Chapter 5

Errors

When we program, we have to deal with errors. Our most basic aim is correctness, but we must deal with incomplete problem specifications, incomplete programs, and our own errors. When we write programs, errors are natural and unavoidable; the question is, how do we deal with them? Organize the software fo minimize errors; eliminate most of the errors **debugging** and **testing**.

Some people say that finding, and correcting errors is 90% or more of the effort for serious software development! A program should produce the desired results for all legal inputs, should give reasonable error messages for illegal inputs, need not worry about misbehaving hardware, need not worry about misbehaving system software and finally, a program is allowed to terminate after finding an error.

As you program you have to handle with:

- **compile-time errors**: syntax and types are not correct;
- **link-time errors**: libraries not found or wrongly used;
- **run-time errors**: detected by computer (crash) or detected by library (exceptions) or detected by user code;
- **logic errors**: when code runs, but produces incorrect output. Compile-time errors are usually the easy to solve.

The sources of errors are generally **poor specification** ("*what this code is supposed to do?*"); **incomplete programs** ("*but I'll not get around to doing that until tomorrow?*"); **unexpected arguments** ("*but sqrt() isn't supposed to be called with -1 as its argument*"); **unexpected input**

("*but the user was supposed to input an integer*") and **code that simply doesn't do what it was supposed to do** ("*fix it!*").

Providing a set of rules for using C++ for a particular purpose in a particular environment (coding standards) also help to achieve correctness (see Sutter and Alexandrescu [1] "C++ Coding Standard" for example).

5.1 Run-time errors

If your program has no compile-time errors and no link-time errors, it will run. Usually, when you write a program you will are able to detect errors, but it is not so easy to know what to do with an error once you catch it at run time.

A very common source of run time errors are bad arguments in functions. A good practice is always check your arguments in a function unless you have a good reason not to (see Program 28)

Program 28 Argument checking

```
1  // let's the callee deal with errors
   int area(int length, int width)
3  {
   if (length<=0 || width<= 0)
5  return -1;
   return length*width;
7  }
```

Once you have chcked the arguments and found an error what should you do? Return a **error value**, or print the error, is an option . However, sometimes you don't have a value to represent the error, and you also need to thust that the callee will check the returned value.

Exceptions is a mechanism that helps to deal with run-time errors. The idea is to separate the detection of an error (which should be done in a called function) from a the handling of an error (which should be done in the calling function) while ensuring that a detected error can not be ignored. Program 30 shows an example of using exceptions.

In C++, exceptions are used to signal errors that cannot be handled locally, such as the failure to acquire a resource in a constructor.

[1] http://www.gotw.ca/publications/c++cs.htm

Program 29 Exception

```
class BadArea
{};

int area(int length, int width)
{
  if (length<=0 || width<= 0)
    throw BadArea();
  return length*width;
}

int main()
{
  try
  {
    int l, w;
    std::cout << "enter a length and a width: ";
    std::cin >> l >> w;
    std::cout << l << "x" << w << ": "<< area(l,w) << std::endl;
  }
  catch(BadArea)
  {
    std::cout << "Exception: bad arguments to calculate area!\n";
  }
  catch(...)
  {
    std::cout << "Something went wrong!\n";
  }
  return 0;
}
```

5.2 Logic errors

Once you have removed the initial compiler and linker errors, the program runs. What happens now it that the output is not produced or that the output that was produced is simply wrong. Let's illustrate a logic error with Program 30.

Can you spot the error(s)? This error is fairly typical; it doesn't cause compilation erros and can get write results for "reasonable" inputs. You have to think about what is reasonable and where do you get this from. Sometimes you have estimation, combine common sense and simple arithmetic applied to a few facts to test your program. Do not test only with the same input that always provide the right results.

Program 30 Exception

```
int main()
{
  vector<double> values;

  for (double val; cin>>val;)
    values.push_back(val);

  double sum=0;
  double high=0;
  double low=0;

  for (int x: values)
  {
    if (x>high) high=x;
      if (x<low) low=x;
    sum += x;
  }

  cout << "Highest: "<< high << endl;
  cout << "Lowest: " << low << endl;
  cout << "Average: "<< sum/values.size() << endl
}
```

5.3 Debugging

Debbuging is the process to find and remove errors (*bugs*) from a drafted program. Some practical advices that will help debugging:

- decide how to report errors: use `try-catch` exception in the `main` is my preferred start point;
- comment your code: say in comments, as clearly and briefly, what can't be said clearly in code:
  - name and purpose of the program
  - who wrote it and when
  - version numbers
  - what complicated code fragments are supposed to do
  - what the general ideas are
  - how the source code is organized
  - what assumptions are made about inputs
  - what parts of the core are still missing and cases are still not handled

- use meaningful names
- use consistent layout
- break code into small functions that express a logical actions
- avoid complicated code sequences
- use library facilities rather than your own code when you can
- insert statements that check invariants (`assert`)
- make sure you state pre and post-conditions form functions
- provide a systematic way of testing your code.

Chapter 6

Templates

Generic programming is writing code that works with a variety of types presented as arguments, as long as those argument types meet specific syntactic and semantic requirements. For example, the elements of a `vector` must be a type that we can copy. Basically, a template is a mechanism that allows a programmer to use types as parameters for a class or a function. The compiler then generates a specific class or function when we later provide specific types as arguments.

6.1 Function templates

When what we want parameterize is a function we get a *function template*, also called an *algorithm* (Program 31).

When you call a function template, the compiler tries to deduce the template type. Most of the time it can do that successfully, but every once in a while you may want to help the compiler deduce the right type –either because it cannot deduce the type at all, or perhaps because it would deduce the wrong type.

For example, you might be calling a function template that doesn't have any parameters of its template argument types, or you might want to force the compiler to do certain promotions on the arguments before selecting the correct function template. In these cases you'll need to explicitly tell the compiler which instantiation of the function template should be called.

Program 31 A function template

```
template<typename T>
void swap(T& x, T& y)
{
  T tmp = x;
  x = y;
  y = tmp;
}

int main()
{
  int         i,j;
  swap(i,j);  // Instantiates a swap for int
  float       a,b;
  swap(a,b);  // Instantiates a swap for float
  char        c,d;
  swap(c,d);  // Instantiates a swap for char
  std::string s,t;
  swap(s,t);  // Instantiates a swap for std::string
}
```

Program 32 Explicitly template instantiation

```
template<typename T>
void f()
{
  // ...
}
void sample()
{
  f<int>();           // type T will be int in this call
  f<std::string>();   // type T will be std::string in this call
}
```

6.2 Class template

When what we parameterize is a class, we get a *class template* (Program 33).

A class template by itself is not a type, or an object, or any other entity. No code is generated from a source file that contains only template definitions. In order for any code to appear, a template must be instantiated: the template arguments must be provided so that the compiler can generate an actual class (or function, from a function template).

Template classes can make use of another kind of template parameter known as an expression parameter. A template expression parameter is a parameter that does not substitute for a type, but is instead replaced by a value. An expression parameter can be any a value that has an integral type or enumeration; a pointer or reference to an object, a function or a class member.

Program 33 A class template

```
template <typename T>
class Array
{
  private:
  int length_;
  T* ptData_;

  public:
  Array():
    length_(0),
    ptData_(nullptr)
  {}

  Array(int nLength):
    length_(nLength),
    ptData_(new T[nLength])
  {}

  ~Array()
  { delete[] ptData_; }

  T& operator[](int nIndex)
  { return ptData_[nIndex]; }

  int GetLength();
};

template <typename T>
int Array<T>::GetLength() { return length_; }

int main()
{
  Array<int> anArray(12);
  Array<double> adArray(12);

  for (int i = 0; i < 12; ++i)
  {
    anArray[i] = i;
    adArray[i] = i + 0.5;
  }

  for (int i = 11; i >= 0; --i)
  std::cout << anArray[i] << "\t" << adArray[i] << std::endl;

  return 0;
}
```

6.3 The Standard Template Library - STL

The STL is part of the C++ standard library. It is a framework for dealing with data as sequences of elements. There are two major aspects of computing: the *computation* and *the data*. Any interesting computation is executed in lots os data (e.g. dozen of shapes, hundreds of temperature readings, thousands of log records, billions of web pages, etc.). Consider some simple examples of something we'd like to do with a *a lot of data*:

- sort the words in dictionary order
- sort the telemetry records by the time stamp
- find a number on a phone book given a name
- find the highest temperature
- find all values larger than 8800
- find the first occurrence of the values 17
- compute the sum of elements
- compute the number of occurrences of "lubia" in the dictionary
- compute the pair-wise product of the elements of two sequences.

Each of these tasks can be described without actually mentioning how the data is stored. The ideal is a templated library that helps with these and many other tasks offering:

- uniform access to data
  - independently of how it is stored
  - independently of its type
- type-safe access to data
- easy traversal of data
- compact storage of data
- fast
  - retrieval of data
  - addition of data

  - deletion of data

- standard versions of the most common algorithms

  - such as copy, find, search, sort, sum, ...

6.3.1 Iterators

The central concept of the STL is the *sequence*, or from the STL point of view, a collection of data. It has a *beginning* and an *end*. We can traverse a sequence from its beginning to its end, optionally reading and writing the value of each element. The beginning and the end of a sequence is defined by *iterators*. An iterator is an object that identifies and element of a sequence.

Figure 6.1: Sequence and iterators.

An STL sequence is "half-open"; that is, the element identified by `begin` is part of the sequence, but the `end` points one beyond the end of the sequence. The arrows from one element to the next indicate that if we have an iterator to one element we ca get an iterator to the next. It is a abstract notion:

- an iterator refers to an element of a sequence (or one beyond the last element)
- you can compare two iterators using `==` and `!=`
- you can refer to the valies of the element pointed to by an iterator using the unary `*` operator ("derereference" or "contents of")
- you can get an iterator to the next element by using `++`.

Iterators are used to connect our code to our data. The writer of the code knows about the iterators (not about the details of how iterators actually get the data). STL algorithms work very well together because they don't know anything about each other.

Program 34 shows a templated version of an algorithm that find the highest value in a sequence using the STL notion of a sequence.

Logically, STL iterators are organized in a hierarchy (Figure 6.3).

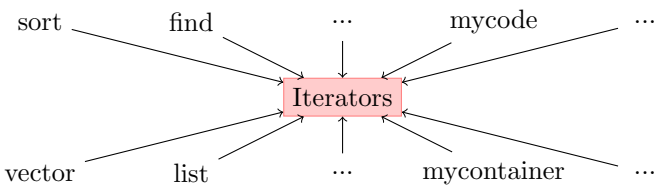

Figure 6.2: Iterators.

Program 34 Templated high algorithm

```
template<typename Iterator>
Iterator high(Iterator first, Iterator last)

{
  Iterator high = first;
  for (Iterator p=first; p!=last; ++p)
    if (*high < *p)
       high = p;
  return high;
}

int main()
{
  vector<int> v{5,7,19,0,10};
    vector<int>::iterator it = high(v.begin(), v.end());
    cout << "high: "<< *it << endl;

    double vv[5] {15,1,35,0,-1};
    double* h = high(&vv[0], &vv[0]+5);
    cout << "high 2: "<< *h << endl;

    return 0;
}
```

6.3.2 Containers

The STL contains a generic collection of class templates and algorithms that allow programmers to easily implement common data structures like queues, lists and stacks. There are three categories of containers – *sequence containers*, *associative containers*, and *unordered associative containers* – each of which is designed to support a different set of operations (see Table 6.1).

Sequence containers implement data structures which can be accessed sequentially. *Associative containers* implement sorted data structures that can be quickly searched ($O(log\ n)$ complexity. *Unordered associative containers* implement unsorted data structures that can be quickly searched ($O(1)$ amortized, $O(n)$ worst-case complexity)

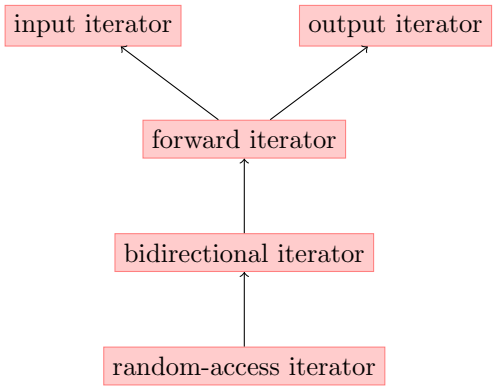

Figure 6.3: Iterators hierarchy.

Table 6.1: Containers

.5 **Sequence containers**	
`array`	static contiguous array
`vector`	dynamic contiguous array
`deque`	double-ended queue
`forward_list`	singly-linked list
`list`	doubly-linked list
Associative containers	
`set`	collection of unique keys, sorted by keys
`map`	collection of key-value pairs, sorted by keys, keys are unique
`multiset`	collection of keys, sorted by keys
`multimap`	collection of key-value pairs, sorted by keys
Unordered associative containers	
`unordered_set`	collection of unique keys, hashed by keys
`unordered_map`	collection of key-value pairs, hashed by keys, keys are unique
`unordered_multiset`	collection of keys, hashed by keys
`unordered_multimap`	collection of key-value pairs, hashed by keys

There are many member types, constructors, destructors, assignments, iterators and element access defined to each of the STL containers. A good summary can be seen in http://en.cppreference.com/w/cpp/container.

6.3.3 Algorithms

There are about 60 algorithms defined in the Algorithms library of STL, they are accessible in the header `<algorithm>`. They all operate on sequences defined by a pair of iterators for inputs or a single iterator for outputs. Some algorithms require random_access iterators (such as `sort`), whereas many, such as `find`, only read their elements in order so that they can make do with a forward iterator. Many algorithms follow the usual convention of returning the end of a sequence to represent "not found".

Program 35 shows an example is the `for_each` that applies a function to a range of elements, it is a *non-modifying sequence* operation. Program 37 shows an example is the `copy` that copies the elements in the range to another, this is a *modifying sequence* operation. Program 38 shows an example is the `sort` that sorts the elements in the range in ascending order.

Program 35 For each algorithm

```
int main()
{
  std::vector<int> nums{3, 4, 2, 8, 15, 267};

  std::cout << "before:";
  for (auto const &n : nums)
    std::cout << ' ' << n;
  std::cout << '\n';

  std::for_each(nums.begin(), nums.end(), [](int &n){ n++; });
}
```

Program 36 Copy algorithm

```
int main()
{
  std::vector<int> from_vector{3, 4, 2, 8, 15, 267};

  std::vector<int> to_vector;
  std::copy(from_vector.begin(), from_vector.end(),
            std::back_inserter(to_vector));
  std::cout << "to_vector contains: ";

  std::copy(to_vector.begin(), to_vector.end(),
              std::ostream_iterator<int>(std::cout, " "));
  std::cout << '\n';
  return 0;
}
```

Program 37 Copy algorithm

```
int main()
{
  std::vector<int> v = {0,1,2,3,4,5,6,7,8,9};
  std::cout << "Original vector:\n    ";
  for (int elem : v) std::cout << elem << ' ';

  auto it = std::partition(v.begin(), v.end(), [](int i){return i % 2
    == 0;});

  std::cout << "\nPartitioned vector:\n    ";
  std::copy(std::begin(v), it, std::ostream_iterator<int>(std::cout,
   " "));
  std::cout << " * ";
  std::copy(it, std::end(v), std::ostream_iterator<int>(std::cout, "
   "));
  return 0;
}
```

Program 38 Sort algorithm

```
int main()
{
  std::vector<int> s{5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

  // sort using the default operator<
  std::sort(s.begin(), s.end());
  for (int a : s)
    std::cout << a << " ";
  std::cout << '\n';
  return 0;
}
```

The library also includes binary search operations (on sorted ranges), set operations (on sorted ranges), heap operations, minimum/maximum operations and numeric operations. A good reference can be seen in http://en.cppreference.com/w/cpp/algorithm.