Iterators

2

Summary: *Iterators are used by algorithms to move through containers. The simplest iterators are common pointers as shown in Section 1.3.4. This chapter describes different types of iterator and their properties in detail.*

A preliminary remark: iterators closely cooperate with containers. A parallel presentation of iterators and containers in a sequential text is however difficult and probably not very clear, and for this reason the containers of the STL are described only in the following chapter. In order to refer as far as possible only to previously explained issues, certain aspects of iterators which can only be understood with a knowledge of containers are temporarily left out. They will be considered at the end of Chapter 3.

Essential properties for all iterators are the capabilities mentioned on page 7 of advancing (++), of dereferencing (*), and of comparison (!= or ==). If the iterator is not a common pointer, but an object of an iterator class, these properties are implemented by means of the corresponding operator functions:

```
// scheme of a simple iterator:
template<class T>
class Iterator {
    public:
        // constructors, destructor ....

        bool operator==(const Iterator<T>&) const;
        bool operator!=(const Iterator<T>&) const;
        Iterator<T>& operator++();          // prefix
        Iterator<T>  operator++(int);       // postfix
        T& operator*() const;
        T* operator->() const;
    private:
        // association with the container ...
};
```

The operator -> allows you to use an iterator in the same way as a pointer. For a vector container, one could obviously imagine that the iterator should also have a

method `operator--()`. Different reasonable and possible capabilities of iterators are discussed further below.

The corresponding implementations of the lines beginning with the comment symbol (`//`) depend on the container with which the iterator is to work. The difference with a normal pointer has already been seen in Section 1.4 which shows an iterator working with a list. The iterator remembers the element of the list to which it points in a private pointer variable `current` (see page 12). Each element of the list contains `Data` and has a variable that points to the following element.

2.1 Iterator properties

2.1.1 States

Iterators are a generalization of pointers. They allow you to work with different containers in the same way. An iterator can assume several states.

- An iterator can be generated even without being associated with a container. The association with the container is then made at a later stage. Such an iterator cannot be dereferenced. A comparable C++ pointer could, for example, have the value 0.
- An iterator can be associated with a container during generation or at a later stage. Typically – but not compulsorily – after initialization it points to the beginning of the container. The method `begin()` of a container supplies the starting position. If the container is not empty, the iterator can in this case be dereferenced. Thus, it can be used to access an element of the container. With the exception of the `end()` position (see next point) the iterator can be dereferenced for all values that can be reached with the `++` operation.
- In C++ the value of a pointer which points to a position directly *past* the last element of a C array is always defined. Similarly, the method `end()` of a container always returns an iterator with exactly this meaning, even if the container is not an array but, for example, a list. This allows you to deal with iterator objects and pointers to C++ basic data types in the same way. A comparison of a current iterator with this past-the-end value signals whether the end of a container has been reached. Obviously, an iterator which points to the position past the end of a container cannot be dereferenced.

2.1.2 Standard iterator and traits classes

One essential advantage of templates is the evaluation of type names at compile time. To use type names that belong to iterators in a program without having to look into the internals of the iterator, it is specified that each iterator of the C++ Standard Library makes certain type names publicly available. The same principle also applies to containers. The `slist` class on page 12 provides such type names. Traits classes are a tool for exporting the type names of an iterator class:

```
template<class Iterator>
struct iterator_traits {
  typedef typename Iterator::difference_type
                                              difference_type;
  typedef typename Iterator::value_type value_type;
  typedef typename Iterator::pointer pointer;
  typedef typename Iterator::reference reference;
  typedef typename Iterator::iterator_category
                                           iterator_category;
};
```

The question arises as to why this task cannot be fulfilled directly by an iterator class itself. It can – in most cases. The algorithms of the C++ Standard Library should, however, be able to work not only on STL containers that provide type names, but also on simple C arrays. Iterators working on such arrays are, however, simply pointers, possibly to basic data types such as `int`. An iterator of type `int*` can certainly not provide any type names. To ensure that a generic algorithm can nevertheless use the usual type names, the above template is specialized for pointers:

```
// partial specialization (for pointers)
template<class T>
struct iterator_traits<T*> {
  typedef ptrdiff_t difference_type;
  typedef T value_type;
  typedef T* pointer;
  typedef T& reference;
  typedef random_access_iterator_tag iterator_category;
};
```

The iterator category is explained from page 33 onward. In order to make life easier for programmers, the C++ Standard Library specifies one standard data type for iterators from which each user-defined iterator can inherit:

```
namespace std {
  template<class Category, class T,
           class Distance = ptrdiff_t,
           class Pointer = T*,
           class Reference = T&>
  struct iterator {
    typedef Distance difference_type;
    typedef T value_type;
    typedef Pointer pointer;
    typedef Reference reference;
    typedef Category iterator_category; // see Section 2.1.4
  };
}
```

Via a `public` inheritance, these names are visible and usable in all derived classes.

2.1.3 Distances

In the examples on pages 6 ff, the required position in the array was determined by the difference of two pointers or iterators. In C++, the difference of a subtraction of pointers is represented by the data type `ptrdiff_t` which is defined by the header `<cstddef>`. However, the distance type may be different, dependent on the type of the iterator. For this purpose, the appropriate data type for the distance between two iterators can be chosen by the user. A standard function template `distance()` then determines the distance.

With the predefined iterator-traits templates it is possible to derive the type names needed, and the `distance()` function can be written as follows:

```
template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator First, InputIterator Second) {
      // calculation
}
```

The calculation for iterators that work with a vector consists only of a subtraction. If the container is a singly-linked list, the calculation will consist of a loop which counts the number of steps from the first iterator to the second.

The advantage of the traits templates is that only one type must be specified for the instantiation of the `distance()`-template. The return type is a distance type specified in the `iterator_traits` class. The `traits` classes allow definition of the data type names such as `difference_type` both for complex iterators and for basic data types such as `int*`.

How does this work in detail? The compiler reads the return type of `distance()` and instantiates the `iterator_traits` template with the corresponding iterator. Two cases must be distinguished:

- The iterator is of more complex nature, for example a list iterator. Then the sought type `iterator_traits<Iteratortype>::difference_type` is identical with `Iteratortype::difference_type`, as results from the evaluation of the instantiated `iterator_traits` template. In the case of the singly-linked list of page 12 this type results in `ptrdiff_t`.

- The iterator is a simple pointer, for example `int*`. For a pointer type, no names such as `difference_type` can be internally defined via `typedef`. The specialization of the `iterator_traits` template for pointers now ensures that *no* access is made to names of the iterator, because the required names can be found directly in the specialization without having to pass through an iterator. Then the sought type `iterator_traits<Iteratortype>::difference_type` is identical with `ptrdiff_t`, as results from the evaluation of the instantiated specialized `iterator_traits` template.

Thus, `distance()` can be described very generally, as shown above. Without the traits mechanism, there would have to be specializations for all the pointers, not only for pointers to basic data types, but also for pointers to class objects.

advance()

In order to advance an iterator by a given distance, the function `advance()` can be used:

```
template<class InputIterator_type, class Distance_type>
void advance(InputIterator_type& I, Distance_type N);
```

The iterator `I` is advanced by `N` steps. For iterators that can move forward and backward (bidirectional iterators) `N` may be negative.

2.1.4 Categories

The STL provides different iterators for the container in question. Each of these iterators can be assigned to one of the following five categories:

- input iterator
- output iterator
- forward iterator
- bidirectional iterator
- random access iterator

The categories correspond to the different capabilities of the iterators. For example, an iterator responsible for writing into a sequential file cannot move backward.

A special kind of iterator used for inserting elements into containers will be described in Section 3.5.

Input iterator

An input iterator is designed for reading a sequential stream of input data, that is, an `istream`. No write access to the object is possible. Thus, dereferencing does not supply an lvalue. The program fragment shows the principle of use:

```
// 'SourceIterator' is an input iterator
SourceIterator = Stream_container.begin();
while(SourceIterator != Stream_container.end()) {
    Value = *SourceIterator;
    // further calculations with Value ...
    ++SourceIterator;
}
```

Because of the stream property of the container associated with the input iterator, it is not possible to remember a special iterator value in order to retrieve an already read object at a later stage. Input iterators are suitable only for a *single* pass.

Output iterator

An output iterator is designed for writing not only into a container, but also into a sequential stream of output data (`ostream`). No read access to the object via dereferencing is possible. Dereferencing results in an lvalue which should exclusively be used on the left-hand side of an assignment.

```
// 'DestinationIterator' is an output iterator
*DestinationIterator = Value;
++DestinationIterator;          // advance
```

The two instructions are usually combined to

```
*DestinationIterator++ = Value;
```

If the output iterator works on a stream, advancing is already carried out by the assignment. Then, the ++ operation is an empty operation and exists only for reasons of syntactic uniformity (see also pages 43 and 63). Output iterators too are suitable for only *one* pass. Only one output iterator should be active on one container – thus we can do without comparison operations of two output iterators.

Forward iterator

As with the input iterator and the output iterator, the forward iterator moves forward. In contrast to the iterators mentioned above, the values of this iterator may be stored in order to retrieve an element of the container. This allows a multi-pass in one direction. A forward iterator would, for example, be suitable for a singly-linked list.

Bidirectional iterator

A bidirectional iterator can do everything that a forward iterator can do. In addition, it can move *backward*, so that it is suitable for a doubly-linked list, for example. A bidirectional iterator differs from a forward iterator by the additional methods `operator--()` (prefix) and `operator--(int)` (postfix).

Random access iterator

A random access iterator can do everything that a bidirectional iterator can do. In addition, it allows random access, as is needed for a vector. Random access is implemented via the index operator `operator[]()`. One consequence of this is the possibility of carrying out arithmetic operations, completely analogous to the pointer arithmetic of C++.

A further consequence is the determination of an order by means of the relational operators <, >, <=, and >=. In the following program, `Position` is a random access iterator associated with `Table`, a vector container. `n1` and `n2` are variables of type `Distance_type` (see page 32).

```
// Position is an iterator which points to a location somewhere inside Table
n1 = Position - Table.begin();
cout << Table[n1] << endl;     // is equivalent to:
cout << *Position  << endl;

if(n1 < n2)
   cout << Table[n1] <<  "lies before "
        << Table[n2] << endl;
```

In the simplest case, `Position` can be of type `int*`, and `n1` and `n2` of type `int`.

2.1.5 Reverse iterators

A reverse iterator is always possible with a bidirectional iterator. A reverse iterator moves *backward* through a container by way of the `++` operation. The start and end of a container for reverse iterators are marked with `rbegin()` (points to the last element) and `rend()` (fictitious position before the first element, an example follows on page 52). Some containers provide reverse iterators. These iterators are realized with the predefined class

```
template<class Iterator>
class reverse_iterator;
```

An object of this class is initialized with a bidirectional iterator or a random access iterator, depending on the type of the template parameter. Internally, a reverse iterator works with the initializing iterator and puts a wrapper with determined additional operations around it. A new interface is created for an existing iterator, so that it can adapt to different situations. For this reason, classes that transform one class into another are called *adaptors*. A bidirectional iterator can move backward with the `--` operation. This property is used to move from the end of a container to its beginning by means of a reverse bidirectional iterator using the `++` operation.

The iterator adaptor `reverse_iterator` also provides the member function `base()` which returns the current position as a bidirectional iterator. `base()` is needed to allow mixed calculations with normal and reverse iterators which work on the same container:

```
container C;          // any container type with public
                      // predefined types for iterators

typename container::iterator I = C.begin();      // start of C
// rbegin() points to the last element of C.
// rend() fictitious position before the first element.
typename container::reverse_iterator RI = C.rbegin();

// .... operations with the iterators, e.g. running backwards through it:
while(RI != C.rend()) {
  // ... do something with (*RI)
  ++RI;
}
```

```
// calculation of distance:
typename container::difference_type Distance =
         distance(RI, I);                    // incorrect
// compiler error message:
// RI and I are not of the same type

typename container::difference_type Distance =
         distance(RI.base(), I);             // correct
```

There are two kinds:

- Reverse bidirectional iterator
  This iterator can do everything that a bidirectional iterator can do. The only difference is the moving direction: the ++ operation of the reverse iterator has the same effect as the −− operation of the bidirectional iterators and vice versa.

- Reverse random access iterator
  This iterator can do everything the bidirectional reverse iterator described above can do. In addition, the arithmetic operations +, −, +=, and −= allow you to jump backward and forward several positions at a time in the container. In the above example, `distance()` uses the ++ operation; with a random access iterator, however, it uses arithmetic. Thus, you can write:

```
Distance = RI.base() - I;
```

The application of a reverse iterator is shown on page 52. Application of iterator categories in connection with containers and examples will be discussed only after the introduction of the different types of containers (Section 3.4).

2.1.6 Const iterators

The standard containers also provide iterators of the type `const_iterator` and `const_reverse_iterator`. These iterators are comparable to a pointer to const, e.g. `const char*`: they are not const but cannot be used to modify an element.

2.1.7 Tag classes

Each iterator of the STL is equipped with one of the following tags which can also be employed in the users' own programs. The tags are predefined as follows:

```
struct input_iterator_tag {};

struct output_iterator_tag {};
```

```
struct forward_iterator_tag
 : public input_iterator_tag {};
struct bidirectional_iterator_tag
 : public forward_iterator_tag {};
struct random_access_iterator_tag
 : public bidirectional_iterator_tag {};
```

2.2 Stream iterators

Stream iterators are used to work directly with input and output streams. The following sections show how stream iterators are employed for reading and writing sequential files. Stream iterators use the << and >> operators known from standard input and standard output.

2.2.1 Istream iterator

The istream iterator `istream_iterator<T>` is an input iterator and uses `operator>>()` for reading elements of type `T` with the well-known properties that 'white space,' that is spaces, tabs, and line feeds are ignored when in front of an element and are interpreted as separators when between two elements. Otherwise, all characters of the input stream are interpreted according to the required data type. Erroneous characters remain in the input and lead to endless loops, if no error treatment is incorporated.

During its construction and with each advance using ++, the istream iterator reads an element of type `T`. It is an input iterator with all the properties described in Section 2.1.4. At the end of a stream, the istream iterator becomes equal to the stream end iterator generated by the default constructor `istream_iterator <T>()`. A comparison with the stream end iterator is the only way of determining the end of a stream. The following very simple program reads all character strings separated by white space from a file (*istring.cpp* in the example) and outputs them line by line:

```
// k2/istring.cpp
#include<fstream>
#include<iostream>
#include<iterator>
#include<string>
using namespace std;

int main( ) {
    // defining and opening of input file
    ifstream Source("istring.cpp");
    istream_iterator<string> Pos(Source), End;
    /*The iterator End has no association with Source because all iterators of a type
      which indicate the past-end position are considered to be equal.
    */
```

```
    if(Pos == End)
       cout << "File not found!" << endl;
    else
       while(Pos != End) {
           cout << *Pos << endl;
           ++Pos;
       }
}
```

Character strings are represented by the standard data type `string`. At first sight, the basic data type `char*` might have been used as well, but there is a hitch to it: the iterator tries to read an object of type `char*`, but it is not possible to allocate memory to this object, and so the program will probably 'crash.'More complex types are possible, as will be shown in the next section. `End` is generated by the default constructor (with no arguments), and `Pos` is the iterator associated with the `Source` stream. The first read operation is already executed during construction with the `istream` argument, so that the subsequent dereferencing in the `while` loop always results in a defined value for the character string which is then written to the standard output.

tip

Structure of an istream iterator

It is possible to write an istream iterator with special properties which inherits from the `istream_iterator` class. An example can be found in Chapter 10. To show the methods usable by derived classes and the way of functioning as well, a possible implementation for an istream iterator is shown. The template parameter `char_traits` defines different types for different types of characters (`char` or wide characters), quite in analogy to the already known traits classes for iterators.

```
namespace std {
// possible implementation of an istream iterator

template<class T,
         class charT = char,
         class traits = char_traits<charT>,
         class Distance = ptrdiff_t>
class istream_iterator :
  public iterator < input_iterator_tag, T, Distance,
                    const T*, const T&> {
  public:
   typedef charT char_type;
   typedef traits traits_type;
   typedef basic_istream<charT,traits> istream_type;

   friend bool operator==(
       const istream_iterator<T, charT, traits, Distance>&,
       const istream_iterator<T, charT, traits, Distance>&);
```

```
  /*The constructor already reads the first element (if present). The private method
    read() (see below) uses the >>-operator.
  */
  istream_iterator(istream_type& s)
  : in_stream(&s)  {
     read();
  }

  // The default constructor generates an end-iterator
  istream_iterator() : in_stream(0) {}

  // copy constructor, assignment operator and destructor omitted!

   const T& operator*() const  { return value; }

   const T* operator->() const { return &(operator*()); }

   istream_iterator<T, charT, traits, Distance>&
   operator++() {
      read();
      return *this;
   }

   istream_iterator<T, charT, traits, Distance>
   operator++(int) {
      istream_iterator<T, charT, traits, Distance> tmp
         = *this;
      read();
      return tmp;
   }

 private:
   istream_type *in_stream;
   T value;

   /*If the stream is all right and not empty, an element is read with read(). The
     check (*in_stream) calls the type conversion operator void* of the class
     basic_ios to yield the stream state.
   */
   void read()  {
     if(in_stream) {              // stream defined?
       if(*in_stream)             // stream all right?
          *in_stream >> value;
       if(!(*in_stream))          // set undefined, if necessary
          in_stream = 0;
     }
   }
};
```

Two istream iterators are equal when both point to the same stream or to the end of a stream, as shown by the equality operator:

```
template<class T, class charT, class traits, class Distance>
bool operator==(const istream_iterator<T, charT, traits,
                  Distance>& x,
                const istream_iterator<T, charT, traits,
                  Distance>& y) {
    return x.in_stream == y.in_stream;
}

template<class T, class charT, class traits, class Distance>
bool operator!=(const istream_iterator<T, charT, traits,
                  Distance>& x,
                const istream_iterator<T, charT, traits,
                  Distance>& y) {
    return !operator==(x, y);
}

} // namespace std
```

2.2.2 Ostream iterator

The ostream iterator `ostream_iterator<T>` uses `operator<<()` for writing elements. This iterator writes at each assignment of an element of type `T`. It is an output iterator with all the properties described in Section 2.1.4.

Consecutive elements are normally written with `<<` directly into the stream, one after the other and without separators. Most often, this is undesirable because the result is often unreadable. To avoid this, the ostream iterator can at its construction be equipped with a character string of type `char*` which is inserted as a separator after each element. In the example on page 43, this is `\n` which is used to generate a line feed after each output.

In contrast to the example on page 37, the data type to be read and written is to be slightly more complex than `string`. Therefore, the task is now to read all *identifiers* from a file, according to the convention of a programming language, and to write them line by line into another file. Identifiers shall be defined as follows:

- An identifier always starts with a letter or an underscore '_'.
- Each following character occurring in an identifier is either alphanumeric (that is, a letter or a digit) or an underscore.

Thus, it is evident that an identifier cannot be read with the usual `>>` operator. Instead, we need an operator which considers these syntax rules and, for example, ignores special characters. Furthermore, an identifier must be able to contain a certain number of characters. This is guaranteed since the standard C++ string class is

used. An identifier should be able to be output with the usual << operator. With this information, we can already construct a simple class for identifiers:

```
// k2/identify/identif.h
#ifndef IDENTIF_H
#define IDENTIF_H
#include<iostream>
#include<string>

class Identifier {
  public:
   const std::string& toString() const { return theIdentifier;}
   friend std::istream& operator>>(std::istream&, Identifier&);
  private:
   std::string theIdentifier;
};
```

The method `toString()` allows you to generate a copy of the private variable which can be read and modified without affecting the original. The comparison operators are not really needed here but, on the other hand, containers are supposed to be comparable, which assumes that the elements of a container are comparable too. The comparison operators ensure that objects of the `Identifier` class can be stored in containers.

```
inline bool operator==(const Identifier& N1,
                       const Identifier& N2) {
    return N1.toString() == N2.toString();
}

inline bool operator<(const Identifier& N1,
                      const Identifier& N2) {
    return N1.toString() < N2.toString();
}

std::ostream& operator<<(std::ostream&, const Identifier&);
#endif
```

In order to find the beginning of an identifier, the implementation of the input operator in the file *identif.cpp* first searches for a letter or an underscore.

```
// k2/identify/identif.cpp
#include"identif.h"
#include<cctype>

std::istream& operator>>(std::istream& is, Identifier& N) {
    std::istream::sentry s(is);
    if(!s) return is;
```

```
    /*The constructor of the sentry-object carries out system dependent work. In particular, it checks the input stream so that in case of error, we can terminate the >> -operator immediately (see Kreft and Langer (2000)).
    */

    std::string IDstring;
    // find beginning of word
    char c = '\0';
    while(is && !(isalpha(c) || '_' == c))
          is.get(c);
    IDstring += c;

    /*When the beginning is found, all following underscores and alphanumeric characters are collected. 'White space' or a special character terminates the reading process.
    */

    // collect the rest
    while(is && (isalnum(c) || '_' == c)) {
       is.get(c);
       if(isalnum(c) || '_' == c)
         IDstring += c;
    }

    /*The last character read does not belong to the identifier. The iostream library offers the possibility of returning an unused character to the input so that it is available to a subsequent program.
    */
    is.putback(c);     // back into the input stream

    N.theIdentifier = IDstring;
    return is;
}
```

Implementation of the output operator is very easy; the internal `string` variable of an identifier is copied to the output `os`:

```
ostream& operator<<(ostream& os, const Identifier& N) {
    std::ostream::sentry s(os);
    if(s)
       os << N.toString();
    return os;
}
```

For `ostream::sentry s(os)` the same applies as for `istream::sentry s(is)` (see above). That is all that is needed to use stream iterators to recognize identifiers. The `main()` program which stores the list of identifiers in the file *idlist* uses the above Identifier class and is surprisingly short.

```
// k2/identify/main.cpp
#include<iterator>
```

```
#include<fstream>
#include"identif.h"

int main( ) {
    // defining and opening of input and output files
    std::ifstream Source("main.cpp");
    std::ofstream Target("idlist");

    std::istream_iterator<Identifier> iPos(Source), End;

    // please note the separator string '\n':
    std::ostream_iterator<Identifier> oPos(Target, "\n");

    if(iPos == End)
       std::cout << "File not found!" << std::endl;
    else
       while(iPos != End) *oPos++ = *iPos++;
}
```

The last line of the above program is only an abbreviated form of the following block:

```
{
    Identifier temp = *iPos;     // dereferencing
    ++iPos;                      // read new identifier
    *oPos = temp;                // write temp
    ++oPos;                      // do nothing
}
```

Looked at more closely, the `++` operation for the ostream iterator is superfluous, because it is already the assignment that calls `operator<<()`, thus triggering the write process. `++oPos` actually causes nothing. There is, however, a good reason why `operator++()` has been incorporated into the ostream iterator: the notation of the line

```
while(iPos != End) *oPos++ = *iPos++;
```

can thus be exactly as it is used with pointers to basic data types. This C++ idiom will be discussed again in Section 3.5.

Structure of an ostream iterator

It is even possible to write an ostream iterator with special features which inherits from the `ostream_iterator` class. To show the usable methods for derived classes and the way how they work, a possible implementation of the `ostream_iterator` is shown. The template-parameter `char_traits` defines different types for different kind of characters (`char` or wide characters).

```
namespace std {
 template<class T, class charT=char,
          class traits=char_traits<charT> >
```

```
class ostream_iterator :
 public iterator <output_iterator_tag, void, void,
                                       void, void> {

  public:
   typedef charT char_type;
   typedef traits traits_type;
   typedef basic_ostream<charT,traits> ostream_type;

   ostream_iterator(ostream_type& s)
   : out_stream(&s), delim(0) {
   }

   ostream_iterator(ostream_type& s,
                    const charT* separator)
   : out_stream(&s), delim(separator) {
   }

   // copy constructor and destructor omitted

   // assignment operator:
   ostream_iterator<T,charT,traits>&
   operator=(const T& value) {
      *out_stream << value;
      if(delim) {      // put out separator?
         *out_stream << delim;
      }
      return *this;
   }

   // operators only for idiomatic notation, for example = *iter++
   ostream_iterator<T,charT,traits>& operator*()  {
     return *this;
   }

   ostream_iterator<T,charT,traits>& operator++() {
     return *this;
   }

   ostream_iterator<T,charT,traits>& operator++(int) {
     return *this;
   }

  private:
   ostream_type* out_stream;
   const char_type* delim; // for separation of output elements
};
} // namespace std
```