VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Teaching C++ Experience at Victoria University of Wellington

Robert Biddle and Ewan Tempero

Technical Report CS-TR-94/18
November 1994

Abstract

In 1992 we introduced C++ as one of the major programming languages used in our teaching at the Computer Science Department at Victoria University of Wellington. We now have three years worth of experience of teaching C++ in both university and professional courses, and this paper is a review of the decisions we have made and the experience we have gained. This paper has three goals. The first is to examine our curriculum design, specifically the choice of programming languages used in our computer science degree programme. We discuss the factors that influenced our choice. The second goal is to present our approach to teaching object-oriented programming in general and C++ in particular. Finally, we discuss our experience associated with teaching C++, in particular focusing on the advantages and disadvantages of the language.

Publishing Information

This paper appeared at the Software Education Conference (SEC'94), November 22-25, 1994. University of Otago, New Zealand

1 Introduction

In 1992 we introduced C++ [8, 5] as one of the major programming languages used in our teaching at the Computer Science Department at Victoria University of Wellington. We now have three years worth of experience of in teaching C++ in both university and professional courses, and this paper is a review of the decisions we have made and the experience we have gained.

We have developed a particular way of introducing and explaining C++, and we have adapted this to several different audiences. Our largest audience is our main second-year course on programming, COMP201 (Program and System Development), and this is the teaching situation we shall discuss in detail. However, we also teach C++ as both a three-day short course and a half day seminar for experienced programming professionals from industry. Teaching to these diverse audiences with different perspectives has provided us with many insights on the C++ language and how best to approach it.

We decided to start using C++ for two reasons. Firstly, we wanted to introduce the object-oriented programming paradigm because we believed it offered the best foundation for the introduction of software engineering concepts. Secondly, we specifically chose C++ because it offered a number of significant practical advantages.

This paper has three goals. The first is to examine our curriculum design, specifically the choice of programming languages used in our degree programme. We discuss the factors that influenced our choice. The second goal is to present our approach to teaching object-oriented programming in general and C++ in particular. Finally, we discuss our experience associated with teaching C++, in particular focusing on the advantages and disadvantages of the language.

2 The move to C++

2.1 Motivation

Prior to 1992, the main programming languages used in our courses were Pascal for our two first year courses, Modula-2 for two of our second year courses (the third, Machine Organization, used assembler), and C for supporting practical work in several of our third year courses. Both first and second year courses used Apple Macintoshs (with Think Pascal and Metrowerks Modula) whereas third year courses used UNIX workstations (with GNU C).

From the student point of view, this arrangement was unsatisfactory. At second year, the students felt that it was odd using a language different to the one they used at first year, but really very similar. Think Pascal "units" resemble Modula "modules", and are used fairly heavily at first year. At third year the students felt that learning to program well in C was a distraction from the main content of course projects in, for example, Operating Systems.

From a staff point of view, the arrangement was also unsatisfactory. At second year, we found it annoying that the systems, software, and programs were different to first year, but really very similar. At third year, we were concerned that student time was diverted from course content in order to learn C. Moreover, we felt that under the pressure of trying to complete the coursework while learning the language, they often adopted poor programming practices in C.

Two other changes were also being considered at second year. For the two second year courses that used Macintoshes, it was felt that Unix workstations might be more appropriate, since that provided scope for programming on a larger scale. For the Computer Organisation course, it was felt that while using assembler was still essential, modern practice indicated that it was more sensible to use assembler together with a higher level language that still allowed close control, such as C.

At the same time, we were asking ourselves whether object-oriented programming was still something for programming languages courses or advanced work, or if perhaps it should be the paradigm we use in the mainstream of our curriculum.

2.2 Language selection

Various factors influence the choice of a programming language for teaching. The main ones are: the particular goals of the courses in which the languages are to be used; the practical implications of the course environment and resources available to support teaching the language; and the longer term requirements of the community in which the students will subsequently work.

In their first year studies, our students learn program design for "programming in the small" and learn about fundamental data structures and algorithms. The main goal of COMP201 is for students to master design and construction of large programs that manage large amounts of data. As such, it represents an introduction to the fundamentals of Software Engineering, and therefore seemed the most appropriate place to introduce object-oriented programming. At first year we do emphasize the advantages of design using "abstract data types" even in small programs. At second year we hope the leverage of object-oriented programming will become clearer with larger programs, where the advantages of design by analogy and of code reusability become significant.

The short term implications of language choice involves further computer science courses. In our department, there are several third year courses that require substantial programming: courses in software engineering, operating systems, computer communications, and computer graphics. For our more advanced laboratory equipment, our preference is for "open systems", systems where the main software components adhere to vendor-neutral standards. In practice this means the Unix operating system and X window system. In this environment, most lower level implementation programming is done in C, and so student projects also require knowledge of C.

The longer term implications of language choice involve professional practice programming in the wider community. Professional programming in New Zealand spans an enormous range, with COBOL continuing in large legacy systems, continued use of assembler in embedded applications, a solid core of use of C and "3rd" generation languages for system development work, and SQL and "4th" generation languages popular in more modern commercial applications. But the trend seems to be toward SQL and 4GL's for high-level work, and an object-oriented 3GL for lower level application programming as well as system programming; for this work, the demand is for expertise in C++. While there is some research work done in New Zealand with more advanced or experimental languages, this represents only a tiny proportion of the community.

2.3 Deciding on C++

What we did was to change our second year courses from Modula-2 to C++. At the same time, we moved from Macintoshes to Unix workstations, and the compiler we chose was GNU C++. We left our first year and third year courses unchanged.

The feeling was that this step addressed many concerns all at once. We made the transition from first year to second year more meaningful, moving from a language well set up for programming in the small, to one well set up for programming in the large. We made the transition from second year to third year more straightforward, because the programming skills the students develop in second year are directly useful in their practical work at third year, because C++ is a superset of C. At the same time, we introduce a language that supports the object-oriented paradigm, and we enable to the students to master a language in demand by local industry.

Of course, the decision to move to C++ at second year was only one of many choices available to us, but it seemed to satisfy the most constraints.

We could have moved to Modula-2 at first year. In fact, we still regard this move (or one to Oberon) as attractive. However, for first year students learning to program, we feel the software available for Pascal is clearly superior to that available for Modula or Oberon. Many of our first year students are completely new to programming, including some who go on to become our best students. We feel that the software environment is therefore very important, and must support the learning experience by allowing the student to focus on the important programming issues, rather than implementation details of the programming environment. Moreover, the Pascal language we use is an extension from the standard, and offers both some facilities helpful for beginners (for example, a built-in string type) and some module facilities similar to those in Modula. We do not feel it is important to constrain our use of Pascal to the formal standard, both because of the learning situation and because the extensions we use are de-facto standards themselves.

Another alternative would have been to use C++ at first year, but we consider this inappropriate for a number of reasons, and discuss this issue in section 4.2.

We could have kept Pascal at first year, but changed second year to another language. The language would have to have supported programming in the large, and also allowed introduction of the object-oriented paradigm. For example, we might have considered Smalltalk, Eiffel, CLOS, Modula-3, or Objective-C.

The argument for any of these languages over C++ is that they present a clearer vision of the object-oriented paradigm. The arguments against are pragmatic. None of these languages has implementation software as widely and easily available as C++. None of these languages is as widely used as C++ in software implementation for practical work in other areas. None of these languages are as widely used in New Zealand industry as C++.

The key question was this: are the advantages of the other languages sufficient to outweigh the disadvantages? We decided they are not. While there is some disagreement over what it means to be "object-oriented", there are a number of common concepts. These include data abstraction, encapsulation, and polymorphism (usually supplied via inheritance), which C++ has explicit support for. C++ also has other features expected of modern programming languages, such as support for generic types and overloading of functions and operators. Of particular pedagogical interest is the fact that its design was specifically intended to support user-defined types as first-class constructs [7].

Even if we had moved to Modula at first year, or another language at second year, it still would have left open the question of supporting use of C later on. Of course, we could change that too. However, many of our third year and more advanced courses involve practical work with software systems that are written in C. Aside from the implications of changing practical aspects of many courses, moving away from C would be questionable. Avoiding C at these levels would not allow our students to work with much important practical software in use today.

In summary, C++ provided sufficient support for teaching the Object Model, was appropriate in the context of other courses we taught, had reasonable compiler and textbook (we thought) support, and was important to the local information technology industry.

3 Teaching C++

3.1 Assumptions

As mentioned in the introduction, we teach C++ in two settings: short courses for professional programmers, and as part of our degree programme at second year. But as both these audiences should be able to learn a new programming language by themselves, the question must be asked: why teach it at all? The answer is that really learning C++ involves understanding the object-oriented approach to programming, and this is not trivial. Indeed, many programmers in our professional courses explain that they have read magazine articles, and flipped through C++ books, but somehow found them unhelpful.

We have long been disturbed than much of the literature about object-orientation has explained the approach in very abstract terms, and proclaimed its advantages on the basis of very vague reasoning, or simply in terms of a bandwagon of popularity. It has been our intention to take a different line, and rely on practical explanations. We have decided that a good understanding of the principles of the objected oriented approach are best developed at the same time as a good understanding of a practical language, and so try to develop both at the same time.

There are a number of aspects to the object-oriented approach to program design, and a number of corresponding features of C++. These work together in many ways, and it was not always clear at the beginning what the the order of presentation should be. We found that we were unhappy with the teaching strategy usually used in textbooks on object-oriented programming and C++, and have gradually evolved a teaching strategy of our own that differs in several ways.

In all our teaching of C++, we assume a familiarity with programming in C. In our short professional courses, this simply matches the demand: there are many programmers familiar with C who wish to extend their knowledge to C++. In our degree course this is not the case at all, and most of our students know only Pascal. Because we intend to introduce the object-oriented approach and C++ at the same time, we first review the general principles of program design. In our degree course, we use this review to recap the features of Pascal, and introduce the features of C at the same time. With some lab exercises, the students are familiar enough with C for us to continue with C++.

3.2 Presentation sequence

We begin with an outline of object-orientation as an approach to program design, discussing the issues of data design and of control design, and stressing the importance of abstraction and encapsulation. (Our degree students are familiar with these concepts from our first year courses.) We show how these ideas work together in the object oriented approach with the use of abstract data types, and the independent specification of their interface and implementation. We then outline how such types work together, and explain the wider benefits of the approach of design by analogy.

At this point we introduce the C++ "class", and explain it in terms of the previous discussion. We explain how it specifies a data type, with the interface and implementation detailed separately, and we show both how to define classes and use classes already defined. All the basic features of the class are introduced: public and private sections, how member functions access components, and how constructors and destructors work.

While the traditional approach to teaching C++ (or any object-oriented programming language) would be to introduce inheritance at this point, we have found it more valuable to look at more complex classes without inheritance.

We first spend some time discussing "composition" of classes with other classes as components. This should really be nothing new to C programmers familiar with structure types with members that are themselves structures or pointers to structures, or indeed to Pascal programmers familiar with records with fields that are records or pointers to records. However, we have found it important to legitimise this kind of construction because it offers significant opportunities for code reuse that are often overlooked.

We next discuss "container" classes, including lists, sets, and maps, where component classes pass through the interface. At this point we introduce implementation strategies for use of dynamic memory structures, such as linked lists, using node classes.

We also take this opportunity to discuss an apparent anomaly of the C++ class. While there is a separation of public and private components, both must be specified together and the specification of both must be provided whenever the class is used. This rightly disturbs many learners of C++, but it is does allow greater run-time efficiency in code accessing objects. We introduce a class implementation technique we call "opaque implementation" that simply uses pointer indirection to allow complete separation of private and public components, and also allows compilation independence of code defining a class, and code using that definition. Most object-oriented languages always use this technique, though invisibly. We make the point that C++ allows the programmer to choose between greater efficiency and greater independence.

Because the most sophisticated classes we have discussed are container classes (such as sets, queues, etc.), we move next to generic containers with parameterised types as made possible with the C++ "template" facility. This is also where we stress the importance of overloaded functions and operators, because they enable generic member functions (and also operators) to be bound appropriately for the type parameters. This allows creation of very useful reusable code, and we encourage students in creating generic resuable container classes, where the component types can be specified later.

Only at this point, well into our presentation of object-oriented programming, do we introduce derived classes. After explaining the basics, we spend some time comparing derivation with composition, introduced much earlier. This is an important issue to us, because we find that many students (indeed many textbooks) mistakenly suggest that the main advantage of derivation is that it allows reuse of the methods of the base class. We point out that both derivation *and* composition allow such reuse, and that neither break encapsulation. We explain that the distinct advantage of derivation is that it allows definition of a new class that conforms to the interface of the base class. This in turn allows reuse, but reuse of code that *uses* the base class: it may now be used in connection with the derived class as well.

This brings us to polymorphism, where code written for a base class is not only reusable for derived classes, but may have different effects depending on the particulars of the derived class. We motivate the need for polymorphism by containers, and show how with C++ "virtual" member functions, items obtained from a container may be processed differently according to their specific class.

This discussion connects well with the earlier discussion of generic containers, where the container must involve items of the same type and offers type safety in return. In contrast, if the items in a container are members of a base class, and possibly members of derived classes, then the container can be regarded as involving items of different types, and yet still offering type safety. Moreover, the container itself can still be created from a generic template.

With this understood, we then introduce abstract base classes, showing how functions may be written in terms of completely abstract containers, and then used for different concrete implementations in the same program.

Finally, we introduce multiple inheritance, still a controversial addition to C++ [3, 9]. To explain the details, we distinguish three cases of multiple inheritance. In the "disjoint" case, we explain the reasonably straightforward situation were a class is derived from completely distinct base classes; in the "parallel" case, we point out how there are some minor complications if those base classes are in turn derived from distinct uses of the same base class; and in the "joint" case we show how a special facility is needed for multiple inheritance from base classes in turn derived from a single base class. We present what we feel are convincing examples of these cases, but we leave the students with questions about the real need and utility of multiple inheritance.

We begin the course with a philosophical introduction about the benefits of object-orientation, and we conclude with a very practical review of the techniques as they relate to the creation of reusable code. In fact, organising these courses has drawn us to examine closely the nature of reusable code, and we have found the subject rewarding and well worth further investigation [1, 2].

Throughout this teaching program, we pause after every main topic for work with lab exercises. The exercises involve using classes created beforehand, creating new classes, and adapting and developing programs involving all the main features of the C++ language. All of the lab exercises stress the student having to apply the facilities of C++ in an appropriate situation to complete or extend a meaningful program. These are all small exercises, designed for the student to check and develop their understanding, and resulting in useful examples for later reference. In our degree course, this is followed by a major design and implementation project.

4 Evaluation

4.1 Disadvantages

Although C++ has become very widely used, it is not without criticism. It is sometimes regarded as a poor language to teach the object-oriented paradigm, because it is not a "pure" object-oriented language, since it allows data and code outside the class structure. This has not been a problem for us as we teach the object-oriented approach along with the language, so the students understand both the fundamental principles and how to use C++ in accordance with those principles.

Another criticism of C++ is that it does not facilitate use of dynamically allocated data with automatic garbage collection. The lack of garbage collection itself has not been a problem, and students have no particular difficulty understanding the responsibilities relating to allocation and deallocation of dynamic structures. However, there are significant problems with some key C++ facilities involved in this management: constructors and destructors. These functions are implicitly called when objects are created and deleted, and allow much more efficient implementation than garbage collection. However, a significant number of errors occur either in these member functions, or as a result of not realising when these functions are being called. The way in which constructor and destructor functions are invoked without explicit function calls continually trips up even experienced programmers, and is particularly frustrating to novices.

One continually bothersome aspect of C++ is that the language does not provide any mechanism to formally indicate or document what classes or other facilities must be provided externally, or are being provided for external use. Even Modula-2, with its import/export declarations, was vastly superior. Instead, in C++, the convention is simply to use preprocessor directives to textually include files containing the class definitions. The hope is that the documentary effect, and any external program construction environment, are sufficient to indicate what is necessary. This is crude and obscures important principles, though it must be admitted that the conventions are well adhered to, and few problems seem to arise.

A problem with using C++, not related to the specifics of the language itself, is the fact that it is not yet a completely stable language. While there are plenty of compilers available, the differences between them can be significant. However, we hope this problem diminishes once the ISO standard is determined. Currently, most compilers do implement templates, and we cover this in our teaching. However, most compilers do not implement exception handling, and so we do not cover that.

Finally, and perhaps surprisingly, we have found textbook support inadequate. There are a tremendous number of books on C++ available; however they are all inadequate as far as explaining the relationship between the language features of C++ and the object-oriented programming paradigm. Moreover, we find irritating and too common the practice of explaining both language features and program design approaches with appeals to what is "natural" instead of with practical reasoning. Where examples are given showing how the C++ features are used, contrived and meaningless examples are unfortunately the norm. The book we currently recommend is in fact Stroustrup's own book [8], mostly because it is perhaps the most comprehensive book available. We also recommend a more advanced book by Meyers [6] that addresses a number of key issues, and suggest students study Cline's FAQ (Frequently Asked Questions) for the Usenet newsgroup `comp.lang.c++` as a very good, compact, and free guide to C++. (The FAQ has recently also been published in book form [4]).

4.2 Advantages

A number of the advantages of C++ concern practical considerations, but the language itself is also attractive in its own right. Fundamentally, C++ does offer all the key components necessary to support programming in the object-oriented paradigm. While there is no universal agreement about the complete list, C++ certainly provides the key essentials: user defined types, encapsulation, separation of interface from implementation, composition and inheritance, and run-time polymorphism. We have found that as we teach about the paradigm, C++ is able to support the teaching well. Even for more theoretical courses with some practical work, C++ has served our department well. For example, one colleague has found C++ quite satisfactory for practical examples involving formal languages and automata, and students use it for related project work.

In some cases, C++ offers facilities that are not well connected with the object-oriented approach. The "template" facility, for example, essentially allows parameterised user-defined types especially useful in constructing generic containers. Often in object-oriented programming, this is done using inheritance and polymorphism with the contained items. However, the template approach offers greater type safety, and increased run-time efficiency. Of course, the other approach is still possible, and indeed desirable where the flexibility is necessary.

One of the most distinguishing characteristics of C++ as an object-oriented language is the concern for run-time efficiency. In fact, programs written in C++ need be no less efficient that programs written in C. In many practical programming situations, this is important, and at least part of the reason for the widespread use of C++. The impact of this priority on the language is quite clear: this is, for example, the reason for the lack of automatic garbage collection, for the binding of public and private class components together, and for the need for the explicit "virtual function" for run-time polymorphism.

Interestingly, we have found that these aspects of the language actually facilitate student learning, rather than impeding it. There are two reasons we see involved with this. Firstly, students learning C++ are also able to see clearly how it works, because the operational structure of the language is explicit. Secondly, C++ features offer more choice to the programmer, particularly between efficiency and convenience. By explicit discussion of these trade-offs, the students better appreciate the nature of the language and of the programming paradigm.

Of course, there are various advantages of C++ that are very practical in nature. As we discused earlier, the compatibility of C++ and C is useful because many of our students must later do work with various software implemented in C. This has worked out well, as students that haved learned C++ at second year deal easily with C where necessary at third year. Beyond this, the employment market for professional programmers highly values C++ expertise. We do not make any formal effort to find out job placement of our degree students, but the informal evidence suggests C++ experience is a useful asset.

Just the widespread use of C++ has advantages. Much software is available written in C++, for example, and it is has become a language often used in examples in books and journals. Moreover, C++ software itself is very available. This includes an increasing variety of C++ compilers, debuggers, editors, libraries, and so on. One important consequence of this is that for any architecture, for any operating system, C++ is one of the few languages you can assume will be available. Our students increasingly have computers at home, and they can easily obtain C++ development software.

For our degree students, the transition from Pascal to C++ seems to be worthwhile experience. After their experience in our first year program, they are quite confident in their understanding of Pascal, and the fundamentals of programming. As a result, they are well motivated to focus on issues of design, and approach both the object-oriented approach and C++ with curiosity but also confidence.

With regard to the object-oriented approach itself, we are also satisfied that second year is the appropriate time for an explicit introduction. We continue to foreshadow the approach in our first year teaching, and both encapsulation and abstract data types are issues our students become quite familiar with. However, our emphasis is mainly on design of procedural programs, algorithms, and data structures. In teaching, there are always difficulties in deciding whether to deal with first with specific cases or with broad concepts. This is especially difficult issue with regard to programming, because most of our first year students start with only superficial understanding of programming, or indeed with no knowledge of the subject at all. In this situation, we feel that the fundamentals of procedural programming are the correct priority, interwoven with an introduction to the concepts design by of procedural abstraction. With this foundation, and by attention to issues first of encapsulation and later introduction abstract data types, students are well prepared to fully understand the object-oriented approach.

In fact, moving to the object-oriented approach and C++ at second year combines two worthwhile experiences into one. Firstly, the learning of second programming language is an important step for the students, and an important skill for the future. Secondly the students deepen their understanding of program design, especially with regard to issues important for larger programs, where reusability and adaptability are crucial.

5 Conclusion

We adopted C++ as a teaching language for a variety of reasons. It allowed us make a practical introduction to object-oriented programming, while also allowing our students to learn a language in widespread use and demand. Also, the broad support for the language has meant easy availability of software and reference material.

Even so, we have found it necessary to develop our own approach to teaching with the language. By far most of the introductory literature presents the language and the programming paradigm without careful explanation and critical reasoning. Our approach takes a "no-nonsense" attitude, and develops an understanding of the paradigm and the language based on software engineering principles of adaptability and reusability.

Of course, C++ has disadvantages and advantages. The disadvantages seem to be: intricacies involved in constructors and destructors and calls to them, the lack of clear module import/export mechanism in C or C++, and inadequacies in textbook support. The advantages of teaching with C++ seem to be: the implementation of the key concepts of the object-oriented paradigm, the usefulness of the connection with C, and the various benefits of a widely supported language in demand by industry.

We feel the advantages outweigh the disadvantages. We consider our move to C++ to have been a success, and feel that the language fits into our curriculum well. Moreover, our experience has caused us to closely examine question the conventional approaches in understanding issues of adaptability and reusability in program and programming language design, and we are keen to more deeply understand the principles involved.

References

[1] Peter Andreae, Robert Biddle, and Ewan Tempero. How to reuse code, really. In *Proceedings of the 1993 New Zealand Computer Society Conference*, Auckland, New Zealand, August 1993. New Zealand Computer Society.

[2] Peter Andreae, Robert Biddle, and Ewan Tempero. Understanding code reusability: Experience with C and C++. Technical Report CS-TR-93/12, Victoria University of Wellington, Wellington, New Zealand, December 1993.

[3] Tom A. Cargill. Controversy: The case against multiple inheritance in C++. *USENIX Computer Systems*, 4(1):69–82, winter 1991.

[4] Marshall P. Cline and Greg A. Lomow. *C++ FAQs: Frequently Asked Questions*. Addison-Wesley, 1995.

[5] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[6] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, Professional Computing Series, 1992.

[7] Bjarne Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3), May 1988. Also appeared in ECOOP'87, pp51–70.

[8] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

[9] Jim Waldo. Controversy: The case for multiple inheritance in C++. *USENIX Computer Systems*, 4(2):157–171, spring 1991.