# 1. Object-oriented programming

the concept of class, with member variables and functions that are used to access the class. An object-oriented C++ program can be entirely build on classes, and objects that represent instantions of a particular class, apart from the main() function that starts the program. Class is an abstract data type that hides the internal implementation of handling the data, and the other parts of the program only consider the public interface of the clas

s.

class Creature {

public:

Creature(const std::string& n, const std::string& t, int h) :

name(n), type(t), hitpoints(h) { }

const std::string& getName() const { return name; }

 The initialization list sets the initial value of member variable **name** to constructor argument '**n**', and so on. Then there are three functions to query the values of the member variables. Note the use of **const** in the function interface: for example getName() (line 10) returns a **const string reference**, i.e. the string cannot be modified through the return value.  On the other hand, const after the function name tells that the function call is not going to change the state (any of the variables) of the object. Because variable type is declared as **const string** (line 16) it cannot be modified after the object initialization.

class Troll : public Creature {

public:

Troll(const std::string& n) : Creature(n, "Troll", 10) { }

};

The first line says that Troll **inherits** from class Creature, i.e., it inherits the functions and member variables defined in base class. This calls the constructor of the base class with given parameters (for name, type and initial hit points). Otherwise the constructor does not include any functionality, as indicated by empty code block inside brackets.

we can use the Troll type object to access functions defined in Creature class because of the inheritance relationship. Likewise, if we create a vector for Creature type objects, we can add any types inherited from the Creature class to the vector.

# 2. Object relationships in C++

## 2.1 Name visibility between classes

Until now we have seen two catagories of variables and functions in classes: **public** variables or functions are visible to all other parts of the program that know about the class definition. **private** variables or functions are only available to functions inside the class.

A third category are **protected** variables (or functions) that are visible to the class itself, and all those classes that are inherited from it.

A class can declare another class to be a **friend**. Such other class has direct access to all protected or private members of the class declaring the friendship. This should be used sparingly and with careful consideration, because it bypasses the principles of information hiding.

A class can also declare an independent **function to be a friend**. Such function can access the "private parts" (no pun intended) of the class. As with friend classes, this mechanism should be used sparingly. A common example of this is when we add I/O stream support by overloading the << and >> operators to handle the class I/O properly. We will see an example of this later.

how class "SomeClass" is declared a friend of Creature, and how function "print()" is declared a friend.

friend class SomeClass;

friend void print(Creature& c);

## 2.2 Static members

Static class members are shared by all instances of the same class. For variables, this means that when a value of class member is altered by one instance, the change will be visible to others. For example, one could use a static variable to keep track on how many instances have been created of the given class. Functions that are marked as static cannot access the object-specific variables or parameters.

Because a static variable or function is not associated with an object, it can be called using the class name directly,

One common use of static functions is to create objects of the particular type.

## 2.3 Polymorphism

Classes that inherit the base class can have alternative implementations of same function. The actual class type then determines which of the functions is used. For example we could have function const std::string warCry() const; for the Creature type, and for Troll and Dragon. The version of warCry then depends on the class type we use to call it.

Polymorphism is one of the most powerful features in object oriented programming. C++ allows member functions of a base class to be declared **virtual**, which we should use with the warCry()function. Polymorphism means that the actual variant of the function is determined at runtime, depending on the actual type of the class. Even though the vector stores Creature members (or rather Creature references), the actual objects still can be Dragons or Trolls, and for virtual function the correct variant of the function is determined at runtime, depending on the actual object time.

By declaring the function **virtual**, the compiler adds necessary code to allow the implementation determine at runtime which is the correct version of the function to be called (this is called **dynamic binding**). This **works only for reference or pointer types**. Because reference types cannot be stored in containers, we have to use pointer type . It is good to note that the vector itself is not a pointer, and therefore the monsters functions are called using the dot notation. Only the elements inside the vector are pointers.

On the other hand, if the virtual specifier is not given for the warCry() function, there will be no runtime information needed for the dynamic binding of the function. Because the vector is defined for Creature pointers, also the function defined in Creature class is therefore used.

## 2.4 Pure virtual functions and abstract classes

A class member function can be declared **pure virtual**, meaning that there is no implementation for the function. **No objects can be created for a class that has pure virtual functions**, but then the class only works as base class for the actual classes that inherit from it, and provide an implementation for the (virtual) functions that are pure virtual in the base class. A class with pure virtual functions is therefore called an **abstract class**. It works as an interface to other classes, but cannot be used for creating objects.

virtual std::string warCry() const = 0;

# 3. Operator Overloading

The standard C (or C++) operators can be redefined through operator overloading. This can be useful with some classes, where e.g. the typical arithmetic operators have an intuitive meaning. It is not advised to do to this in counter-intuitive way, however, because it would only make the program very hard to understand.

he function for operator + overloading. Such operators can be defined in two ways: as member functions of the class, in which case binary operators have only one function parameter, and the other part of the binary operator being the object itself (as done with the + operator), or as global external functions in which case binary operators have two function parameters (as done with the \* operator).

friend GeomVector operator\*(double a, const GeomVector& b);

GeomVector GeomVector::operator+(const GeomVector& a)

This variant of overloading allows mixing different types on the both sides of a binary operator. Here we have a double multiplier for a GeomVector type. As with function overloading, we can have multiple versions of overloading the same operator for different types: we could add support for integer scalar multiplication, or multiplication of two GeomVectors, e.g., for dot product.

A particular useful mechanism is to overload output and input stream operators << and >>.

friend std::ostream &operator<<(std::ostream &out,

const GeomVector& a);

# 4. Dynamic Memory in C++

Even though the C++ standard library and its containers readily implement some of the basic cases of dynamic memory management such as dynamically resizing vectors or linked lists, oftentimes we need to allocate some memory by ourselves.

C++ has built-in mechanisms for allocating memory and releasing it. The language reserves keywords **new** for allocation, and **delete** for releasing the memory. malloc or free functions should never be used.

 In addition to allocating the used memory, new calls the constructor of a class. In addition to releasing the used memory, delete calls the destructor of a class.

Destructor is almost like any other function, except that it does not have return value, and it is called as the last action when the object going to be released. Destructor is called automatically, when an object is going to disappear, and the program should never (or very rarely) call it explicitly. Destructor should not do anything else, but clean up the resources used by the object,

Troll \*tr = new Troll("Peikko");

Because *Creature* is a virtual class, and the actual objects are either Trolls or Dragons, we need to create a virtual destructor also for Creatures, to make the compiler happy. When these objects are deleted, destructor for them is called. In the lack of explicit destructor, compiler will normally perform standard actions for object cleanup, but for virtual classes we need to specify the destructor ourselves. Trolls and Dragons may need specific destructors for cleaning up the class-specific things, although in this case there isn't anything that needs special action. Nevertheless, we will need the following addition to *Creature* class:

virtual ~Creature() { }

This shows that destructors can (and should) be virtual as any other functions implementing polymorfism, and in fact they must be virtual, if there are any other virtual functions in a class.

# 5. I/O in C++

cin is a readily created object of [**istream**](http://www.cplusplus.com/istream) type. istream is the class that implements input streams. cout is an object of [**ostream**](http://www.cplusplus.com/reference/ostream/ostream/) type, for output streams. In addition, there is class [**iostream**](http://www.cplusplus.com/iostream) that is for both input and output. This class inherits both istream and ostream (this is called [*multiple inheritance*](http://www.cprogramming.com/tutorial/multiple_inheritance.html), of which we haven't discussed here).

## 5.1 File streams

File streams are streams that are intended for reading or writing to file. The file streams are [**ifstream**](http://www.cplusplus.com/ifstream), [**ofstream**](http://www.cplusplus.com/ofstream) and [**fstream**](http://www.cplusplus.com/fstream), for input, output or both, respectively. These inherit from istream, ostream or iostream. The file stream classes are defined in a different header file, though: you should include **fstream** header in order to use them.

Because file streams inherit from basic I/O streams, they can use the same operations as the normal streams, including the stream operators << and >>. If you have overloaded these operators for your class, you can use them as usual also with files. They also work normally with the built-in data types and standard library types.

There is a close() function for file streams, as with traditional C streams, but also the destructor of file stream automatically closes the file. Therefore separately calling the close() function is not usually needed. By default, an ofstream object truncates (i.e., deletes content in) any existing file of the same name.

append new content at the end of the file:

std::ofstream os("outfile", std::ofstream::app)

## 5.2 Condition bits and manipulators

The current I/O stream status bits can be queried using the **[rdstate]** function. The function returns an object of type *iostate* (in the stream's namespace), that can be handled like a bit mask. The following values are in use:

* **badbit**: the stream is corrupted
* **failbit**: an I/O operation has failed
* **eofbit**: stream is at the end of file
* **goodbit**: stream is ok. This value is equivalent to zero (useful to know in condition tests).
* The output can be affected using **manipulators**, that are mixed together with the actual output using the <<operator. We have already seen one manipulator, **endl** that changes the line of output.
* **ends**: inserts a *null* character to the buffer
* **flush**: output all data currently in stream buffer. For example: std::cout << "Hey!" << std::flush;
* **unitbuf**: changes the buffering behavior to output each write directly. For example: std::cout << std::unitbuf;
* **nounitbuf**: reverts the buffering behavior back to normal system behavior.

## 5.3 String streams

String streams ([**stringstream**](http://www.cplusplus.com/reference/sstream/stringstream/)) are sometimes useful in when formatted output operations need to be applied to string object. When string stream is created, it can be used normally like any stream. All written content will be stored as a string, that can be used normally like any [string] object. String streams can also be used for input, and a string stream can be initialized with a predefined string. Stringstreams are defined in **sstream** header.