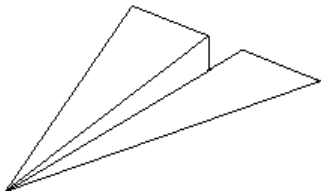
white paper



DEVELOPMENTOR[SM]

volume 3

ANSI C++

ABSTRACT

During the past several years, the C++ programming language has undergone a long and winding standardization process, the result of which is not only refinements and clarifications, but also a substantial number of new language features. Each of the new features was added to the language because a real-world problem had been identified that otherwise couldn't be solved elegantly and reliably. By using these new features, software engineers can improve the quality of their source code. Programs written in ANSI C++ can be more predictable, more readable, and more robust. An integral part of ANSI C++ is the standard library with its rich set of prefabricated abstractions. Using the library, a programmer can easily solve problems with a few C++ statements that in pre-standard C++ required deep thought and significantly more lines of code. The result is a visible reduction of complexity in ANSI C++ programs, which in turn makes programmers more productive and effective. In short, ANSI C++ increases productivity and quality, in addition to the classic benefits of a standardized language such as portability and prevalence.

MOTIVATION

Nowadays, hardly any C++ programmer can ignore ANSI C++. Modern compilers gradually manage to implement a growing percentage of the standardized language. The standard library is widely available and in numerous shops it has already replaced proprietary foundation libraries. Worldwide, engineers are exploring the new possibilities that the language and standard library now facilitates. As a result, new programming techniques emerge, are published and discussed, become common knowledge, and find their way into production code. As this trend continues, a thorough knowledge of the ANSI C++ features will be required of every C++ programmer.

Here is a list of the major new features available under ANSI C++:

• THE STANDARD LIBRARY

The standard library provides a rich set of efficient building blocks such as strings, containers, algorithms, complex numbers, streams, locales, and many more. There is neither a justification for reinventing the wheel again and implementing yet another string or linked list class, nor is there any need to resort to vendor-specific collection classes.

• TEMPLATES

Class and function templates add another dimension of programming power to the C++ language. Computation of constants, evaluation of expressions, and polymorphic dispatch of functions are all examples of tasks that – using templates – can be solved at compile time rather than at run time. I will never forget the committee meeting where a colleague of mine stepped into the plenary session with a program that did not even compile, but computed the prime numbers and emitted them in successive error messages. The program was stupid, but demonstrated impressively the power of compile-time computations. Even if you do not want to unleash this power, templates make life a lot easier. Whenever you find yourself tempted to solve a problem by "copy and paste," pause for a second and consider using a template instead. Let the compiler do the work instead of doing the tedious job yourself.

• EXCEPTION HANDLING

In pre-standard C++, it was cumbersome to indicate failure of constructors. The common technique of error indication via return codes does not work for functions such as constructors, destructors, or cast operators that do not have a return code, or for yet another category of functions whose return code is used for purposes other than error indication. First, it was kind of tedious to report errors from such functions and secondly, it was inevitable to put the burden of recognizing the error onto the user's shoulders. By and large, error handling in pre-standard C++ was unreliable and error-prone. The language feature of exceptions solves this problem and provides a uniform means for error indication and error handling.

• NAMESPACES

Name collisions happen more often than many engineers believe. Overloaded binary operators, for instance, must be implemented as global friend functions in order to make them behave symmetrically. The stream inserters and extractors are classical examples of such friend functions. If programmer A implements an inserter for objects of type X and programmer B implements yet another version of the same inserter, then we have a name clash that cannot be avoided, because operators have a name that we cannot change without losing the convenience of the operator notation. ANSI C++ introduced the concept of namespace to address this (and other) problems.

• THE NEW-STYLE CASTS

The classic C-style cast had a multitude of meanings and purposes. The new-style casts replace it completely and document more clearly the intent of a cast. Additionally, they enable the compiler to perform certain checks. For instance, in ANSI C++ we have a safe downcast that checks type information at run time.

• THE EXPLICIT KEYWORD

Unsolicited type conversions can happen if we provide one-argument constructors. These "converting constructors" are used by the compiler whenever it must set up a sequence of implicit type conversions. Often, the results are surprising. To prevent the compiler from silently using one-argument constructors from implicit conversions, the explicit keyword was introduced.

• THE MUTABLE KEYWORD

Sometimes you have to cast away the constness of an object. In ANSI C++, some of these cases can be eliminated by using the mutable keyword, which is for qualification of data members of a class, so that they can be modified from within a const member function without having to cast away the constness of the this pointer.

Each of these features has a potential to increase your productivity and the quality of your programs. That being said, we should still not forget that each of the new features adds to the complexity and power of an already complex and powerful programming language. Some of them, in particular templates, exceptions, and the library, have a learning curve of their own. There are gotchas and pitfalls and new challenges to master. For illustration, let us explore an example that uses the standard library.

PRE-STANDARD C++ VS. ANSI C++

In the following, we will discuss a simple program that reads lines from a file, sorts the lines, and writes the resulting sorted list to an output file. Before the advent of ANSI C++, we had to do quite a bit of work to make that happen. Below is a sample implementation of a read-sort-write program in classic C++. Don't attempt to understand the source code; just make a note of its size and complexity. In a minute, we will compare is to an ANSI C++ solution that is much more fun to look at. So, here is a pre-standard C++ solution:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>

void doIt(const char* in,const char* out)
{
   /* allocate buffer with initial capacity */
   size_t bufSiz =1024;
   char** buf = (char**) malloc(sizeof(char*)*bufSiz);
   if (buf == 0) quit();
   size_t linCnt = 0;
   buf[linCnt] = 0;

   /* allocate line buffer as destination for read */
   size_t linBufSiz = 256;
   char* linBuf = (char*) malloc(sizeof(char)*linBufSiz);
   if (linBuf == 0) quit();
   linBuf[0] ='\0';

   /* open input file */
   ifstream inFile(in);

   /*********************************************************/
   /*                          read input                   */
   while (!(inFile.getline(linBuf,linBufSiz)).eof() && !inFile.bad())
   {/* while there is still input */
      expandLinBuf(linBuf,linBufSiz,inFile);
      storeTok(buf,linCnt,bufSiz,linBuf);
   }
```

```
   /* sort strings */
   qsort(buf, linCnt ,sizeof(char*), (int(*)(const void*,const void*))strcmp);

   /* open output file and write sorted strings to output file */
   ofstream outFile(out);
   for (size_t i = 0; i<linCnt;i++)
      outFile<<buf[i]<<endl;
}
```

It needs a couple of helper functions, which are shown below:

```
static void quit()
{ cerr << “memory exhausted” << endl;
  exit(1);
}

static void expandLinBuf(char*& linBuf, size_t& linBufSiz,ifstream& inFile )
{
      while (!inFile.eof()&&!inFile.bad()&&strlen(linBuf)==linBufSiz-1)
      { /* while line does not fit into string buffer */

        /* reallocate line buffer */
        linBufSiz += linBufSiz;
        linBuf = (char*) realloc(linBuf,sizeof(char)*linBufSiz);
        if (linBuf == 0) quit();

        /* read more into buffer */
        inFile.getline(linBuf+linBufSiz/2-1,linBufSiz/2+1);
      }
}

static void storeTok(char**& buf, size_t& linCnt, size_t& bufSiz, const char* token)
{
         /* allocate memory for a copy of the token */
         size_t tokLen =strlen(token);
         buf[linCnt] = (char*) malloc(sizeof(char)*tokLen+1);
         if (buf[linCnt] == 0) quit();

         /* copy the token */
         strncpy(buf[linCnt++],token,tokLen+1);

         /* expand the buffer, if full */
         if (linCnt == bufSiz)
         {  bufSiz +=bufSiz;
            buf = (char**) realloc(buf,sizeof(char*)*bufSiz);
            if (buf == 0) quit();
         }
}
```

Quite a bit of code, isn't it? Basically, the program must provide and manage the memory for a line buffer into which the characters extracted from the file are stored. Plus it manages the memory for an array that holds all lines for subsequent invocation of the qsort() function. Both buffers must be of dynamic size, because neither the length nor the number of lines is known in advance. Reallocations might be necessary, which complicates matters even further.

In ANSI C++ the read-sort-write program boils down to something as concise and elegant as this:

```
#include <fstream>
#include <string>
#include <set>
#include <algorithm>
using namespace ::std;

void doIt(const char* in,const char* out)
  {
   set<string> buf;
   string linBuf;
   ifstream inFile(in);

   while(getline(inFile,linBuf))
         buf.insert(linBuf);

   ofstream outFile(out);
   copy(buf.begin(),buf.end(),ostream_iterator<string>(outFile,"\n"));
  }
```

Why is it such a piece of cake in ANSI C++ compared to the effort that it takes in classic C++? The answer lies in the use of abstractions such as string and set. They take over all the memory management chores that we had to do manually in the pre-standard version of the program. All the allocation and reallocation goo is handled by string and set; they manage their memory themselves and we don't have to care any longer. Plus, the set is an ordered collection of elements and we do not even have to sort it explicitly. Error indication is also much simpler. Situations such as memory exhaustion need not be indicated explicitly; instead the operator new, which is called somewhere in the innards of string and set, will raise a bad_alloc exception that is automatically propagated to the caller of our doIt() function. We need not do anything for error indication.

EXPLORING A BIT MORE OF THE STANDARD LIBRARY

If life in ANSI C++ is so easy, let's go and explore it a bit further. We often have to care about efficiency of our programs. Is the ANSI C++ solution above really as efficient as the classic C++ solution? Well, not really. In the ANSI solution we used a set container for storing the lines read from the input file whereas we used a plain C++ array of character C-style strings in the classic C++ solution. The set container is organized as a binary tree, and for that reason additional data must be maintained for linking the nodes together. Can we eliminate the resulting space overhead? Yes, we can. The standard library has a dynamic array container, called vector, that is less space consuming than the binary tree based set abstraction. Let us optimize the solution shown above and use vector<string> instead of set<string>. The vector's insert member function has a different signature. A set container is always ordered and therefore there is a "right" place for a new element that is inserted. This is different in a vector, and the insert() function asks for location where it shall insert the new element. Initially, the

vector is empty and there are not many choices for a location where new elements shall be inserted: we can either insert at the beginning or at the end, both being identical anyway. Say, we insert at the end. Then the re-implemented doIt() function looks like this:

```
void d(Itconst char* in,const char* out)
{
   vector<string> buf;
   vector<string>::iterator insAt = buf.end();
   string linBuf;
   ifstream inFile(in);

   while(getline(inFile,linBuf))
         buf.insert(insAt,linBuf);

   sort(buf.begin(),buf.end());

   ofstream outFile(out);
   copy(buf.begin(),buf.end(),ostream_iterator<string>(outFile,"\n"));
}
```

Looks good, doesn't it? It compiles, but — too bad L— at run time it crashes. Why? What is wrong here?

We need to look under the hood of the vector container if we want to understand what is happening here. How is vector organized and what precisely does the insert() function do? A vector internally is a contiguous memory space. Insertion into a vector means that all elements after the point of insertion are moved to the back, in order make room for the new element, and then the new element is added to the collection. A side effect is that all references to elements after the point of insertion become invalid. Now, the insert() function inserts the new element before the specified location. The point of insertion itself, in our example designated by the iterator insAt, becomes invalid as a side effect of the insertion. Any subsequent access to the element referred to by insAt might lead to a crash. This explains why our innocent program crashes after the first insertion of a line into the vector container.

There are several solutions to this problem. The insert() function returns an iterator to the newly inserted element and we can use this new, valid position as the point of insertion for subsequent additions to the vector. It would look like this:

```
void d(Itconst char* in,const char* out)
{
   vector<string> buf;
   vector<string>::iterator insAt = buf.end();
   string linBuf;
   ifstream inFile(in);

   while(getline(inFile,linBuf))
         insAt = buf.insert(insAt,linBuf);

   sort(buf.begin(),buf.end());

   ofstream outFile(out);
   copy(buf.begin(),buf.end(),ostream_iterator<string>(outFile,"\n"));
}
```

More elegant and easier to comprehend is the use of the push_back() function instead of the insert() function. It inserts elements at the end of a vector. Our example then looks like this:

```
void doIt(const char* in,const char* out)
{
   vector<string> buf;
   string linBuf;
   ifstream inFile(in);

   while(getline(inFile,linBuf))
         buf.push_back(linBuf);

   sort(buf.begin(),buf.end());

   ofstream outFile(out);
   copy(buf.begin(),buf.end(),ostream_iterator<string>(outFile,"\n"));
}
```

What do we conclude from the program crash that we inadvertently caused? To effectively use the standard library, and all the other new language features, we need to thoroughly understand them. They come with subtle pitfalls that we need to know, so that we can avoid them.

Before you get scared and think: "Well, the new stuff looks cool and will most likely save me lot of work, but it also lures me into lots of booby traps—is it really worth it?", let me tell you that we have barely touched on the possibilities that open up for you by using the standard library. Just as an example: How are the lines ordered in the code snippet above? We didn't care, so what happens? Basically, what happens is a strcmp() style comparison: the strings are ordered by comparing the ASCII codes of the contained characters. Where did we say so? Well, we did not. It is the default behavior of the sort() algorithm. If no compare function, in ANSI C++ more generally called a comparitor, is provided to the sort() function, then it uses the operator<() of the element type, which in our example is string. The ANSI string class has an operator<() defined and this operator performs an ASCII compare. The sort() algorithm implicitly uses it as the sorting criteria in the example above.

Equipped with this knowledge, we can consider other sorting orders. Ordering by ASCII codes does not meet the requirements of dictionary-like sorting, where upper case letters appear next to their lower case equivalents. In ASCII the capital letters precede all the lower case letters, so that capital 'Z' precedes lower case 'a'. Can we provide a dictionary-type ordering instead of the ASCII default? How about a case-sensitive ordering? How about culture-dependent sorting? Foreign alphabets include interesting special characters. How do they affect the sorting order? Lots of questions ....

As an example, let us consider a culture-sensitive sorting order. The standard library includes predefined abstractions for internationalization of programs. Among them is class locale, which provides culture-dependent string collation via its overloaded function call operator. An object of a class type that has the function call operator overloaded is called a functor in ANSI C++ and can be invoked like a function. In particular, we can pass it to the sort() algorithm as the sorting criteria.

Here is the respective code:

```
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include <locale>
using namespace ::std;

void doIt(const char* in,const char* out)
{
   vector<string> buf;
   string linBuf;
   ifstream inFile(in);

   while(getline(inFile,linBuf))
         buf.push_back(linBuf); // works

   sort(buf.begin(),buf.end(),locale("German"));

   ofstream outFile(out);
   copy(buf.begin(),buf.end(),ostream_iterator<string>(outFile,"\n"));
}
```

The culture-dependent sorting order just serves as an example here. We can define any other sorting criteria, as a function or as a functor, and plug it in with comparable ease. It works so nicely because the sort() algorithm is a function template that has the type of the comparitor as a template argument. This way you can use any type of comparitor for sorting. As you can see, the standard library makes your programs significantly more flexible and easy to extend.

CONCLUSIONS

In all but the stodgiest of work environments, software engineers currently make the transition from pre-standard C++ dialects and old-fashioned proprietary foundation libraries to ANSI C++ for two important reasons: portability and productivity. Every engineer knows from personal, painful experience how hard it is to write software that runs correctly and efficiently. So once a program works in one environment, we don't want to repeat the effort if we move to another (version of) compiler or processor or operating system. Portability is what standards are for, and ANSI C++ supports it in two ways: by eliminating language dialects and by providing a foundation library. In addition, it defines new language features that are designed to help us in pulling off our daily programming tasks more efficiently and with greater ease. If we intend to stay competitive, we cannot afford to ignore the new language standard. Nevertheless, while ANSI C++ has the potential to increase productivity, software quality, and portability, it also needs to be understood thoroughly if we want to get the most out of it.

Angelika Langer

white paper

DEVELOPMENTOR™
www.develop.com

What is new in ANSI C++?

What can you do with templates apart from parameterizing types?

What is generic programming, why and how would you want to use it?

How can you get the most out of the Standard Library?

How can you use stream I/O effectively?

How can you catch and cope with exceptions?

$1,795

FW320

essential ANSI C++

inside the new language features

ANSI C++: inside the new language features reveals the new language constructs in depth, and makes clear where and why you would want to use (or avoid) them in practice. You will learn how the ANSI C++ standardization has taken C++ beyond its object-oriented features to become a multi-paradigm programming language. Practical application of new C++ features will be demonstrated through lab exercises including changes to templates, which can now be used for generic programming, compile-time computations, static policy mix-ins, and much more. This course shows you why exceptions are now an integral part of the language, and why exception safety is an issue today. You will also learn about STL and the rich set of containers and algorithms it provides. The course has been designed by Angelika Langer, who is a member of the ANSI C++ Standards Committee and a columnist for C++ Report.

PREREQUISITES

You should have at least 1-2 years of C++ programming and be comfortable with the traditional object-oriented language features of C++, including classes and inheritance. This course assumes you have taken **Essential C++** or have equivalent knowledge, and have been using C++ in practice since.

COURSE HIGHLIGHTS

- CLASS AND FUNCTION TEMPLATES
- TEMPLATE SPECIALIZATION
- MEMBER TEMPLATES
- THROWING AND CATCHING EXCEPTIONS
- EXCEPTIONS IN CONSTRUCTORS/DESTRUCTORS
- AUTO POINTER AND STANDARD EXCEPTIONS
- EXCEPTION SPECIFICATIONS
- RUNTIME TYPE INFORMATION
- NEW-STYLE CASTS
- NAMESPACES
- CONTAINERS, ITERATORS, AND ALGORITHMS (STL)
- PROGRAMMING WITH FUNCTION OBJECT TYPES
- USER-DEFINED ITERATORS AND CONTAINERS
- STRINGS AND CHARACTER TRAITS
- STREAM INPUT/OUTPUT
- INPUT/OUTPUT OF USER-DEFINED TYPES

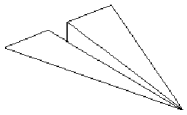
5 DAYS

COURSE INCLUDES STUDY-KIT, PC FOR LAB EXERCISES

Subject matter relative to ANSI C++

FOR MORE INFORMATION ON STANDARD C++ SEND E-MAIL TO
STANDARDCXX@DEVELOP.COM

registration



to register by phone, call
800-699-1932
from within the UK, call
08000-562-265
from within Europe, call
+44 1242 525 108

student information

STUDENT NAME

COMPANY NAME

ADDRESS

PHONE NO. MAIN LINE

DIRECT LINE

FAX NO.

E-MAIL ADDRESS

billing information

CONTACT NAME

COMPANY NAME

BILLING ADDRESS

PHONE NO. MAIN LINE

DIRECT LINE

FAX NO.

E-MAIL ADDRESS

NUMBER	COURSE DESCRIPTION	DATE	FEE	LOCATION
FW320	ANSI C++			

METHOD OF PAYMENT ☐ CHECK ☐ PURCHASE ORDER ☐ CREDIT CARD

PURCHASE ORDER NO.

CARD NO. EXP DATE

CARD TYPE ☐ VISA ☐ MASTERCARD ☐ AMERICAN EXPRESS

SIGNATURE

There is no cancellation fee if notification is received 15 days prior to the first day of instruction. Cancellation notices of 14 days or less are subject to 50% cancellation fee, applicable to the same course if rescheduled. Cancellations occurring without notice are responsible for the full tuition. DevelopMentor reserves the right to cancel a course if necessary, and in that event, all deposits will be promptly refunded or a new course date will be scheduled. Course content, prices, and availability are subject to change without prior notice.

Company purchase orders are also accepted per prior arrangement. Payment in full is required upon receipt of invoice. Two weeks prior to first day of instruction, only paid reservations can be held as confirmed.
For additional information, email info@develop.com

DEVELOPMENTOR℠

PLEASE MAIL OR FAX THIS REGISTRATION FORM TO:

21535 Hawthorne Blvd
Fourth Floor
Torrance, CA 90503
(310) 543 1716
fax (310) 543 2136
www.develop.com℠

Subject matter relative to ANSI C++

FOR MORE INFORMATION ON STANDARD C++ SEND E-MAIL TO **STANDARDCXX@DEVELOP.COM**