Programming Expertise
C++ Inheritance

Detlef Groth
2023-06-22

Last week

- functions
- call by value
- call by reference
- lambdas
- classes
- member variables
- member methods
- public, protected, private
- getter, setter functions
- constructors

1 C++ Destructors and Inheritance
Outline

C++ std::string

http://www.cplusplus.com/reference/string/string/

```
$ man std:: string # or
$ cppman string string

typedef basic_string<char> string;

String class
Strings are objects that represent sequences of characters.

The standard string class provides support for such objects
with an interface similar to that of a standard container of
bytes, but adding features specifically designed to operate
with strings of single-byte characters.
```

cppman in .bashrc

```
function cppman {
    if [ -z $1 ] ; then
        links http://www.cplusplus.com/reference/
    elif [ -z $2 ] ;  then
        links http://www.cplusplus.com/reference/$1/
    elif [ -z $3 ] ; then
        links http://www.cplusplus.com/reference/$1/$2
    else
        links http://www.cplusplus.com/reference/$1/$2/$3
    fi
}
```

std::string methods

Member functions

(constructor)	Construct string object (public member function)
(destructor)	String destructor (public member function)
operator=	String assignment (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return length of string (public member function)
length	Return length of string (public member function)
max_size	Return maximum size of string (public member function)
resize	Resize string (public member function)
capacity	Return size of allocated storage (public member function)
reserve	Request a change in capacity (public member function)
clear	Clear string (public member function)
empty	Test if string is empty (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Element access:

operator[]	Get character of string (public member function)
at	Get character in string (public member function)

C++11 regexp

```
#include <iostream>
#include <string>
// C++11 !!
#include <regex>
using namespace std;

int main (int argc, char **argv) {
  regex rxFullname("[A-Z][a-z]+ [A-Z][a-z]+");
  string name = "Detlef Groth";
  cout << "Enter you full name: " << flush;
  cin >> name;
  if (! regex_match (name, rxFullname)) {
   cout <<"\nError: Name not entered correctly, your name is '" <<
        name << "'??" << endl;
  } else {
```

```
    cout << name << endl <<
        "Great you entered the name correctly as: \n"
        << name << endl;
  }
  return 0;
}
```

$ g++ -o regexp.cpp.bin -std=c++17 -fconcepts regexp.cpp &&
./regexp.cpp.bin
Enter you full name:
Error: Name not entered correctly, your name is 'C++'??
⇒ http://www.cplusplus.com/reference/regex/
⇒ https://devdocs.io/cpp/regex/regex_replace
⇒ https://devdocs.io/cpp/regex/regex_search
⇒ https://devdocs.io/cpp/regex/regex_match

Destructor:

A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.
A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

[Tutorialspoint — The Class Destructor, 2019]

Use constructor for:

- cleanup like close db connection, close file, ...
- deal with static variables (class variables)

Static members:

We can define class members static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

[Tutorialspoint — Static Members of a C++ Class, 2019]

- must be declared inside of the class
- must be initialized outside of the class (not with C++17 ?)

$\Rightarrow$ static member variables are different to static variables of functions

```
#include <iostream>
using namespace std;
class Animal {
  public:
     // constructor 1
     Animal(int age);
     // constructor 2
     Animal();
     ~Animal(); // (painless) destructor
     int getAge();
     static int animalCount; // variable initialize outside!
     const static int MAXANIMALS = 10; // const must be inside
  protected:
    int itsAge = 0;
};
```

```
// static variable initialize outside
int Animal::animalCount = 0;
int Animal::getAge () {
    return(itsAge);
}
// no argument constructor
Animal::Animal () {
    animalCount += 1;
    cout << "simple animal creation ..." << endl;
}
// parameterized constructor
Animal::Animal (int age) {
    animalCount += 1;
    cout << "complex animal creation ..." << endl;
    itsAge=age;
}
```

```
// destructor definition (always with no arguments)
Animal::~Animal () {
    cout << "destroy an animal (painless!)" << endl;
    animalCount -= 1;
}
void func () {
    Animal animal ;
    cout << "inside func we have " <<
        Animal::animalCount << " animals " << endl;
}
int main () {
    Animal animal;
    cout <<"Animal 1's age: " <<animal.getAge() << endl;
    Animal animal2(12);
    cout <<"Animal 2's age: " <<animal2.getAge() << endl;
    func();
```

```
    cout << "inside main we have " <<
        Animal::animalCount << " animals " << endl;
    cout << "we leave main now ..." << endl;
    return(0);}
```

$ g++ -o static.cpp.bin -std=c++17 -fconcepts static.cpp &&
./static.cpp.bin

```
simple animal creation ...
Animal 1's age: 0
complex animal creation ...
Animal 2's age: 12
simple animal creation ...
inside func we have 3 animals
destroy an animal (painless!)
inside main we have 2 animals
we leave main now ...
destroy an animal (painless!)
destroy an animal (painless!)
```

Static member functions:

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator `::`.
A static member function can only access static data member, other static member functions and any other functions from outside the class.

[Tutorialspoint — Static Members of a C++ Class, 2019]

$\Rightarrow$ let's rewrite the last class using a static member function

$\Rightarrow$ we prefere methods over direct variable access

$\Rightarrow$ `Animal::animalCount` $\Rightarrow$ `Animal::getAnimalCount()`

```
#include <iostream>
using namespace std;
class Animal {
  public:
     Animal(int age);
     Animal();
     ~Animal();
     int getAge();
     // static method with access to static vars only
     static int getAnimalCount () {
        return animalCount ;
     }
  protected:
    int itsAge = 0;
    static int animalCount ;
};
```

```
// initialize outside;
int Animal::animalCount = 0;
int Animal::getAge () {
    return(itsAge);
}
// no argument constructor
Animal::Animal () {
    animalCount += 1;
    cout << "simple animal creation ..." << endl;
}
// parameterized constructor
Animal::Animal (int age) {
    animalCount += 1;
    cout << "complex animal creation ..." << endl;
    itsAge=age;
}
```

```
Animal::~Animal () {
    cout << "destroy an animal (painless!)" << endl;
    animalCount -= 1;
}
void func () {
    Animal animal ;
    cout << "inside func we have " <<
        Animal::getAnimalCount() << " animals " << endl;
}
int main () {
    Animal animal;
    cout <<"Animal 1's age: " <<animal.getAge() << endl;
    Animal animal2(12);
    cout <<"Animal 2's age: " <<animal2.getAge() << endl;
    func();
    cout << "inside main we have " <<
```

```
        Animal::getAnimalCount() << " animals " << endl;
}
```

$ g++ -o static2.cpp.bin -std=c++17 -fconcepts static2.cpp &&
./static2.cpp.bin

```
simple animal creation ...
Animal 1's age: 0
complex animal creation ...
Animal 2's age: 12
simple animal creation ...
inside func we have 3 animals
destroy an animal (painless!)
inside main we have 2 animals
destroy an animal (painless!)
destroy an animal (painless!)
```

Const member functions:

You have used the const keyword to declare variables that would not change. You can also use the const keyword with member functions within a class. If you declare a class method const, you are promising that the method won't change the value of any of the members of the class.

To declare a class method constant, put the keyword const after the parentheses enclosing any parameters, but before the semicolon ending the method declaration.

[Liberty and Bradley, 2005]

```
#include <iostream>
using namespace std;
class Animal {
  public:
     Animal(int age);
     Animal();
     ~Animal();
     int getAge() const; // no change on object
     void setAge (int age);
     static int animalCount ;
  protected:
    int itsAge = 0;
};
// initialize outside;
int Animal::animalCount = 0;
int Animal::getAge () const {
```

```
    return(itsAge);
}
void Animal::setAge (int age) {
    itsAge=age;
}
// no argument constructor
Animal::Animal () {
    animalCount += 1;
    cout << "simple animal creation ..." << endl;
}
// parameterized constructor
Animal::Animal (int age) {
    animalCount += 1;
    cout << "complex animal creation ..." << endl;
    itsAge=age;
}
```

```
Animal::~Animal () {
    cout << "destroy an animal (painless!)" << endl;
    animalCount -= 1;
}
int main () {
    Animal animal;
    cout <<"Animal 1's age: " <<animal.getAge() << endl;
    animal.setAge(animal.getAge()+1);
    cout <<"Animal 1's age: " <<animal.getAge() << endl;
    cout << "inside main we have " <<
        Animal::animalCount << " animals " << endl;
}
```

$ g++ -o constmeth.cpp.bin -std=c++17 -fconcepts constmeth.cpp && ./constmeth.cpp.bin

```
simple animal creation ...
Animal 1's age: 0
Animal 1's age: 1
```

```
inside main we have 1 animals
destroy an animal (painless!)
```

Inheritance:

In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. Also defined as deriving new classes (sub classes) from existing ones (super class or base class) and forming them into a hierarchy of classes. In most class-based object-oriented languages, an object created through inheritance (a "child object") acquires all the properties and behaviors of the parent object (except: constructors, destructor, overloaded operators and friend functions of the base class). Inheritance allows programmers to create classes that are built upon existing classes, to specify a new implementation while maintaining the same behaviors (realizing an interface), to reuse code and to independently extend original software via public classes and interfaces. ... Inheritance was invented in 1969 for Simula.

[Wikipedia — Inheritance (object-oriented programming), 2019]

Terminology

- Class: template for creating objects
- Class (static) variable: Variables shared by all objects of a class
- Class (static) method: functions for the class
- Object: a certain instance of a class
- Instance variable: a variable for each object
- Instance method: functions working for the objects
- Inheritance: methods & variables inherited from other classes
- Overriding: functions and operators can be overwritten by a child class to give them a new functionality
- Overloading: in languages like Java and Cpp classes can have method with the same name but with different arguments
- Component: An embedded but not inherited class/object inside a new class
- Delegation: concept of delegating methods to embedded components
- Mixins: other classes/objects can be embedded on the fly

Base Class and Derived Class

```
DerivedClass : public BaseClass {
    // derived class has the same interface as the
    // BaseClass, it can implement more methods.
};
DerivedClass : protected BaseClass {
    // class inheriting from DerivedClass has access to
    // all protected/public (now protected) methods of BaseClass
    // objects created from derived class have no!
    // access to methods and properties in BaseClass
};
DerivedClass : private BaseClass {
    // class inheriting from DerivedClass has access to
    // all protected/public (now private) methods of BaseClass
    // object created from derived class have no!
    // access to methods and properties in BaseClass
};
```

Class A, B, C, D examples

```
class A {
  public:
    int x;
  protected:
    int y;
  private:
    int z;
};
class B : public A {
  // x is public
  // y is protected
  // z is not accessible
};
```

```
class C : protected A {
  // x is protected
  // y is protected
  // z is not accessible
};
```

```
class D : private A {
  // 'private' is default
  // for classes
  // x is private
  // y is private
  // z is not accessible
};
```

Python and C++ (added in 2023)

R for package requirements in file DESCRIPTION:

- `import` (private inheritance of namespaces)
- `depends` (public inheritance of namespaces)
- S3 objects have only public inheritance
- R6 objects can have public and private methods

Python:

- `method` - public method or variable
- `_method` - protected method or variable
- `__` - private method or variable

Inheritance example

```
#include <iostream>

class Animal {
  public:
    void run () { std::cout << "Run, run, run!\n"; }
    void eat () { std::cout << "Eating generic food!\n"; }
};

class Cat : public Animal {
  public:
    void meow () {
        std::cout << "Meow, meow!\n";
    }
    // we have run automatically as we used: public Animal
};
```

```
// private inheritance is the default
// which is different to Python where public is the default
class Dog : private Animal {
  public:
    void wuff () {
        std::cout << "Wuff, wuff!\n";
    }
    // as it is private inheritance
    // we must reimplement run and call Animal::run()
    // by the Dog::run() method
    void run () {
        Animal::run();
    }
};
```

```
int main () {
    Cat mike = Cat();
    Dog fido = Dog();
    mike.run();
    mike.eat();
    mike.meow();
    fido.run(); // possible because we reimplemented it
    // fido.eat() not possible due to private inheritance :(
    fido.wuff();
}
```

```
$ g++   -o pubpriv.cpp.bin -std=c++17 -fconcepts pubpriv.cpp &&
./pubpriv.cpp.bin
Run, run, run!
Eating generic food!
Meow, meow!
Run, run, run!
Wuff, wuff!
```

In the beginning

...there was no inheritance and no composition, only code.
And the code was unwieldy, repetitive, blocky, unhappy, verbose, and tired.
Copy and Paste were the primary mechanisms of code reuse. Procedures and functions were rare, newfangled gadgets viewed with suspicion. Calling a procedure was expensive! Separating pieces of code from the main logic caused confusion!
It was a Dark Time.
Then the light of object-oriented programming (OOP) shone upon the world. And the world pretty much ignored it for a few decades. Until graphical user interfaces, which turned out to really, really need OOP. ...
After that, OOP took off. Numerous books have been written and countless articles have proliferated. So by now, everyone understands object-oriented programming in detail, right? Sadly, the code (and the Internet) says no.
The biggest point of confusion and contention seems to be composition versus inheritance, often summarized in the mantra favor composition over inheritance.
Let's talk about that.

[Steven Lowe: Composition vs. Inheritance: How to Choose?, 2015]

Inheritance and Composition

Composition:

In computer science, object composition is a way to combine objects or data types into more complex ones. Compositions relate to, but are not the same as, data structures, and common ones are the tagged union, set, sequence, and various graph structures, as well as the object used in object-oriented programming.

[Wikipedia — Object composition, 2019]

- inheritance - class B *is-a* Class-A. A dog `is-a` animal.
- composition - Class-B *has-a* Class-A. A dog `has-a` leg.

$\Rightarrow$ prefer composition over inheritance if possible

Components

- Inheritance: is_a $\Rightarrow$ Student is a Person
- Component: part_of $\Rightarrow$ Head is part of Student
- choose inheritance:
  - both classes are in the same logical domain
  - subclass is a proper subtype of superclass
  - superclass implementation is necessary or appropriate for the subclass
  - enhancements made by the subclass are primarily additive.
- choose composition when:
  - there is truly a part of relationship
  - you only need very few methods of the base class
  - you like to hide methods from the base class
  - you like to delegate methods to different components from an object

[Steven Lowe: Composition vs. Inheritance: How to Choose?, 2015]

Example for Composition (Python)

```
#!/usr/bin/python3
class Tail:
    def wag(self):
        print('Wag, wag!')
class Dog:
    def __init__(self,name,breed):
       self.tail=Tail() # component
       self.name=name
       self.breed=breed
       self.confidence=0

    def chase(self,thing):
        print(self.name,' chases ',thing)
        print('Wuff! Wuff!', end=' ')
        self.tail.wag()
        self.confidence=self.confidence+1
```

```
    def getConfidence(self):
        if self.confidence > 3:
            return('confidence of '+
             self.name+' is good!')
        else:
            return('confidence of '+
             self.name+' is ok!')

fido=Dog('Fido','Australian Shepherd')
for cat in ['Susi','Kathi', 'Moritz']:
    fido.chase('cat '+cat)
print(fido.getConfidence())
fido.chase('Student Martin')
print(fido.getConfidence())
print(fido.tail.wag()) # bad style
Fido  chases  cat Susi
Wuff! Wuff! Wag, wag!
Fido  chases  cat Kathi
```

```
Wuff! Wuff! Wag, wag!
Fido  chases  cat Moritz
Wuff! Wuff! Wag, wag!
confidence of Fido is ok!
Fido  chases  Student Martin
Wuff! Wuff! Wag, wag!
confidence of Fido is good!
Wag, wag!
None
```

Now the same in C++

```
#include <iostream>
#include <string>
class Tail {
  public:
    void wag () { std::cout << "Wag, wag!" << std::endl; }
};
class Dog {
  public:
    Dog(std::string name, std::string breed);
    int getConfidence () const { return(confidence); }
    void chase (std::string  thing);
  protected:
    int confidence = 0;
    std::string itsName ;
    std::string itsBreed;
    Tail itsTail;
```

```
};
Dog::Dog(std::string name, std::string breed) {
    itsName = name;
    itsBreed=breed;
    itsTail = Tail();
}
void Dog::chase (std::string thing) {
    std::cout << itsName << " chases "<<thing<< "!"<<std::endl;
    itsTail.wag();
    confidence += 1;
}
int main () {
  Dog fido("Fido PEX XXII","Australian Shepherd");
  fido.chase("Kathi PEX XXII");
  std::cout<<"Fido confidence: "<<fido.getConfidence()<<std::endl;
}
```

$ g++ -o composition.cpp.bin -std=c++17 -fconcepts composition.cpp &&

./composition.cpp.bin

```
Fido PEX XXII chases Kathi PEX XXII!
Wag, wag!
Fido confidence: 1
```

Multiple Inheritance

⇒ Avoid if possible: diamond problem!

⇒ Pegasus can't decide what to eat :(

```
#include <iostream>
using namespace std;
class Bird {
   public:
   void fly () {
        cout << "I believe I can fly ..." << endl;
   }
   void beat () { // can't have eat in Bird and Horse
       cout << "Mhmm, dendrobena ..." << endl;
   }

};
```

```
class Horse {
  public:
    void run () {
        cout << "Run, run, run, ..." << endl;
    }
    void heat () { // can't have eat in Bird and Horse
        cout << "Mhmm, apples ..." << endl;
    }

};
class Pegasus : public Bird, public Horse {
    public:
    void whoami () {
        cout << "Pegasus" << endl;
    }
};
```

```
int main () {
    Pegasus pegi ;
    pegi.fly();
    pegi.run();
    pegi.beat(); // can't have eat in Bird and Horse
    pegi.heat(); // can't have eat in Bird and Horse
    pegi.whoami();

}
```

```
$ g++  -o multi.cpp.bin -std=c++17 -fconcepts multi.cpp && ./multi.cpp.bin

I believe I can fly ...
Run, run, run, ...
Mhmm, dendrobena ...
Mhmm, apples ...
Pegasus
```

Multiple inheritance:

... is a feature of some object-oriented computer programming languages in which an object or class can inherit characteristics and features from more than one parent object or parent class. It is distinct from single inheritance, where an object or class may only inherit from one particular object or class.

Multiple inheritance has been a sensitive issue for many years, with opponents pointing to its increased complexity and ambiguity in situations such as the "diamond problem", where it may be ambiguous as to which parent class a particular feature is inherited from if more than one parent class implements said feature. This can be addressed in various ways, including using virtual inheritance. Alternate methods of object composition not based on inheritance such as mixins and traits have also been proposed to address the ambiguity.

[Wikipedia — Multiple inheritance, 2019]

Abstract class:

... is, conceptually, a class that cannot be instantiated and is usually implemented as a class that has one or more pure virtual (abstract) functions. A pure virtual function is one which must be overridden by any concrete (i.e., non-abstract) derived class. This is indicated in the declaration with the syntax "$= 0$" in the member function's declaration.

[Wikibooks — C++ Programming / Abstract Classes, 2019]

$\Rightarrow$ to avoid multiple inheritance (Java interfaces).

Example abstract class

```
class AbstractClass {
  public:
    // Pure virtual function makes
    // this class Abstract class.
    virtual void AbstractMemberFunction() = 0;
    // Virtual function.
    virtual void NonAbstractMemberFunction1();
    void NonAbstractMemberFunction2();
};
```

In general an abstract class is used to define an implementation and is intended to be inherited from by concrete classes. It's a way of forcing a contract between the class designer and the users of that class. If we wish to create a concrete class (a class that can be instantiated) from an abstract class we must declare and define a matching member function for each abstract member function of the base class.

[Wikibooks — C++ Programming / Abstract Classes, 2019]

Virtual Functions

```
#include <iostream>
class Animal {
  public:
     void eat() {
        std::cout << "I'm eating generic food. " ;
     }
};

class Cat : public Animal {
  public:
    void eat() { std::cout << "I'm eating a rat. "; }
};

void gomensa(Animal xyz) { xyz.eat(); }
```

```
int main () {
    Animal animal = Animal();
    Cat cat = Cat();
    animal.eat(); // Outputs: "I'm eating generic food."
    cat.eat();   // Outputs: "I'm eating a rat."
    std::cout << std::endl << "But in the mensa: " << std::endl;
    gomensa(animal);
    gomensa(cat); // Why not a rat?
}
```

```
$ g++  -o nonvirt.cpp.bin -std=c++17 -fconcepts nonvirt.cpp &&
./nonvirt.cpp.bin
I'm eating generic food. I'm eating a rat.
But in the mensa:
I'm eating generic food. I'm eating generic food.
```

Two Functions ...?

```
#include <iostream>
class Animal {
  public:
     void eat() {
        std::cout << "I'm eating generic food. " ;
     }
};
class Cat : public Animal {
  public:
    void eat() { std::cout << "I'm eating a rat. "; }
};

void gomensa(Animal xyz) { xyz.eat(); }
// we add another func for cats' but that is tedious ...
void gomensa(Cat xyz) { xyz.eat(); }
```

```
int main () {
    auto animal = Animal(); // or Animal animal = Animal();
    auto cat = Cat();
    animal.eat();
    cat.eat();
    std::cout << "Now in the mensa: " << std::endl;
    gomensa(animal);
    gomensa(cat);
}
```

$ g++ -o twofunc.cpp.bin -std=c++17 -fconcepts twofunc.cpp && ./twofunc.cpp.bin

```
I'm eating generic food. I'm eating a rat. Now in the mensa:
I'm eating generic food. I'm eating a rat.
```

Better we make Animal::eat virtual ...

```
#include <iostream>
class Animal {
  public:
     virtual void eat() {
        std::cout << "I'm eating generic food. " ;
     }
};

class Cat : public Animal {
  public:
    void eat() { std::cout << "I'm eating a rat. "; }
};

void gomensa(Animal xyz) { xyz.eat(); }
```

```
int main () {
    auto animal = Animal();
    auto cat = Cat();
    animal.eat();
    cat.eat();
    std::cout << std::endl <<"Now in the mensa: " << std::endl;
    gomensa(animal);
    gomensa(cat);
}
```

$ g++ -o virtfunc.cpp.bin -std=c++17 -fconcepts virtfunc.cpp && ./virtfunc.cpp.bin

```
I'm eating generic food. I'm eating a rat.
Now in the mensa:
I'm eating generic food. I'm eating generic food.
```

⇒ Not working yet ... we need pointers here ... Brrr ...

vritual Animal::eat needs pointers !

```
#include <iostream>
class Animal {
  public:
     virtual void eat() {
        std::cout << "I'm eating generic food. " ;
     }
};

class Cat : public Animal {
  public:
    void eat() { std::cout << "I'm eating a rat. "; }
};

void gomensa(Animal *xyz) { xyz->eat(); }
```

```
int main () {
    auto *animal  = new Animal();
    auto *cat = new Cat(); // pointer
    auto cat2 = Cat(); // no pointer
    animal->eat();
    cat->eat();
    cat2.eat();
    std::cout << std::endl <<"Now: " << std::endl;
    gomensa(animal);
    gomensa(cat);
    gomensa(&cat2); // by address pointer
}
```

$ g++ -o virtfunc.cpp.bin -std=c++17 -fconcepts virtfunc.cpp && ./virtfunc.cpp.bin

```
I'm eating generic food. I'm eating a rat. I'm eating a rat.
Now:
I'm eating generic food. I'm eating a rat. I'm eating a rat.
```

⇒ avoids declaring more `gomensa` functions

⇒ virtual functions are doing late binding ...

⇒ seems to work only with pointers

What about references?

```
#include <iostream>
class Animal {
  public:
     virtual void eat() {
        std::cout << "I'm eating generic food. " ;
     }
};

class Cat : public Animal {
  public:
    void eat() { std::cout << "I'm eating a rat. "; }
};

void gomensa(Animal &xyz) { xyz.eat(); }
```

```
int main () {
    auto animal = Animal();
    auto cat = Cat();
    animal.eat();
    cat.eat();
    std::cout << std::endl <<"Now: " << std::endl;
    gomensa(animal);
    gomensa(cat);
}
```

$ g++ -o virtref.cpp.bin -std=c++17 -fconcepts virtref.cpp && ./virtref.cpp.bin

```
I'm eating generic food. I'm eating a rat.
Now:
I'm eating generic food. I'm eating a rat.
```

⇒ References without virtual would not work :(

⇒ but with virtual keyword they did work!!

What about template?

```
#include <iostream>
class Animal {
  public:
     void eat() {
        std::cout << "I'm eating generic food. " ;
     }
};

class Cat : public Animal {
  public:
    void eat() { std::cout << "I'm eating a rat. "; }
};

template <typename T>
void gomensa(T xyz) { xyz.eat(); }
```

```
int main () {
    auto animal = Animal();
    auto cat = Cat();
    animal.eat();
    cat.eat();
    std::cout << std::endl <<"Now: " << std::endl;
    gomensa(animal);
    gomensa(cat);
}
```

```
$ g++   -o virtfunc.cpp.bin -std=c++17 -fconcepts virtfunc.cpp &&
./virtfunc.cpp.bin
I'm eating generic food. I'm eating a rat.
Now:
I'm eating generic food. I'm eating a rat.
```

⇒ Template does work ... so does auto?

What about auto?

```
#include <iostream>
class Animal {
  public:
     void eat() {
        std::cout << "I'm eating generic food. " ;
     }
};

class Cat : public Animal {
  public:
    void eat() { std::cout << "I'm eating a rat. "; }
};

void gomensa(auto xyz) { xyz.eat(); }
```

```
int main () {
    auto animal = Animal();
    auto cat = Cat();
    animal.eat();
    cat.eat();
    std::cout << std::endl <<"Now: " << std::endl;
    gomensa(animal);
    gomensa(cat);
}
```

```
$ g++   -o virtauto.cpp.bin -std=c++17 -fconcepts virtauto.cpp &&
./virtauto.cpp.bin
I'm eating generic food. I'm eating a rat.
Now:
I'm eating generic food. I'm eating a rat.
```

⇒ **Yes: auto works as well!**

(But only with actual compilers. No pointers needed.)

Solutions

- virtual with pointers: error prone syntax must change the base class syntax
- virtual with reference: clear syntax but must change as well the base class by adding the virtual keyword
- templates: base class can stay as it is, but template syntax
- auto: base class can stay as it is, but not obvious parameter, how should go mensa?
- your choice?

https://www.learncpp.com/cpp-tutorial/the-auto-keyword/

Starting in C++20, the auto keyword can be used as a shorthand way to create function templates, so the above code will compile and run. Note that this use of auto does not perform type inference.

Best practice
Avoid using type inference for function return types.

DG: Hmm, I found this add example quite useful ... simpler to understand than the template one.

Stylistic rules

- every class is in its own file of its own name
  `Animal` $\Rightarrow$ `Animal.cpp`
- filenames and header files should have different extensions
  (cxx, cpp, cc and hxx, hpp, hh)
- class names should start with uppercase letters: `Animal`
- object names should start with lowercase letters:
  `Animal fido("Fido")`
- for longer names use `CamelCase` or `Under_Scores`
- constants should be written in capital letters
- avoid `#defines` use `const`
- use namespaces, at least one for your project
- namespace should/could match the folder name

- avoid multiple inheritance
- you can have your own naming rules but be consistent
- use the new C++ features, at least C++11, best until C++17
- C++17 is ~~almost~~ completly implemented in all the major compilers (gcc, clang and Microsoft MSVC and Apple Clang)
- compile your code not in the source directory but in a directory like `build`
- add comments where neccessary
- document your code with a code documenting system like doxygen or our own mkdoc.py
- short license & author statement at the beginning of file
- See for more: `http://wiki.ros.org/CppStyleGuide`

Code documentation

- Doxygen
- Sphinx
- Natural Docs
- Robodoc
- GtkDoc
- ...

Purpose of code documentation:

- documentation should be used if you program with your classes
- you should not search in your source code to guess on how to use your code
- this is different to the purpose of code comments

mkdoc

- simple approach embedding Markdown into source code
  $\Rightarrow$ cross programming language
- Python script $\Rightarrow$ cross platform
- we used this already in the courses:
  - Databases and Practical Programming (Python)
  - Machine Learning in Bioinformatics (R)

```
cpp code
// comment
/*
comment
#' ## Markdown documentation
#'
#' **int add (int x, int y)**
*/
cpp code
```

mkdoc example

```
/*
#' ## Animal class
#'
#' Class to create animals which can eat.
#'
#' ## SYNOPSIS
#'
#' ```
#' #include "Animal.hh"
#' Animal animal("Fido",1);
#' animal.eat(1.5);
#' std::cout << animal.getWeight() << std::endl;
#' ```
#'
#' ## TABLE OF CONTENTS
#'
```

```
#' * [METHODS](#methods)
#' * [EXAMPLE](#example)
#' * [SEE ALSE](#seealso)
#' * [AUTHOR](#author)
#' * [LICENSE](#license)
#'
#' ## <a name='methods'>METHODS</a>
#' The class provides the following methods:
#'
#' * [Animal(int age)](#constructor)
#' * [eat(float amount)](#eat)
#' * [getAge()](#getAge)
#' * [getWeight()](#getAge)
#' * [run(float km)](#run)
*/
class Animal {
  public:
```

```
    // constructor 1
    Animal(string name, int age);
    int getAge();
    float getWeigh();
    void run (float km);
  protected:
    int itsAge = 0;
    float itsWeight = 0.0;
};
/*
(We indent here method and constructor descriptions
   for better readability)
#'
#' <a name="constructor">**Animal(int age)**</a>
#'
#' > Creates a new animal with given age.
#'
```

```
#' > Parameter:
#'
#' > * _int age_: the age in years
#'
#' > Returns: Object of class Animal
#'
#' > Example:
#'
#' > ```
#' Animal fido(12);
#' > ```
#'
*/
Animal::Animal (int age) {
    itsAge=age;
    std::cout << "simple animal creation ..." << std::endl;
}
```

```
#'
#' <a name="getAge">**object.getAge()**</a>
#'
#' > Returns the actual age for the animal.
#'
#' > Parameter: None
#'
#' > Returns: The actual age of the animal.
#'
#' > Example:
#'
#' > ```
#' Animal fido(12);
#' std::cout << fido.getAge() << std::endl;
#' > ```
#'
*/
```

```
int Animal::getAge () {
    return(itsAge);
}
/*
#' ## <a name="example">EXAMPLE</a>
#'
#' Here a example program which can be compiled.
#'
#' ```
#' #include <iostream>
#' #include "Animal.hh"
#'
#' int main () {
#'   Animal fido(1);
#'   fido.eat(1.0)
#'   fido.run(4)
#'   std::cout << animal.getAge() << std::endl;
```

```
#' }
#' ```
#'
#' ## <a name="seealso">SEE ALSO</a>
#'
#' ## <a name="author">AUTHORS</a>
#'
#' ## <a name="license">LICENSE</a>
*/
```

Most important formatting markups

```
# header 1
## header 2
### header 3
__bold__ _italic_ `code`
* list
** subblist
1. numlist
<a name="anchor">text</a>
[Linkname](#anchor)
> indent
```
code blocks
```
```

⇒ **Less is more, be clear and concise!**

Summary

- destructors (virtual destructors if virtual functions)
- static members variables and functions
- const member functions
- inheritance vs composition
- (multiple inheritance)
- (abstract classes)
- std::string
- templates
- auto as template replacement
- regex

Next week

- friend functions
- copy constructor
- this pointer
- operator overloading
- more on namespaces
- stack and heap
- more on templates
- STL (standard template library)

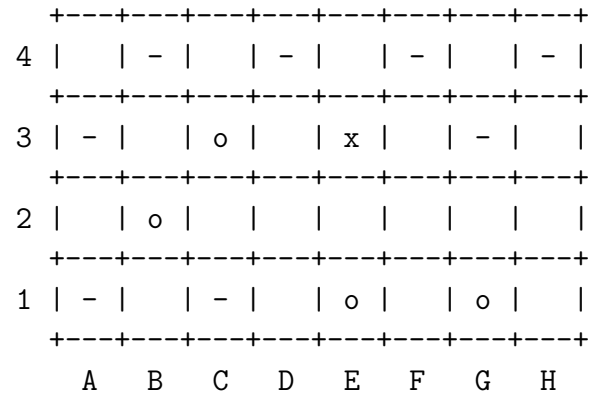
Exercise Extend C++ Classes

Exercise and Homework will be uploaded next week!
The homework will deal with command line arguments for our SheepBoard application. See next slide.

Homework

- implement wolf moves
  - the first preference of the wolf should be to move forward, you can choose randomly one of the two possibilities
  - if this is not possible move backward, but prefer more central positions, so avoid the border of the board where you can be easily trapped
  - may be you as well have an idea to check for a breakthrough possibility
- command line options
  - command line options are an important tool to control terminal and GUI programs
  - expand the command line facilities in the main function with long and short options
  - use for instance one of the following single file header libraries for extending the main function with command line parsing:

- `https://github.com/p-ranav/argparse`
- `https://github.com/mmahnic/argumentum`
- `https://github.com/hbristow/argparse`
- all these three are modeled similar like the Python library argparse `https://docs.python.org/3/library/argparse.html`
- the extended application should be finally callable like: `sheepboard -size 10` or `sheepboard -m` to invoke the game with a medium sized board 10x10.
- Other options could be -s for a small 8x8 board and -l for a 12x12 board. After startup your game menu should be started, asking the user for its moves.
- upload as well the header file
- you should include it like this: '#include p-ranav/argparse.hpp', so placed in a subfolder based n the Github user name



Here the wolf should favor positions where he can move to positions where there is a possible next forward step not blocked by a sheep, so here E3-D2 would be preferred as it wins immediately, E3-F2 would only win later ...

References

Tutorialspoint — The Class Destructor. Tutorialspoint, SIMPLYASLEARNINMG, 2019. URL https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm. [Online; accessed 7-June-2019].

Tutorialspoint — Static Members of a C++ Class. Tutorialspoint, SIMPLYASLEARNINMG, 2019. URL https://www.tutorialspoint.com/cplusplus/cpp_static_members.htm. [Online; accessed 7-June-2019].

J. Liberty and J. Bradley. SAMS - Teach yourself C++ in 21 days. Sams Publishing, 2005.

Wikipedia — Inheritance (object-oriented programming). Wikipedia, The Free Encyclopedia, 2019. URL https://en.wikipedia.org/wiki/

Inheritance_(object-oriented_programming). [Online; accessed 6-June-2019].

Steven Lowe: Composition vs. Inheritance: How to Choose? ThoughtWorks.com, 2015. URL https://www.thoughtworks.com/de/insights/blog/composition-vs-inheritance-how-choose. [Online; accessed 11-June-2019].

Wikipedia — Object composition. Wikipedia, The Free Encyclopedia, 2019. URL https://en.wikipedia.org/wiki/Object_composition. [Online; accessed 11-June-2019].

Wikipedia — Multiple inheritance. Wikipedia, The Free Encyclopedia, 2019. URL https://en.m.wikipedia.org/wiki/Multiple_inheritance. [Online; accessed 12-June-2019].

Wikibooks — C++ Programming / Abstract Classes. Wikibooks, 2019. URL

https://en.wikibooks.org/wiki/C%2B%2B_Programming/Classes/Abstract_Classes. [Online; accessed 11-June-2019].