base
```

  - Abstract classes may contain data (although usually do not)

- Interface-implementation
  - C++: Multiple virtual public inheritance

```
class Derived : virtual public Base1,
        virtual public Base2
```

    - virtual inheritance merges copies of a base class multiply included via diamond patterns
  - Abstract classes usually contain no data
  - Interfaces are (typically) not used to own (destroy) the object

- Often combined

```
class Derived : public Base,
        virtual public Interface1,
        virtual public Interface2
```

Misuse of inheritance

- Misuse of inheritance - #1

```
class Real { public: double Re; };
class Complex : public Real { public: double Im; };
```

- Leads to slicing:

```
double abs( const Real & p) { return p.Re > 0 ? p.Re : - p.Re; }

Complex x;
double a = abs( x); // it CAN be compiled - but it should not
```

- Reference to the derived class may be assigned to a reference to the base class
  - Complex => Complex & => Real & => const Real &

Misuse of inheritance

- Misuse of inheritance - #2

```
class Complex { public: double Re, Im; };
class Real : public Complex { public: Real( double r); };
```

- Mistake: Objects in C++ are not mathematical objects

```
void set_to_i( Complex & p) { p.Re = 0; p.Im = 1; }

Real x;
set_to_i( x);   // it CAN be compiled - but it should not
```

- Real => Real & => Complex &

Two worlds of classes in C++

Classes without inheritance

- No virtual functions
- No visible pointers usually required
  - When multiple objects exist
    - Allocated usually via containers

```
std::vector< MyClass> k;
```

  - When standalone

```
MyClass c;
```

- If ownership must be transferred, moving may be used

```
std::vector< MyClass> k2 = move( k);
MyClass c2 = std::move( c);
```

- Move required
  - For insertion into containers
  - For transfer of ownership
- Copy often required too

- Individual allocation required only if
  - ownership must be transferred
  - and observers are required

```
auto p = std::make_unique< MyClass>();
MyClass * observer = p.get();
auto p2 = move( p);
```

Classes with inheritance

- Concrete classes of different size and layout
  - Usually mixed in a data structure
  - Cannot be allocated in a common block
  - Individual dynamic allocation
- Common base class
  - Serves as a unified handle for different concrete classes
  - Pointers required

```
std::vector<std::unique_ptr<Base>> k;
```

  - Virtual destructor required
- Copy/move not required/supported
  - Pointers are copied/moved instead
  - Objects often have identity

Conversions

Special member functions

- Conversion constructors

```
class T {
  T( U x);
};
```

  - Generalized copy constructor
  - Defines conversion from U to T
  - If conversion effect is not desired, all one-argument constructors must be "explicit":

```
explicit T( U v);
```

- Conversion operators

```
class T {
  operator U() const;
};
```

  - Defines conversion from T to U
  - Returns U by value (using copy-constructor of U, if U is a class)
    - U may be a reference like V& if life-time considerations allow

- Compilers will never use more than one user-defined conversion in a chain
  - The user-defined conversion may be combined with several built-in conversions

Type cast

- Various syntax styles
  - C-style cast

```
(T)e
```

    - Inherited from C
  - Function-style cast

```
T(e)
```

    - Equivalent to (T)e
    - T must be single type identifier or single keyword
  - Type conversion operators
    - Differentiated by intent (strength and associated danger) of cast:

```
const_cast<T>(e)
static_cast<T>(e)
reinterpret_cast<T>(e)
```

    - New - run-time assisted cast:

```
dynamic_cast<T>(e)
```

Const cast

`const_cast<T>(e)`

- Suppressing const flags of pointers/references
  - const U & => U &
  - const U * => U *

- It allows violation of const-ness
- In most cases, **mutable** is a better solution
  - Example: Counting references to a logically constant object

```
class Data {
public:
  void register_pointer() const
  { references++; }
private:
  /* ... data ... */
  mutable int references;
};
```

Static cast

`static_cast<T>(e)`

- All implicit conversions
  - Explicit cast used to enforce the conversion in ambiguous situations
  - Loss-less and lossy number conversions (e.g. int <=> double)
  - Adding const/volatile modifiers to pointers/references
  - Pointer to void*
  - Derived-to-base pointer/reference conversions
  - Invoke any constructor of T capable to accept e
    - Including copy/move-constructors and explicit constructors
  - Invoke a conversion operator T()
- Some explicit conversions
  - Anything to void, i.e. discarding the value (e.g. in a conditional expression)
  - Base-to-derived pointer/reference conversions
    - No runtime checks, it may produce invalid pointers – use **dynamic_cast** to check
  - Integer to an enumeration
    - May produce undefined results if not mappable
  - void* to any pointer
    - No runtime checks possible (even if the object contain type information)

Reinterpret cast

`reinterpret_cast<T>(e)`

- Implementation-dependent conversions
  - Pointer to integer
  - Integer to pointer
  - Any function-pointer to any function-pointer
  - Any data-pointer to any other data-pointer
    - No address correction even if pointers are related by inheritance
  - Any reference to any other reference

    - Mostly used to read/write binary files/packets/...

```
void put_double( std::ostream & o, const double & d)
{
  o.write( reinterpret_cast< char *>( & d), sizeof( double));
}
```

- The file contents is implementation-dependent – not portable