# Dědičnost
# Filosofický pohled

# Dědičnost

- Mechanismus dědičnosti v C++ je velmi silný
  - Bývá používán i pro nevhodné účely
- Ideální použití dědičnosti je pouze toto
  - IS-A hierarchie (typicky pro objekty s vlastní identitou)
    - Živočich-Obratlovec-Savec-Pes-Jezevčík
    - Objekt-Viditelný-Editovatelný-Polygon-Čtverec
  - Vztah interface-implementace
    - Readable-InputFile
    - Writable-OutputFile
    - (Readable+Writable)-IOFile

# Dědičnost

- IS-A hierarchie
  - C++: Jednoduchá nevirtuální veřejná dědičnost

```
class Derived : public Base
```

  - Abstraktní třídy někdy obsahují datové položky

- Vztah interface-implementace
  - C++: Násobná virtuální veřejná dědičnost

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

  - Abstraktní třídy obvykle neobsahují datové položky
  - Interface nebývají využívány k destrukci objektu

  - Oba přístupy se často kombinují

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

# Dědičnost

- Ideální použití dědičnosti je pouze toto
  - ISA hierarchie (typicky pro objekty s vlastní identitou)
  - Vztah interface-implementace
- Jiná použití dědičnosti obvykle signalizují chybu v návrhu
  - Výjimky samozřejmě existují (traits...)
- Speciálně, dědičnost není správným nástrojem pro reusabilitu kódu
  - Důvod nevhodnosti: automatické konverze potomek-předek

```
void f(container & k) { k.insert(/*...*/); }
class improved_container : public container { public:
  void insert(/*...*/) { container::insert(/*...*/); index_.insert(/*...*/); }
private: index_data index_;
};
void g( improved_container & ik) { f(ik);}
```

- Problém: Funkce f neaktualizuje index_
  - Kdyby funkce container::insert byla virtuální, bylo by to funkční řešení
    - Poznámka: Kvalifikované jméno container::insert vypíná virtuální volání
    - V C++ ale nechceme zbytečně platit za virtuální funkce ztrátou výkonnosti
  - Správné řešení je datová položka

```
class improved_container {
  void insert(/*...*/) { c_.insert(/*...*/); index_.insert(/*...*/); }
private: container c_; index_data index_;
};
void g( improved_container & ik) { f(ik);}
```

- Překladová chyba vynutí reimplementaci f pro improved_container - lze i genericky:

```
template< typename K> void f(K && k) { k.insert(/*...*/); }
```

# Nesprávné užití dědičnosti

- Nesprávné užití dědičnosti č. 1

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

- Porušuje pravidlo "každý potomek má všechny vlastnosti předka"
  - např. pro vlastnost "má nulovou imaginární složku"
- Důsledek - slicing:

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

Complex x;
double a = abs( x); // tento kód LZE přeložit, a to je špatně
```

- Důvod: Referenci na potomka lze přiřadit do reference na předka
  - Complex => Complex & => Real & => const Real &

# Nesprávné užití dědičnosti

- Nesprávné užití dědičnosti č. 2

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

  - Vypadá jako korektní specializace:
    "každé reálné číslo má všechny vlastnosti komplexního čísla"
  - Chyba: Objekty v C++ nejsou hodnoty v matematice
  - Třída Complex má vlastnost "lze do mne přiřadit Complex"
    - Tuto vlastnost třída Real logicky nemá mít, s touto dědičností ji mít bude

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

Real x;
set_to_i( x);   // tento kód LZE přeložit, a to je špatně
```

    - Důvod: Referenci na potomka lze přiřadit do reference na předka
    - Real => Real & => Complex &

# Virtuální funkce

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

- Mechanismus virtuálních funkcí se uplatní pouze v přítomnosti ukazatelů nebo referencí
  - Podstatou objektového programování v jakémkoliv jazyce je konverze odkazů (ukazatelů/referencí) ve směru potomek-předek

```
std::unique_ptr<Base> p = std::make_unique< Derived>();// konverze ukazatelů
p->f();   // volá Derived::f
```

- V jiné situaci není virtuálnost funkcí užitečná

```
Derived d;
d.f();    // volá Derived::f i kdyby nebyla virtuální

Base b = d;  // konverze způsobuje slicing = kopie části objektu
b.f();    // volá Base::f ikdyž je virtuální
```

  - Slicing je specifikum jazyka C++
    - Je nežádoucí, ale nelze jej z jazyka odstranit
    - Copy/move metody předka...

```
Base::Base(const Base &)
```

    - ...vždy akceptují reference na potomka

# Two worlds of (instantiated) 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);
```

- User-defined operators may be used

```
MyClass operator+(
  const MyClass & a, const MyClass & b);
k[0] = k[1] + k[2];
```

## 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

- User-defined operators useless

```
class Base { public:
  virtual std::unique_ptr<Base>
    operator+(const Base &b) const=0;
};
k[0] = *k[1] + *k[2]; // ???
```

# Class

# Class

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

- Třída v C++ je velmi silná univerzální konstrukce
    - Jiné jazyky většinou mívají několik slabších (class+interface)
  - Vyžaduje opatrnost a dodržování konvencí
- Tři stupně použití konstrukce class
  - Ne-instanciovaná třída = balík deklarací (pro generické programování)
  - Třída s datovými položkami a metodami (nejčastější použití v C++)
  - Třída s dědičností a virtuálními funkcemi (objektové programování)
- class = struct
  - struct: prvky implicitně veřejné
    - konvence: neinstanciované třídy a jednoduché třídy s daty
  - class: prvky implicitně privátní
    - konvence: složité třídy s daty a třídy s dědičností

# Tři stupně použití class

### Neinstanciovaná třída

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

### Třída nesoucí data

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

### Třídy s dědičností

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

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

# Typy, konstanty a statické položky ve třídách

```
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_;  // declaration of X::v_
};

X::t X::v_ = X::c;  // definition of X::v_

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

- Typové a statické položky...
  - nested class definitions
  - typedef definitions
  - static member constants
  - static member functions
  - static member variables
- ... nejsou vázány na žádnou instanci třídy (objekt)
- Ekvivalentní globálním typům a proměnným, ale
  - Používány s kvalifikovanými jmény (prefix X::)
  - Zapouzdření chrání proti kolizím
    - Ale namespace to dokáže lépe
  - Některé prvky mohou být privátní
  - Třída může být parametrem šablony

## 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 const t c = 1;
  static t f( t p)
  { return p + v; }
  static t v; // declaration of X::v
};
```

- Class must be defined in one piece
  - Definitions of class members may be placed outside

```
X::t X::v = X::c;// definition of X::v

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

- A class may become a template argument

```
using T = 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;
  const t c = 1;
  extern t v;  // declaration of X::v
};
```

- Namespace may be reopened
  - Namespace may be split into several header files

```
namespace X {
  inline t f( t p)
  { return p + v; }
  t v = c;     // definition of X::v
};
```

- Namespace members can be made directly visible
  - "using namespace"

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

# Namespaces and name lookup

```
namespace X {
  class N { /*...*/ };
  typedef unsigned long t;
  const t c = 1;
  extern t v;   // declaration of X::v
  inline t f( N p) { return p.m + v; }
};
```

- Namespace members can be made directly visible
  - "using", "using namespace"
- Functions in namespaces are visible by **argument-dependent lookup**
  - functions from a namespace may be visible even without "using"
  - "using" on functions does not hide previously visible versions

```
void f2()
{
  X::N a;
  auto b = f( a);   // calls X::f because the class type of a is a member of X
  using X::t;
  t b = 2;
  using namespace X;
  b = c;
}
```

# Classes as value types

# 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 (raw/smart) pointer

```
Y* r = new Y;
std::unique_ptr< Y> p =
  std::make_unique< Y>();
std::shared_ptr< Y> 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!

# 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;
std::unique_ptr<Y> 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(); // secondary
pointer
pp->set_m(0) // error
```

# 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)

# 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 ) { /* ... */ }
```

# 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

# Dědičnost a virtuální funkce

# Dědičnost

```
class Base { /* ... */ };
class Derived : public Base { /* ... */ }
```

- Třída Derived je odvozena z třídy Base
  - Obsahuje všechny datové položky i metody třídy Base
  - Může k nim doplnit další
    - Není vhodné novými zakrývat staré, vyjma virtuálních
  - Může změnit chování metod, které jsou v Base deklarovány jako virtuální

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

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

- final – zákaz dalších potomků
- override – test existence této virtuální metody v některém předku

# Názvosloví

- Abstraktní třída
  - Definice v C++: Třída obsahující alespoň jednu čistě virtuální funkci
    - Následkem toho nesmí být samostatně instanciována

```
class Base {
  //...
  virtual void f() = 0;
};
```

  - Běžná definice: Třída, která sama nebude instanciována
  - Představuje rozhraní, které mají z ní odvozené třídy (potomci) implementovat

- Konkrétní třída
  - Třída, určená k samostatné instanciaci
  - Implementuje rozhraní, předepsané abstraktní třídou, ze které je odvozena

# Dědičnost a destruktor

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

class Derived : public Base {
public:
  // ...
};
```

```
{
  Base * p = new Derived;
  // ...
  delete p;
}
{
  std::unique_ptr<Base> p =
  std::make_unique< Derived>();
   // ...
  // destruktor unique_ptr volá delete
}
```

- Pokud je objekt destruován operátorem delete aplikovaným na ukazatel na předka, musí být destruktor v tomto předku deklarován jako virtuální

- Odvozené pravidlo:
  - Každá abstraktní třída má mít virtuální destruktor
    - Je to zadarmo
    - Může se to hodit

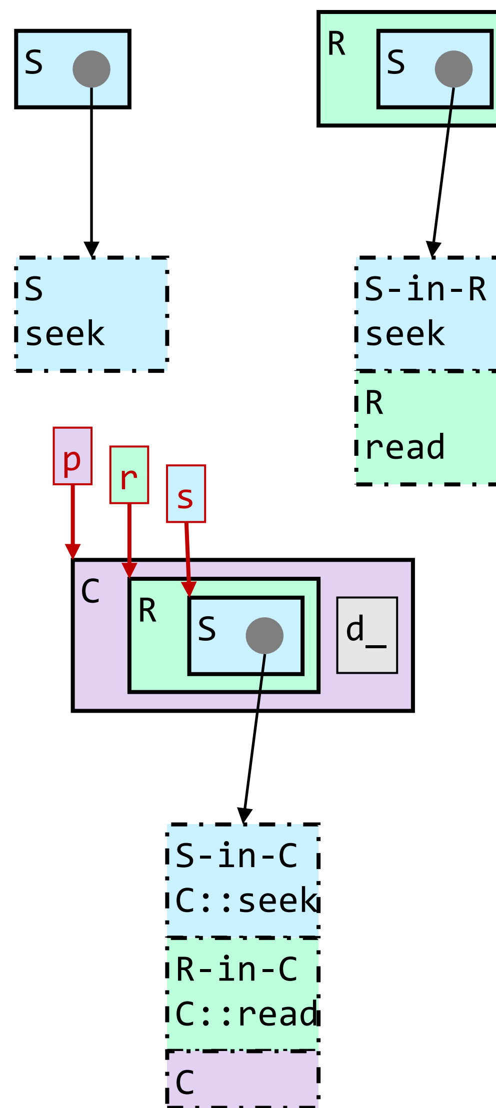
# Single non-virtual inheritance - example

```
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()
```



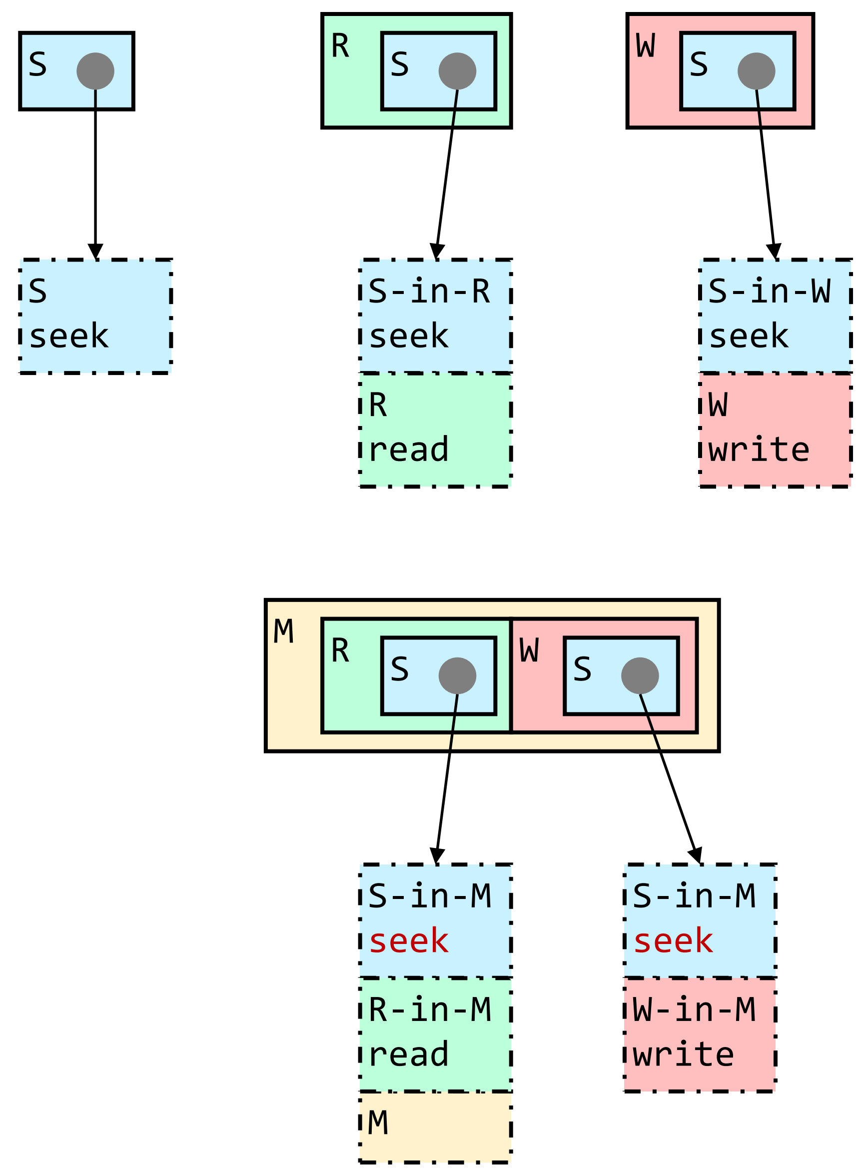
# Multiple non-virtual inheritance - example

```
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 &);
```