
6.11. Container Elements

Elements of containers must meet certain requirements because containers handle them in a special way. In this section, I describe these requirements and discuss the consequences of the fact that containers make copies of their elements internally.

6.11.1. Requirements for Container Elements

Containers, iterators, and algorithms of the STL are templates. Thus, they can process both predefined or user-defined types. However, because of the operations that are called, some requirements apply. The elements of STL containers must meet the following three fundamental requirements:

1. An element must be *copyable* or *movable*. Thus, an element type implicitly or explicitly has to provide a copy or move constructor.

   A generated copy should be equivalent to the source. This means that any test for equality returns that both are equal and that both source and copy behave the same.
2. An element must be *(move) assignable* by the assignment operator. Containers and algorithms use assignment operators to overwrite old elements with new elements.
3. An element must be *destroyable* by a destructor. Containers destroy their internal copies of elements when these elements are removed from the container. Thus, the destructor must not be private. Also, as usual in C++, a destructor must not throw; otherwise, all bets are off.

These three operations are generated implicitly for any class. Thus, a class meets the requirements automatically, provided that no special versions of these operations are defined and no special members disable the sanity of those operations.

Elements might also have to meet the following requirements:

- For some member functions of sequence containers, the *default constructor* must be available. For example, it is possible to create a nonempty container or increase the number of elements with no hint of the values those new elements should have. These elements are created without any arguments by calling the default constructor of their type.
- For several operations, the *test of equality* with operator `==` must be defined and is especially needed when elements are searched. For unordered containers, however, you can provide your own definition of equivalence if the elements do not support operator `==` (see Section 7.9.7, page 379).
- For associative containers, the operations of the *sorting criterion* must be provided by the elements. By default, this is the operator `<`, which is called by the `less<>` function object.
- For unordered containers, a *hash function* and an *equivalence criterion* must be provided for the elements. See Section 7.9.2, page 363, for details.

6.11.2. Value Semantics or Reference Semantics

Usually, all containers create internal copies of their elements and return copies of those elements. This means that container elements are equal but not identical to the objects you put into the container. If you modify objects as elements of the container, you modify a copy, not the original object.

Copying values means that the STL containers provide *value semantics*. The containers contain the values of the objects you insert rather than the objects themselves. In practice, however, you may also need *reference semantics*. This means that the containers contain references to the objects that are their elements.

The approach of the STL to support only value semantics has both strengths and weaknesses. Its strengths are that

- Copying elements is simple.
- References are error prone. You must ensure that references don't refer to objects that no longer exist. You also have to manage circular references, which might occur.

Its weaknesses are as follows:

- Copying elements might result in bad performance or may not even be possible.
- Managing the same object in several containers at the same time is not possible.

In practice, you need both approaches; you need copies that are independent of the original data (value semantics) and copies that still refer to the original data and get modified accordingly (reference semantics). Unfortunately, there is no support for reference semantics in the C++ standard library. However, you can implement reference semantics in terms of value semantics.

The obvious approach to implementing reference semantics is to use pointers as elements.[15] However, ordinary pointers have the usual problems. For example, objects to which they refer may no longer exist, and comparisons may not work as desired because pointers instead of the objects are compared. Thus, you should be very careful when you use ordinary pointers as container elements.

[15] C programmers might recognize the use of pointers to get reference semantics. In C, function arguments are able to get passed only by value, so you need pointers to enable a call-by-reference.

A better approach is to use a kind of *smart pointer*: objects that have a pointer-like interface but that do some additional checking or processing internally. Since TR1, in fact, the C++ standard library provides class `shared_ptr` for smart pointers that can share the same object (see Section 5.2.1, page 76). In addition, you could use class `std::reference_wrapper<>` (see Section 5.4.3, page 132) to let

STL containers hold references. Section 7.11, page 388, provides examples for both approaches.