Learning Standard C++ as a New Language

Bjarne Stroustrup

AT&T Labs

ABSTRACT

To get the most out of Standard C++ [C++,1998], we must rethink the way we write C++ programs. An approach to such a "rethink" is to consider how C++ can be learned (and taught). What design and programming techniques do we want to emphasize? What subsets of the language do we want to learn first? What subsets of the language do we want to emphasize in real code?

This paper compares a few examples of simple C++ programs written in a modern style using the standard library to traditional C-style solutions. It argues briefly that lessons from these simple examples are relevant to large programs. More generally, it argues for a use of C++ as a higher-level language that relies on abstraction to provide elegance without loss of efficiency compared to lower-level styles.

1 Introduction

We want our programs to be easy to write, correct, maintainable, and acceptably efficient. It follows that we ought to use C++ – and any other programming language – in ways that most closely approximate this ideal. It is my conjecture that C++ community has yet to internalize the facilities offered by Standard C++ so that major improvements relative to the ideal can be obtained from reconsidering our style of C++ use. This paper focuses on the styles of programming that the facilities offered by Standard C++ support – not the facilities themselves.

The key to major improvements is a reduction of the size and complexity of the code we write through the use of libraries. Below, I demonstrate and quantify these reductions for a couple of simple examples such as might be part of a introductory C++ course.

By reducing size and complexity, we reduce development time, ease maintenance, and decrease the cost of testing. Importantly, we also simplify the task of learning C++. For toy programs and for students who program only to get a good grade in a nonessential course, this simplification would be sufficient. However, for professional programmers efficiency is a major issue. Only if efficiency isn't sacrificed can we expect our programming styles to scale to be usable in systems dealing with the data volumes and real-time requirements regularly encountered by modern services and businesses. Consequently, I present measurements that demonstrate that the reduction in complexity can be obtained without loss of efficiency.

Finally, I discuss the implications of this view on approaches to learning and teaching C++

2 Complexity

Consider a fairly typical second exercise in using a programming language:

```
write a prompt "Please enter your first name"
read the name
write out "Hello <name>"
```

In Standard C++, the obvious solution is:

```
#include<iostream>    // get standard I/O facilities
#include<string>      // get standard string facilities
```

```
int main()
{
    using namespace std;       // gain access to standard library

    cout << "Please enter your first name:\n";
    string name;
    cin >> name;
    cout << "Hello " << name << '\n';
}
```

For a real novice, we need to explain the ''scaffolding:'' What is ***main()***? What does ***#include*** mean? What does ***using*** do? In addition, we need to understand all the ''small'' conventions, such as what ***\n*** does, where semicolons are needed, etc.

However, the main part of the program is conceptually simple and differs only notationally from the problem statement. We have to learn the notation, but doing so is relatively simple: ***string*** is a string, ***cout*** is output, **<<** is the operator we use write to output, etc.

To compare, consider a traditional C-style solution†:

```
#include<stdio.h>     // get standard I/O facilities

int main()
{
    const int max = 20;        // maximum name length is 19 characters
    char name[max];

    printf("Please enter your first name:\n");
    scanf("%s", name);         // read characters into name
    printf("Hello %s\n", name);

    return 0;
}
```

Objectively, the main logic here is slightly – but only slightly – more complicated than the C++-style version because we have to explain about arrays and the magic ***%s***. The main problem is that this simple C-style solution is shoddy. If someone enters a ''first name'' that is longer than the magic number 19 (the stated number 20 minus one for a C-style string terminating zero), the program is corrupted.

It can be argued that this kind of shoddiness is harmless as long as a proper solution is presented ''later on.'' However, that line of argument is at best ''acceptable'' rather than ''good.'' Ideally, a novice user isn't presented with a program that brittle.

What would a C-style program that behaved as reasonably as the C++-style one look like? As a first attempt we could simply prevent the array overflow by using ***scanf()*** in a more appropriate manner:

```
#include<stdio.h>     // get standard I/O facilities

int main()
{
    const int max = 20;
    char name[max];

    printf("Please enter your first name:\n");
    scanf("%19s", name);       // read at most 19 characters into name
    printf("Hello %s\n", name);

    return 0;
}
```

There is no standard way of directly using the symbolic form of the buffer size, ***max***, in the ***scanf()*** format string, so I had to use the integer literal. That is bad style and a maintenance hazard. The expert-level alternative is not one I'd care to explain to novices:

---

† For aesthetic reasons, I use C++ style symbolic constants and C++ style //-comments. To get strictly conforming ISO C programs, use ***#define*** and /* */ comments.

```
char fmt[10];
sprintf(fmt,"%%%ds",max-1);  // create a format string: plain %s can overflow
scanf(fmt,name);             // read at most max-1 characters into name
```

Furthermore, this program throws ‘‘surplus’’ characters away. What we want is for the string to expand to cope with the input. To achieve that, we have to descend to a lower level of abstraction and deal with individual characters:

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>

void quit()                                    // write error message and quit
{
    fprintf(stderr,"memory exhausted\n");
    exit(1);
}

int main()
{
    int max = 20;
    char* name = (char*)malloc(max);           // allocate buffer
    if (name == 0) quit();

    printf("Please enter your first name:\n");

    while (true) {                             // skip leading whitespace
        int c = getchar();
        if (c == EOF) break;                   // end of file
        if (!isspace(c)) {
            ungetc(c,stdin);
            break;
        }
    }

    int i = 0;
    while (true) {
        int c = getchar();
        if (c == '\n' || c == EOF) {           // at end; add terminating zero
            name[i] = 0;
            break;
        }
        name[i] = c;
        if (i==max-1) {                        // buffer full
            max = max+max;
            name = (char*)realloc(name,max);   // get a new and larger buffer
            if (name == 0) quit();
        }
        i++;
    }

    printf("Hello %s\n",name);
    free(name);                                // release memory
    return 0;
}
```

Compared to the previous versions, this seems rather complex. I feel a bit bad adding the code for skipping whitespace because I didn’t explicitly require that in the original problem statement. However, skipping initial whitespace is the norm and the other versions of the program skip whitespace.

One could argue that this example isn’t all that bad. Most experienced C and C++ programmers would – in a real program – probably (hopefully?) have written something equivalent in the first place. We might even argue that if you couldn’t write that program, you shouldn’t be a professional programmer. However, consider the added conceptual load on a novice. This variant uses nine different standard library functions, deals with character-level input in a rather detailed manner, uses pointers, and explicitly deals with free

store. To use ***realloc***( ) while staying portable, I had use ***malloc***( ) (rather than ***new***). This brings the issues of sizes and casts† into the picture. It is not obvious what is the best way to handle the possibility of memory exhaustion in a small program like this. Here, I simply did something obvious to avoid the discussion going off on another tangent. Someone using the C-style approach would have to carefully consider which approach would form a good basis for further teaching and eventual use.

To summarize, to solve the original simple problem, I had to introduce loops, tests, storage sizes, pointers, casts, and explicit free-store management in addition to whatever a solution to the problem inherently needs. This style is also full of opportunity for errors. Thanks to long experience, I didn't make any of the obvious off-by-one or allocation errors. Having primarily worked with stream I/O for a while, I initially made the classical beginner's error of reading into a ***char*** (rather that into an ***int***) and forgetting to check for ***EOF***. In the absence of something like the C++ standard library, it is no wonder that many teachers stick with the ''shoddy'' solution and postpone these issues until later. Unfortunately, many students simply note that the shoddy style is ''good enough'' and quicker to write than the (non-C++ style) alternatives. Thus they acquire a habit that is hard to break and leave a trail of buggy code behind.

This last C-style program is 41 lines compared to 10 lines for its functionally equivalent C++-style program. Excluding ''scaffolding,'' the difference is 30 lines vs 4. Importantly, the C++-style lines are also shorter and inherently easier to understand. The number and complexity of concepts needed to be explained for the C++-style and C-style versions are harder to measure objectively, but I suggest a 10-to-1 advantage for the C++-style version.

3 Efficiency

Efficiency is not an issue in a trivial program like the one above. For such programs, simplicity and (type) safety is what matters. However, real systems often have parts where efficiency is essential. For such systems, the question becomes ''can we afford a higher level of abstraction?''

Consider a simple example of the kind of activity that occurs in programs where efficiency matters:

```
read an unknown number of elements
do something to each element
do something with all elements
```

The simplest specific example I can think of is a program to find the mean and median of a sequence of double precision floating-point numbers read from input. A conventional C-style solution would be:

```
// C-style solution:

#include<stdlib.h>
#include<stdio.h>

int compare(const void* p, const void* q)          // comparison function for use by qsort()
{
    register double p0 = *(double*)p;              // compare doubles
    register double q0 = *(double*)q;
    if (p0 > q0) return 1;
    if (p0 < q0) return -1;
    return 0;
}

void quit()                                        // write error message and quit
{
    fprintf(stderr,"memory exhausted\n");
    exit(1);
}
```

† I know that C allows this to be written without explicit casts. However, that is done at the cost of allowing unsafe implicit conversion of a ***void**** to an arbitrary pointer type. Consequently, C++ requires that cast.

```
int main(int argc, char* argv[])
{
    int res = 1000;                          // initial allocation
    char* file = argv[2];

    double* buf = (double*)malloc(sizeof(double)*res);
    if (buf==0) quit();

    double median = 0;
    double mean = 0;
    int n = 0;                               // number of elements

    FILE* fin = fopen(file,"r");             // openfile for reading
    double d;
    while (fscanf(fin,"%lg",&d)==1) {        // read number, update running mean
        if (n==res) {
            res += res;
            buf = (double*)realloc(buf,sizeof(double)*res);
            if (buf==0) quit();
        }
        buf[n++] = d;
        mean = (n==1) ? d : mean+(d-mean)/n; // prone to rounding errors
    }

    qsort(buf, n, sizeof(double), compare);

    if (n) {
        int mid = n/2;
        median = (n%2) ? buf[mid] : (buf[mid-1]+buf[mid])/2;
    }

    printf("number of elements = %d, median = %g, mean = %g\n",n,median,mean);

    free(buf);
}
```

To compare, here is an idiomatic C++ solution:

```
// Solution using the Standard C++ library:

#include<vector>
#include<fstream>
#include<algorithm>

using namespace std;

int main(int argc, char* argv[])
{
    char* file = argv[2];
    vector<double> buf;

    double median = 0;
    double mean = 0;

    fstream fin(file,ios::in);                          // open file for input
    double d;
    while(fin>>d) {
        buf.push_back(d);
        mean = (buf.size()==1) ? d : mean+(d-mean)/buf.size();     // prone to rounding errors
    }

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

    if (buf.size()) {
        int mid = buf.size()/2;
        median = (buf.size()%2) ? buf[mid] : (buf[mid-1]+buf[mid])/2;
    }
```

```
        cout << "number of elements = " << buf.size()
             << ", median = " << median << ", mean = " << mean << '\n';
}
```

The size difference is less dramatic than in the previous example (43 vs 24 non-blank lines). Excluding, irreducible common elements such as the declaration of ***main()*** and the calculation of the median (13 lines) the difference is 20 lines vs 11. The critical input-and-store loop and the sort are both significantly shorter in the C++-style program (9 vs 4 lines for the read-and-store loop, and 9 lines vs 1 line for the sort). More importantly, their logic is far simpler in the C++ version – and therefore far easier to get right.

Again, memory management is implicit in the C++-style program; a ***vector*** grows as needed when elements are added using ***push_back()***. In the C-style program, memory management is explicit using ***realloc()***. Basically, the ***vector*** constructor and ***push_back()*** in the C++-style program does what ***malloc(), realloc(),*** and the code tracking the size of allocated memory does in the C-style program. In the C++ style program, I rely on the exception handling to report memory exhaustion. In the C-style program, I added explicit tests to avoid the possibility of memory corruption.

Not surprisingly, the C++ version was easier to get right. I constructed this C++-style version from the C-style version by cut-and-paste. I forgot to include ***<algorithm>***, I left ***n*** in place rather than using ***buf.size()*** twice , and my compiler didn't support the local *using-directive* so I had to move it outside ***main()***. One the other hand, after fixing these four errors, the program ran correctly first time.

To a novice, ***qsort()*** is ''odd.'' Why do you have to give the number of elements? (because the array doesn't know it). Why do you have to give the size of a ***double***? (because ***qsort()*** doesn't know that it is sorting ***double***s). Why do you have to write that ugly function to compare ***double***s? (because ***qsort()*** needs a pointer to function because it doesn't know the type of the elements that it is sorting). Why does ***qsort()***'s comparison function take ***const void**** arguments rather than ***char**** arguments? (because ***qsort()*** can sort based on non-string values). What is a ***void**** and what does it mean for it to be ***const***? (''Eh, hmmm, we'll get to that later''). Explaining this to a novice without getting a blank stare of wonderment over the complexity of the answer is not easy. Explaining, ***sort(v.begin(),v.end())*** is comparatively easy: ''Plain ***sort(v)*** would have been simpler in this case, but sometimes we want to sort part of a container so it's more general to specify the beginning and end of what we want to sort.''

To compare efficiencies, I first determined how much input was needed to make an efficiency comparison meaningful. For 50,000 numbers the programs ran in less than half a second each, so I choose to compare runs with 500,000 and 5,000,000 input values:

Read, sort, and write floating-point numbers						
	unoptimized			optimized		
	C++	C	C/C++ ratio	C++	C	C/C++ ratio
500,000 elements	3.5	6.1	**1.74**	2.5	5.1	**2.04**
5,000,000 elements	38.4	172.6	**4.49**	27.4	126.6	**4.62**

The key numbers are the ratio; a ratio larger than one means that the C++-style version is faster. Comparisons of languages, libraries, and programming styles are notoriously tricky, so please do not draw sweeping conclusions from these simple tests. The numbers are averages of several runs on an otherwise quiet machine. The variance between different runs of an example was less than 1%. I also ran strictly ISO C conforming versions of the C-style programs. As expected there were no performance difference between those and their C-style C++ equivalents.

I had expected the C++-style program to be only slightly faster. Checking other C++ implementations, I found a surprising variance in the results. In some cases, the C-style version even outperformed the C++-style version for small data sets. However, the point of this example is that a higher level of abstraction and a better protection against errors can be affordable given current technology: The implementation I used is widely available and cheap – not a research toy. Implementations that claim higher performance are also available.

It is not unusual to find people being willing to pay a factor of 3, 10, or even 50 for convenience and better protection against errors. Getting the benefit together with a doubling or quadrupling of speed is spectacular. These figures should be the minimum that a C++ library vendor would be willing to settle for.

To get a better idea of where the time was spent, I ran a few additional tests:

500,000 elements						
	unoptimized			optimized		
	C++	C	C/C++ ratio	C++	C	C/C++ ratio
read	2.1	2.8	**1.33**	2.0	2.8	**1.40**
generate	.6	.3	**.5**	.4	.3	**.75**
read&sort	3.5	6.1	**1.75**	2.5	5.1	**2.04**
generate&sort	2.0	3.5	**1.75**	.9	2.6	**2.89**

Naturally, ''read'' simply reads the data and ''read&sort'' reads the data and sorts it but doesn't produce output. To get a better feel for the cost of input, ''generate'' produces random numbers rather than reading.

5,000,000 elements						
	unoptimized			optimized		
	C++	C	C/C++ ratio	C++	C	C/C++ ratio
read	21.5	29.1	**1.35**	21.3	28.6	**1.34**
generate	7.2	4.1	**.57**	5.2	3.6	**.69**
read&sort	38.4	172.6	**4.49**	27.4	126.6	**4.62**
generate&sort	24.4	147.1	**6.03**	11.3	100.6	**8.90**

From other examples and other implementations, I had expected streamio to be somewhat slower than stdio. That was actually the case for a previous version of this program where I use ***cin*** rather than a ***filestream***. It appears that on some C++ implementations, file I/O is much faster than ***cin***. The reason is at least partly poor handling of the tie between ***cin*** and ***cout***. However, these numbers demonstrate that C++-style I/O can be as efficient as C-style I/O.

Changing the programs to read and sort integers instead of floating-point values did not change the relative performance – though it was nice to note that making that change was much simpler in the C++ style program (2 edits as compared to 12 for the C-style program). That is a good omen for maintainability.

The differences in the ''generate'' tests reflect a difference in allocation costs. A ***vector*** plus ***push_back***( ) ought to be exactly as fast as an array plus ***malloc***( ) / ***free***( ), but it wasn't. The reason appears to be failure to optimize away calls of initializers that do nothing. Fortunately, the cost of allocation is (always) dwarfed by the cost of the input that caused the need for the allocation.

As expected, ***sort***( ) was noticeably faster than ***qsort***( ). The main reason is that ***sort***( ) inlines its comparison operations whereas ***qsort***( ) must call a function.

It is hard to choose an example to illustrate efficiency issues. One comment I had from a colleague was that reading and comparing numbers wasn't realistic. I should read and sort strings. So I tried this program:

```
#include<vector>
#include<fstream>
#include<algorithm>
#include<string>

using namespace std;

int main(int argc, char* argv[])
{
    char* file = argv[2];       // input file name
    char* ofile = argv[3];      // output file name

    vector<string> buf;

    fstream fin(file,ios::in);
    string d;
    while(getline(fin,d)) buf.push_back(d); // add line from input to buf

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

```
    fstream fout(ofile, ios::out);
    copy(buf.begin(), buf.end(), ostream_iterator<string>(fout, "\n")); // copy to output
}
```

I transcribed this into C and experimented a bit to optimize the reading of characters. The C++-style code performs well even against hand-optimized C-style code that eliminates copying of strings. For small amounts of output there is no significant difference and for larger amounts of data ***sort***( ) again beats ***qsort***( ) because of its better inlining:

Read, sort, and write strings					
	C++	C	C/C++ ratio	C (no string copy)	optimized-C/C++ ratio
500,000 elements	8.4	9.5	**1.13**	8.3	**.99**
2,000,000 elements	37.4	81.3	**2.17**	76.1	**2.03**

I used 2 million strings because I didn't have enough main memory to cope with 5 million strings without paging.

To get an idea of what time was spent where, I also ran the program with the ***sort***( ) omitted:

Read and write strings					
	C++	C	C/C++ ratio	C (no string copy)	optimized-C/C++ ratio
500,000 elements	2.5	3.0	**1.20**	2.0	**.80**
2,000,000 elements	9.8	12.6	**1.29**	8.9	**.91**

The strings were relatively short (seven characters on average).

Note that ***string*** is a perfectly ordinary user-defined type that just happens to be part of the standard library. What we can do efficiently and elegantly with a ***string***, we can do efficiently and elegantly with many other user-defined types.

Why do I discuss efficiency in the context of programming style and teaching? The styles and techniques we teach must scale to real-world problems. C++ is – among other things – intended for large-scale systems and systems with efficiency constraints. Consequently, I consider it unacceptable to teach C++ in a way that leads people to use styles and techniques that are effective for toy programs only; that would lead people to failure and to abandon what was taught. The measurements above demonstrate that a C++ style relying heavily on generic programming and concrete types to provide simple and type-safe code can be efficient compared to traditional C styles. Similar results have been obtained for object-oriented styles.

It is a significant problem that the performance of different implementations of the standard library differ dramatically. For a programmer who wants to rely on standard libraries (or widely distributed libraries that are not part of the standard), it is often important that a programming style that delivers good performance on one system give at least acceptable performance on another. I was appalled to find examples where my test programs ran twice as fast in the C++ style compared to the C style on one system and only half as fast on another. Programmers should not have to accept a variability of a factor of four between systems. As far as I can tell, this variability is not caused by fundamental reasons, so consistency should be achievable without heroic efforts from the library implementors. Better optimized libraries may be the easiest way to improve both the perceived and actual performance of Standard C++. Compiler implementors work hard to eliminate minor performance penalties compared with other compilers. I conjecture that the scope for improvements is larger in the standard library implementations.

Clearly, the simplicity of the C++-style solutions above compared to the C-style solutions was made possible by the C++ standard library. Does that make the comparison unrealistic or unfair? I don't think so. One of the key aspects of C++ is its ability to support libraries that are both elegant and efficient. The advantages demonstrated for the simple examples hold for every application area where elegant and efficient libraries exist or could exist. The challenge to the C++ community is to extend the areas where these benefits are available to ordinary programmers. That is, we must design and implement elegant and efficient libraries for many more application areas and we must make these libraries widely available.

4 Learning C++

Even for the professional programmer, it is impossible to first learn a whole programming language and then try to use it. A programming language is learned in part by trying out its facilities for small examples. Consequently, we always learn a language by mastering a series of subsets. The real question is not ''Should I learn a subset first?'' but ''Which subset should I learn first?''

One conventional answer to the question ''Which subset of C++ should I learn first?'' is ''The C subset of C++.'' In my considered opinion, that's not a good answer. The C-first approach leads to an early focus on low-level details. It also obscures programming style and design issues by forces the student to face many technical difficulties to express anything interesting. The examples in §2 and §3 illustrate this point. C++'s better support of libraries, better notational support, and better type checking are decisive against a ''C first'' approach. However, note that my suggested alternative isn't ''Pure Object-Oriented Programming first.'' I consider that the other extreme.

For programming novices, learning the programming language should support the learning of effective programming techniques. For experienced programmers who are novices at C++, the learning should focus on how effective programming techniques are expressed in C++ and on techniques that are new to the programmer. For experienced programmers, the greatest pitfall is often to concentrate on using C++ to express what was effective in some other language. The emphasis for both novices and experienced programmers should be concepts and techniques. The syntactic and semantic details of C++ are secondary to an understanding of design and programming techniques that C++ supports.

Teaching is best done by starting from well-chosen concrete examples and proceeding towards the more general and more abstract. This is the way children learn and it is the way most of us grasp new ideas. Language features should always been presented in the context of their use. Otherwise, the programmer's focus shifts from producing systems to delight over technical obscurities. Focussing on language-technical details can be fun, but it is not effective education.

On the other hand, treating programming as merely the handmaiden of analysis and design doesn't work either. The approach of postponing actual discussion of code until every high-level and engineering topic has been thoroughly presented has been a costly mistake for many. That approach drives people away from programming and leads many to serious underestimate the intellectual challenge in the creation of production-quality code.

The extreme opposite to the ''design first'' approach is to get a C++ implementation and start coding. When encountering a problem, point and click to see what the online help has to offer. The problem with this approach is that it is completely biased towards the understanding of individual features and facilities. General concepts and techniques and not easily learned this way. For experienced programmers, this approach has the added problem of reinforcing the tendency to think in a previous language while using C++ syntax and library functions. For the novice, the result is a lot of if-then-else code mixed with code snippets inserted using cut-and-paste from vendor-supplied examples. Often the purpose of the inserted code is obscure to the novice and the method by which it achieves its effect completely beyond comprehension. This is the case even for clever people. This ''poking around approach'' can be most useful as an adjunct to good teaching or a solid textbook, but on its own it is a recipe for disaster.

To sum up, I recommend an approach that

- proceeds from the concrete to the abstract,
- presents language features in the context of the programming and design techniques that they exist to support,
- presents code relying on relatively high-level libraries before going into the lower-level details (necessary to build those libraries),
- avoids techniques that do not scale to real-world applications,
- presents common and useful techniques and features before details, and
- focus on concepts and techniques (rather than language features).

No. I don't consider this particularly novel or revolutionary. Mostly, I see it as common sense. However, common sense often gets lost in heated discussion about more specific topics such as whether C should be learned before C++, whether you must write Smalltalk to really understand Object-Oriented programming, whether you must start learning programming in a pure-OO fashion (whatever that means), and whether a thorough understanding of the software development process before trying to write code.

Fortunately, there is some experience with approaches that meet my criteria. My favorite approach is to

start teaching the basic language concepts such as variables, declarations, loops, etc. together with a good library. The library is essential to enable to students to concentrate on programming rather than the intricacies of, say C-style strings. I recommend the use of the C++ standard libraries or a subset of those. This is the approach taken by the Computer Science Advanced Placement course taught in American high schools [Horwitz,1999]. A more advanced version of that approach aimed at experienced programmers has also proved successful; for example, see [Koenig,1998].

A weakness of these specific approaches is the absence of a simple graphics and graphical user interfaces early on. This could (easily?) be compensated for by a very simple interface to commercial libraries. By ''very simple,'' I mean usable by students on day two of a C++ course. However, no such simple graphics and graphical user interface C++ library is widely available.

After the initial teaching/learning that relies on libraries, a course can proceed in a variety of ways based on the needs and interests of the students. At some point, the messier and lower-level features of C++ will have to be examined. One way of teaching/learning about pointers, casting, allocation, etc. is to examine the implementation of the classes used to learn the basics. For example, the implementation of ***string***, ***vector***, and ***list***
classes are excellent contexts for discussions of language facilities from the C subset of C++ that are best left out of the first part of a course.

Classes, such as ***vector*** and ***string***, that manage variable amounts of data require the use of free store and pointers in their implementation. Before introducing those, classes that doesn't require that (concrete classes), such as a ***Date***, a ***Point***, and a ***Complex*** types can be used to to introduce the basics of class implementation.

I tend to present abstract classes and class hierarchies after the discussion of containers and the implementation of containers, but there are many alternatives here. The actual ordering of topics should depend on the libraries used. For example, a course using a graphics library relying on class hierarchies will have to explain the basics of polymorphism and the definition of derived classes relatively early.

Finally, please remember there is no one right way to learn and teach C++ and its associated design and programming techniques. The aims and backgrounds of students differ and so does the backgrounds and experience of their teachers and textbook writers.

5 Summary

We want our C++ programs to be easy to write, correct, maintainable, and acceptably efficient. To do that, we must design and program at a higher level of abstraction than has typically been done with C and early C++. Through the use of libraries, this ideal is achievable without loss of efficiency compared to lower-level styles. Thus, work on more libraries, on more consistent implementation of widely-used libraries (such as the standard library), and on making libraries more widely available can yield great benefits to the C++ community.

Education must play a major role in this move to cleaner and higher-level programming styles. The C++ community doesn't need another generation of programmers who by default use the lowest level of language and library facilities available out of misplaced fear of inefficiencies. Experienced C++ programmers as well as C++ novices must learn to use Standard C++ as a new and higher-level language as a matter of course and descend to lower levels of abstraction only where absolutely necessary. Using Standard C++ as a glorified C or glorified C with Classes only would be to waste the opportunities offered by Standard C++.

6 Acknowledgements

Thanks to Chuck Allison for suggesting that I wrote a paper on learning Standard C++. Thanks to Andrew Koenig and Mike Yang for constructive comments on earlier drafts. My examples were compiled using Cygnus' EGCS1.1 and run on a Sun Ultrasparc 10. The programs I used can be found on my homepages: http://www.research.att.com/~bs.

7 References

[C++,1998] X3 Secretariat: *Standard – The C++ Language*. ISO/IEC 14882:1998(E). Information Technology Council (NCITS). Washington, DC, USA. (See http://www.ncits.org/cplusplus.htm).

[Horwitz,1999] Susan Horwitz:*Addison-Wesley's Review for the Computer Science AP Exam in C++*. Addison-Wesley. 1999. ISBN 0-201-35755-0.

[Koenig,1998] Andrew Koenig and Barbara Moo: *Teaching Standard C++*. (part 1, 2, 3, and 4) Journal of Object-Oriented Programming, Vol 11 (8,9) 1998 and Vol 12 (1,2) 1999.

[Stroustrup,1997] Bjarne Stroustrup: *The C++ Programming language (Third Edition)*. Addison-Wesley. 1997. ISBN 0-201-88954-4.