# 1. Generic programming using templates

By using **templates** we can specify generic classes or functions that can handle varying types. With templates, the compiler will determine and generate the needed type-specific versions of the classes or functions. We have already used templates with containers, where we have specified which type of objects the container stores by giving an appropriate template argument.

template <typename T>

T largest(T a, T b)

{

if (a > b)

return a;

else

return b;

}

We can implement **specialized templates** for certain types that need specialized handling.

template <> char largest(char a, char b)

{

if (toupper(a) > toupper(b))

return a;

else

return b;

}

# 2. More about dynamic memory

**Copy constructor** is a variant of constructor that is called when a new object is initialized as a direct copy of some other constructor, for example through assignment. In simple classes the compiler can implicitly construct such object just by doing a direct memory copy, but when class is using dynamic resources (files, dynamically allocated memory, etc,), we will need to implement our own copy constructor that allocates the resources properly also for the new copied class.

when we copy a dungeon, we need to allocate new instances of the Creatures as well, just copying the pointers is not enough.

Likewise, for classes that manage dynamic resources, we need to implement our own **assignment operator**(operator=). The assignment operation needs to **1) first properly release the resources owned by the to-be-overwritten object before the assignment**, and then properly **2) allocate the new resources needed for the copied class**, similarly to copy constructor.

This exaple presents an problem with abstract classes: because the list elements are pointers of Creature type, we cannot use new directly, because we would need to specify the actual type with it, which we do not know when writing the program. We solve this by adding a new virtual clone() function to all Creature types, that creates a dynamically allocated copy of the object. For example, the clone() implementation for Dragon looks like this:

virtual Creature\* clone() { return new Dragon(\*this); }

## 2.2 Rule of Three and Rule of Five

"[**Rule of Three**](https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming))" is was brought up in the 1990s to guide C++ programmers in handling classes with dynamic resources. It states that **whenever your object needs an explicit destructor** (because of dynamically allocated resources), it also needs **an explicit copy constructor** and **an explicit copy assignment** to work properly.

In C++11 there are additional **move semantics** for objects: 1) there is **move constructor** for moving an object from another reference. The move constructor should ensure that the original object does not have ownership for any of the dynamic resources in the object. 2) There is also **move assignment** for similar assignment operation. The move operations can often be implemented more efficiently than copy operations, by direct assignment of reference instead of deep copying.

When class has dynamic resources, e.g. it needs an explicit destructor, it then **also needs to implement the move constructor and move assignment**, when the C++11 move semantics are applied. Therefore, with C++11 the rule of three becomes the "[**Rule of Five**](https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming)#Rule_of_Five)". The [example](https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming)#Example_in_C.2B.2B) in the Wikipedia page shows how these are implemented.

# 3. Smart pointers and resource management

**Smart pointers** are very helpful tool in avoiding memory leaks with pointers. Smart pointer is a template class that wraps an actual pointer with logic that helps in proper handling of the pointer, particularly tracking when the memory should be released. We discuss three kinds of smart pointers: **shared pointer**, **unique pointer** and **weak pointer**, that differ on how the pointer ownership is defined. Smart pointers are defined in **memory** header.

## 3.1 Shared pointer

Shared pointer ([**shared\_ptr**](http://www.cplusplus.com/reference/memory/shared_ptr/?kw=shared_ptr)) can safely be stored in multiple places in the program. The shared\_ptr type holds a reference count that tracks the number of copies that the program currently has of the pointer. Shared pointer makes a good use of operator overloading: when pointer is assigned to another variable, the reference count is automatically increased. The shared pointer can also be used similarly to normal pointer, because it supports the \*and -> dereferencing operators. shared\_ptr is defined in **memory** header.

With shared pointer one should usually **avoid using the new operator for memory allocation**. Instead, new objects should be allocated using **make\_shared** function, that does the memory allocation and initialization, and initializes the reference counter properly. **make\_shared** returns a new shared pointer of given type.

With shared pointers we do not need to worry about releasing the memory: the class does it automatically when reference count drops to zero. After this using the traditional raw pointers feels quite untempting.

std::vector< std::shared\_ptr<Creature> > animals;

// Allocate new troll as a shared pointer, store in local variable

std::shared\_ptr<Troll> tr = std::make\_shared<Troll>("Peikko");

// add the troll to vector

animals.push\_back(tr);

// create new Dragon and add it directly to vector

animals.push\_back(std::make\_shared<Dragon>("Rhaegal"));

## 3.2 Unique pointer

Unique pointer ([**unique\_ptr**](http://www.cplusplus.com/reference/memory/unique_ptr/?kw=unique_ptr)) is a sole owner of the pointer it holds. The pointer cannot be shared, i.e., unique pointer does not implement copying or assignment of the pointer.

Unique pointer does not have any special function for allocating the pointer, but we just need to use the built-in newexpression to create the object

std::unique\_ptr<Creature> uptr(new Troll("Hemmo"));

The pointer can be "stolen" from *unique\_ptr* using the *release()* function. It returns raw pointer out of the unique pointer, and assigns null value to the original *unique\_ptr*. After this the programmer is again responsible of proper handling and releasing of the raw pointer. For example (adding to above example):

Creature \*cp = uptr.release();

C++ uses **nullptr** constant to indicate null pointer. It can be assigned also to a *unique\_ptr*. uptr = nullptr;would cause the memory allocated for Troll to be released. After this uptr can be set again using the *reset(somepointer)* function.

Unique pointer can be returned from a function, even though this technically means copying the pointer. Compiler is smart enough to see that the original *unique\_ptr* is going to be deleted at the same time we return from the function, so there is no risk of duplicate copies of the *unique\_ptr* type.

## 3.3 Weak pointer

Weak pointer ([**weak\_ptr**](http://www.cplusplus.com/reference/memory/weak_ptr/?kw=weak_ptr)) does not track the reference count of the pointer it owns. Otherwise it works quite similarly to shared pointer. A shared pointer (or another weak pointer) can be assigned to weak pointer, but this does not affect the reference count of the shared pointer.

Therefore it is possible that an object controlled by a shared pointer is deleted while there are weak pointers pointing to it. However, we can check it this has happened using the expired() function. If it returns true we should not try to use the weak pointer, because the object it points to has been deleted.

Weak pointer cannot be dereferenced directly because of the risk of handling an expired pointer. Instead, we must use the lock() function, that returns a new shared pointer based on the weak pointer, or nullptr if the weak pointer was expired. Weak pointer can only be initialized from a shared pointer.

## 3.4 Array new/delete

new and delete operations can also be applied to arrays of objects.

SomeClass \*ptr = new SomeClass[10];

Constructor is called for every item in the array.

Deletion of an array allocation is done in the following way:

delete[] ptr;

E.g., there is the array operator after delete, but the number of objects does not need to be given.