---

### 6.8. Functions as Algorithm Arguments

To increase their flexibility and power, several algorithms allow the passing of user-defined auxiliary functions. These functions are called internally by the algorithms.

## 6.8.1. Using Functions as Algorithm Arguments

The simplest example is the `for_each()` algorithm, which calls a user-defined function for each element of the specified range. Consider the following example:

```
// stl/foreach1.cpp

#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

// function that prints the passed argument
void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    vector<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // print all elements
    for_each (coll.cbegin(), coll.cend(),     // range
              print);                         // operation
    cout << endl;
}
```

The `for_each()` algorithm calls the passed `print()` function for every element in the range [ `coll.cbegin()` , `coll.cend()` ). Thus, the output of the program is as follows:

```
1 2 3 4 5 6 7 8 9
```

Algorithms use auxiliary functions in several variants: some optional, some mandatory. In particular, you can use auxiliary functions to specify a search criterion or a sorting criterion or to define a manipulation while transferring elements from one collection to another.

Here is another example program:

Click here to view code image

```
// stl/transform1.cpp

#include <set>
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>
#include "print.hpp"

int square (int value)
{
    return value*value;
}

int main()
{
    std::set<int>    coll1;
    std::vector<int> coll2;

    // insert elements from 1 to 9 into coll1
```

```
    for (int i=1; i<=9; ++i) {
        coll1.insert(i);
    }
    PRINT_ELEMENTS(coll1,"initialized: ");

    // transform each element from coll1 to coll2
    // - square transformed values
    std::transform (coll1.cbegin(),coll1.cend(),    // source
                    std::back_inserter(coll2),      // destination
                    square);                        // operation

    PRINT_ELEMENTS(coll2,"squared:    ");
}
```

In this example, `square()` is used to square each element of `coll1` while it is transformed to `coll2` (Figure 6.11). The program has the following output:

```
initialized: 1 2 3 4 5 6 7 8 9
squared:     1 4 9 16 25 36 49 64 81
```

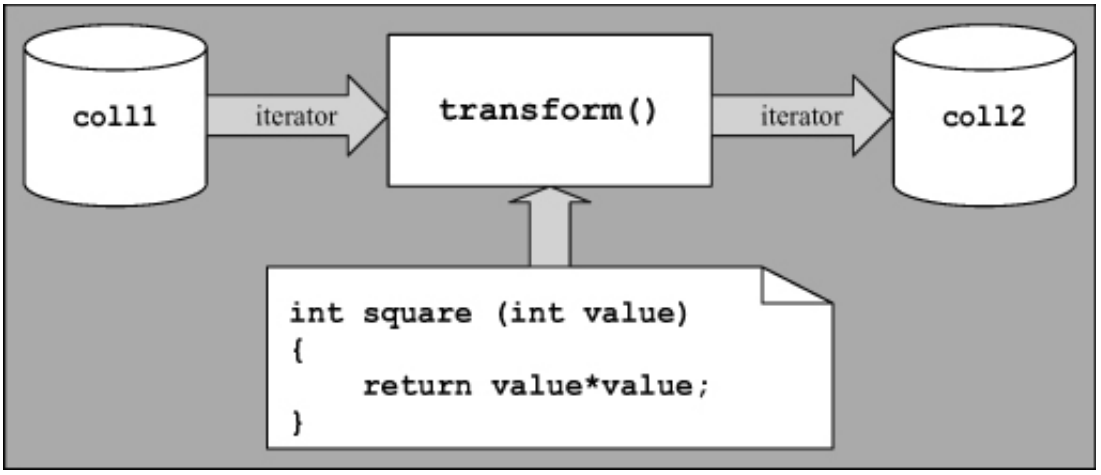


**Figure 6.11. How `transform()` Operates**

## 6.8.2. Predicates

A *predicate* is a special kind of auxiliary function. Predicates return a Boolean value and are often used to specify a sorting or a search criterion. Depending on their purpose, predicates are unary or binary.

Not every unary or binary function that returns a Boolean value is a valid predicate. In addition, the STL requires that predicates be stateless, meaning that they should always yield the same result for the same value. This rules out functions that modify their internal state when they are called. See Section 10.1.4, page 483, for details.

#### Unary Predicates

Unary predicates check a specific property of a single argument. A typical example is a function that is used as a search criterion to find the first prime number:

**Click here to view code image**

```
// stl/prime1.cpp

#include <list>
#include <algorithm>
#include <iostream>
#include <cstdlib>      // for abs()
using namespace std;

// predicate, which returns whether an integer is a prime number
bool isPrime (int number)
{
    // ignore negative sign
    number = abs(number);

    // 0 and 1 are no prime numbers
    if (number == 0 || number == 1) {
        return false;
    }

    // find divisor that divides without a remainder
    int divisor;
    for (divisor = number/2; number%divisor != 0; --divisor) {
```

```
        ;
    }

    // if no divisor greater than 1 is found, it is a prime number
    return divisor == 1;
}

int main()
{
    list<int> coll;

    // insert elements from 24 to 30
    for (int i=24; i<=30; ++i) {
        coll.push_back(i);
    }

    // search for prime number
    auto pos = find_if (coll.cbegin(), coll.cend(),     // range
                        isPrime);                       // predicate
    if (pos != coll.end()) {
        // found
        cout << *pos << " is first prime number found" << endl;
    }
    else {
        // not found
        cout << "no prime number found" << endl;
    }
}
```

In this example, the `find_if()` algorithm is used to search for the first element of the given range for which the passed unary predicate yields `true`. Here, the predicate is the `isPrime()` function, which checks whether a number is a prime number. By using this predicate, the algorithm returns the first prime number in the given range. If it does not find any element that matches the predicate, the algorithm returns the end of the range (its second argument). This is checked after the call. The collection in this example has a prime number between 24 and 30, so the output of the program is as follows:

```
29 is first prime number found
```

#### Binary Predicates

Binary predicates typically compare a specific property of two arguments. For example, to sort elements according to your own criterion, you could provide it as a simple predicate function. This might be necessary because the elements do not provide operator `<` or because you wish to use a different criterion.

The following example sorts elements of a deque by the first name and last name of a person:

```
// stl/sort1.cpp

#include <algorithm>
#include <deque>
#include <string>
#include <iostream>
using namespace std;

class Person {
  public:
    string firstname() const;
    string lastname() const;
    ...
};

// binary function predicate:
// - returns whether a person is less than another person
bool personSortCriterion (const Person& p1, const Person& p2)
{
    // a person is less than another person
    // - if the last name is less
    // - if the last name is equal and the first name is less
    return p1.lastname()<p2.lastname() ||
           (p1.lastname()==p2.lastname() &&
            p1.firstname()<p2.firstname());
}

int main()
{
    deque<Person> coll;
    ...

    sort(coll.begin(),coll.end(),  // range
         personSortCriterion);     // sort criterion
```

```
        ...
    }
```

Note that you can also implement a sorting criterion as a function object. This kind of implementation has the advantage that the criterion is a type, which you could use, for example, to declare sets that use this criterion for sorting their elements. See Section 10.1.1, page 476, for such an implementation of this sorting criterion.