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

What every manager should know about C++

Robert Biddle and Ewan Tempero

Technical Report CS-TR-95/5
May 8, 1995

Abstract

C++ is the most commonly used object-oriented software development tool in New Zealand. Like any tool, C++ works best when it is used properly. This requires understanding what its intended use is, how to use it effectively, and what its benefits are. In this paper we explain the philosophy of C++ and discuss its costs and benefits. Our intended audience is people who have not had much exposure to object technology and C++, however we expect that more experienced readers will find some points of interest.

Publishing Information

This paper appeared in the proceedings to Uniforum '95, May 1995.

C++ is the most commonly used object-oriented software development tool in New Zealand. Like any tool, C++ works best when it is used properly. This requires understanding what its intended use is, how to use it effectively, and what its benefits are. In this paper we explain the philosophy of C++ and discuss its costs and benefits. Our intended audience is people who have not had much exposure to object technology and C++, however we expect that more experienced readers will find some points of interest.

Management must understand object technology to get its benefits

A common misconception with any technology is that only the "techies" need to know anything about it. This is not true of technology in general and it is particularly not true of object technology. The work habits required to make the most effective use of object technology are different from those required for traditional software development techniques. Management must recognize these differences when allocating resources and planning schedules. To best determine resources and schedules, management must know what the costs and benefits of the technology are, which means must have a good enough understanding of the principles of object technology to appreciate the implications.

C++ is an object-oriented programming language based on C

There are many styles of programming. The most fashionable one at the moment is "object-oriented" programming. C++[7] was designed by Bjarne Stroustrup during the early 1980s to be an object-oriented version of C[4].

C is a systems programming language, that is, it provides easy access to the hardware thus allowing efficient code. Stroustrup chose C because it was a language that had no inherent limitations to the kinds of programs that could be written in it, C was available on practically any platform in existence, and if it wasn't available, it was fairly straight-forward to port it. He saw the main problem with C as being that it did not provide much help in organizing large programs. He chose to extend C, rather than create a new language from scratch, because it was much quicker to do so.

C++ is the result of Stroustrup's choice of C and his very pragmatic design philosophy [8]. For example, he only included features when he was convinced they would be used to solve real problems, rather than because they had some theoretical elegance. He also wanted the language to be available immediately, and was prepared to accept small deficiencies rather delay release. He also wanted to keep compatibility with C to provide an upgrade path, however he was also prepared to depart from C when the C feature was considered harmful [5].[1]

Object-oriented programming can be done in any language (to some degree) but it requires following conventions. The difference between *allowing* a style of programming and *supporting* it, is whether the user can unknowingly break the conventions. C++ enforces the object-oriented programming conventions. You can break these conventions if you really want to, but it requires explicit actions on your part — you won't be allowed to do so by accident. So, in order to understand how to use C++ effectively, you must understand the conventions associated with object-oriented programming.

Object-oriented programming is programming by analogy

When we write programs, we are dealing with analogies: a program that manages an employee database does not deal with the actual employees but instead manipulates software analogies

[1] In fact, the design of C++ affected the standardization effort for C.

of employees. One of the main problems when developing software is finding the right analogy. (Should the database keep track of hair colour? It depends on whether the database is to be used for producing the payroll for an Accounting firm or ordering makeup for a Model Agency.) What object-oriented programming gives us are more (and more accurate) analogies to use. To see what analogies object-oriented programming can provide, we just need to look at what's in the real world.

Programming by analogy improves communication with customers

Probably the greatest benefit of programming by analogy is that it improves communication. If the software analogy accurately reflects the real world, then it will be about as complex as the real world. We can then justify the cost of the software by showing the customer how complex their problem really is. This isn't to say that it will be easy (the world is usually more complicated than people realize), but at least we can explain things in terms that the customer is more likely to understand. Furthermore, if the requirements of the application change, the cost of the changing the software is going to be similar to the degree of complexity of the changes. Since the customer will have some idea of how complex the changes are, there should be no nasty surprises when we tell them how much their changes will cost.

For example, suppose we have built a payroll application and the customer decides that, instead of mailing payslips to the employee's home address, as is currently done, the payroll application should allow the employee the option of specifying a separate mailing address. The customer will probably think this is a fairly straight-forward change, and so is likely to be upset if we tell them the change will take several months. However, if the analogy we have chosen matches what the customer expects, such a small change should be cheap to implement.

The reverse is also true. If the customer requests a change that they perceive as being quite complex, they are not going to be surprised when you present them with a large bill. For example, if the payroll customer comes to us (probably with a very mystified look!) and tells us that they have discovered that half of their employees are "zombie-doubles" of the other half. That is, while there are two bodies, there is really only one person, and so should only be paid once. Since this represents quite a large difference from the original real world analogy the customer was using, they will expect the change in the software to be expensive.

Programming by analogy gives a place to start

The biggest problem with software development is deciding where to start. By creating software that is analogous to the real world, we have a good place to start — the real world. Furthermore, if a problem arises during development, the fact that the software is supposed to be analogous with the real world can indicate the nature of the problem. If less time is spent dealing with these problems, the time (and thus cost) of the software is reduced.

This does not mean that software development becomes easy using object technology. The real world is a complicated place [2] and the world that we, as software developers, have to deal with is even more complicated. Not only do we have to deal with physical entities as they appear in the world (such items in an inventory or device controllers), but we also have to deal with non-tangible entities we humans use to deal with life (addresses, schedules, records, encryption) and even abstract ideas that are not well-defined in the real world (low-pressure weather systems, schools of fish). Deciding what the analogy is requires the developer and the customer agreeing on a good description of the problem.

[2] For example, according to Richard Feynman, light doesn't travel in straight lines! [3]

Even a good description of the problem can be misleading as it will usually contain concepts that must be present to provide the context of the problem, but do not need to appear in the application. (The application is "Manage boats for rent on a lake". Should "lake" be represented?) Finding the right analogy requires a complete understanding of the functional requirements of the application.

Object-oriented programming can result in more reusable code

Reusable code is the holy grail of software development. Reusable code reduces the time it takes to build new applications because that code does not have to be reinvented (and rewritten and redebugged). Reusable code makes applications easier to understand because there is less new code to learn, and the same code is used when the same thing happens (instead of code that has been rewritten that only looks similar). Reusable code reduces maintenance costs, because if it turns out to be broken, it only has to be fixed in one place. In other words, having reusable code can significantly decrease the cost of software.

Object-oriented programming is about creating analogies with entities in the real world. Many applications deal with the same part of the real world. If they are designed around their functional requirements, it is less likely their code can be shared. This is certainly true if their functional requirements are different (for example, car mechanical repair versus panel beaters). However, even if the functional requirements are similar, the context in which they must be met may prevent code from being shared (private car repairs versus fleet car repairs). If applications are designed around the entities they have to deal with, rather than their functional requirements, then applications that have to deal the same entities (cars) can probably share the code corresponding to those entities.

Object-oriented programming can result in more reusable code, but there is no guarantee. An organization will not create much reusable code if it does not explicitly try to do so. Every time a class is created, the question "how might this be reused" must be asked. This does increase the initial cost of the software and so must be budgeted for. However, if the right choices have been made, this cost can be amortized over many applications.

Object-oriented programming is about objects

To create a software analogy of the real world, we need to represent the entities in the world that we are interested in. We represent the entities as *objects*. So object-oriented programming is all about deciding what objects are needed, what exactly they do (their responsibilities), how they are related, and how they cooperate to solve the problem.

When we create a software analogy of something, we have to decide what details are important and what aren't. (Do we need hair colour in an employee record?) The process of deciding what details to include is called *abstraction*, and so objects are often described as abstractions of real-world entities.

Objects in the real world are usually thought of in terms what we can do with them. When we're driving a car, we generally think in terms of the steering wheel, the gear shift, and the ignition. We generally don't think about the complicated stuff, such as water pump, that's under the bonnet. The way we expect to interact with an object is its *interface*, which corresponds to the *operations* the object supports (or the *messages* it understands).

When we use the interface of an object correctly, the object will behave in a certain way. If we circumvent the interface, not only will we not get the expected behaviour, we also risk breaking the object. The inner workings of real world objects are generally protected from such circumvention by being hidden below, that is, *encapsulated by*, its cover. This isn't to say we can't mess with the inner workings, we can, but it requires a conscious effort on our part. The encapsulation of entities provides certain guarantees as to its behaviour, we want the same guarantees to hold for software objects.

Objects collaborate to solve problems

It is useful to think of each object is a robot, that is, a mechanical device that operates under its own power. These robots generally do their own thing, although their behaviour may be affected by other robots (for example "ATM", "account", and "account balance" robots). An object-oriented application is a group of objects that cooperate to solve a problem. The cooperation consists of the objects sending messages to each other (the ATM robot sends the "debit $1000" message to the account robot).

An object performs different actions depending on the kind of message and message's contents ("debit $1000" is different from both "debit $100" and "credit $1000"). The actions it performs may result in it sending messages to other objects ("subtract $1000" from the account balance robot). An object will only respond to certain messages (accounts will not respond to the "validate PIN" message). The set of messages it responds to corresponds to its responsibilities in the system (accounts are not responsible for dealing with PINs).

Object-oriented programming includes classes, composition, inheritance, polymorphism, and genericity

Many entities in the real world are identical except for their identity (My brown 1987 Nissan Bluebird SGS is not the same as your brown 1987 Nissan Bluebird SGS; employee Jones is not the same as employee Smith). We group the corresponding objects into *classes*. Every object is an *instance* of a class.

A car is made up of many parts (engine, wheels, water pump, radiator) — we represent this kind of relationship between objects using *composition*. Nissan Bluebirds are not Toyota Corollas but they share certain characteristics with the generic concept of "car" — we represent this kind of relationship between classes by creating a class that represents the shared characteristics which are then *inherited* by the original classes (Nissan Bluebirds inherit the characteristics of "car"). Many different things respond to the same operation (change gear) but in different ways (different gearboxes). This different behaviour for the same operation is represented using *polymorphism*. There are many kinds of "containers" in the real world. Some place few restrictions on what can go in them (boxes), others place restrictions to varying degrees (you can put anything with a loop-like attachment on a hook, and car transporters can hold any kind of car, but only cars) — we represent this concept with *genericity*.

Object-oriented programming is *not* just inheritance

Most descriptions of object technology say (explicitly or implicitly) that the most important concept is inheritance (or polymorphism). While it is true that the introduction of inheritance is what sparked the concept of "object-oriented programming", most of the benefits of object technology are due to organizing software around objects and classes — inheritance is just a small part of that.

Code can be reused without inheritance

The biggest myth associated with object-oriented programming is that it is good because inheritance allows reuse, with the implication that we need inheritance before we can reuse code — this is not true. Reuse is also possible using composition, a technique that has been around since before the object-oriented craze, and even now, far more code is reused with composition than it is with inheritance. Also, genericity is another way to reuse code, and again, more code is reused that way than with inheritance. Furthermore, the kind of reuse usually associated with inheritance can be simulated with composition.

In fact, the real benefit of inheritance is that it allows polymorphism, and it is the flexibility that polymorphism provides that is new. It is important to distinguish between the mechanisms that allow code to be reused (which include "cut-and-paste" with an editor), and mechanisms that make the code easier to reuse, that is, that make the code more reusable. Object-oriented programming tends to encourage reusable code because when real-world entities reappear in new applications, their software analogies in old applications can be reused. By making relationships such as composition and inheritance easier to represent, more ways of reusing code are available. Polymorphism and genericity can be used to make software more flexible, and thus more reusable.

C++ is mainly C

C++ is often described as a large language, much bigger than C. However, there are only a few syntactic differences between C and C++. The most important difference is the `class` construct, and even that is similar to the original `struct` construct in C. The support for inheritance (derived classes), polymorphism (virtual functions), genericity (templates) are really just minor variations on the class construct. The other commonly-cited differences between C and C++ (for example, overloaded operators) are also only minor variations on existing C syntax. What is different about C++ is the programming philosophy. It is here that the consequences of the new syntax have the most effect.

C++ supports classes, interface, encapsulation, and composition with `class`

The C++ `class` construct declares a new class. It specifies what operations the class understands, what the implementation details are, and where the encapsulation boundary is.

The example below declares the class `Account`, which has the operations `credit`, `debit`, and `balance` in its interface (as indicated by `public:`). Each `Account` is *composed* of an integer and an `TransactionList` (a class declared elsewhere). These components are part of the inner workings of an `Account` (as indicated by `private:`).

```
class Account {
public:
  void credit(int amount) {
        current_balance += amount;
        // record transaction
  }
  void debit(int amount) {
        current_balance += amount;
        // record transaction
  }
  int balance() {
        return current_balance;
  };
private:
      int current_balance;
      TransactionList audittrail;
};
```

An instance of a new class is declared just like any other variable in C++.

```
Account my_account;
```

When applying an operation to an object, both the object and the operation must be specified.

```
my_account.credit(1000);
```

However the parts declared private may not be accessed

```
my_account.current_balance = 1000000; // error!
```

C++ supports inheritance with derivation

In C++, the class that represents the shared characteristics is called a *base class*; The classes representing the differences and related by inheritance are called *derived classes.*

```
class Savings_Account : public Account {
public:
  int interest_earned();
private:
  float interest_rate;
};
```

In the example above, `Savings_Account` is derived from the base class `Account`. A `Savings_Account` is just like an `Account` but it has the operation `interest_earned()` as well as `credit`, `debit`, and `balance`. It also has another component. The use of `public` in the first line indicates that the relationship between these classes is public knowledge.

A consequence of this relationship is that `Savings_Account` may be used anywhere that `Account` is expected. The same is true of pointers to those classes:

```
Account* bank[10];   // Array of pointers to Account
Savings_Account* acct = new Savings_Account; // Make a new savings account
bank[0] = acct;      // Put a Savings_Account where an Account is expected
```

C++ supports polymorphism with virtual functions

It may be that when we create a derived class, we want to keep the name of one of the inherited operations but want it to behave differently. For example, the `debit` operation on a `Savings_Account` may also deduct a $5 fee. We can get the new behaviour by redeclaring the function we want to change, however that means there are now two versions of that function. This raises the question as to which version should be called.

```
bank[0]->debit(100); // Which debit behaviour?
```

Usually we want to call the version associated with the derived class. To achieve this, we must identify the *base class* functions whose behaviour may be changed in derived classes. We do so by declaring such functions `virtual`.

```
class Account {
...
  virtual void debit(int amount);
...
};
```

C++ supports genericity using `template`

There are situations where we find ourselves writing almost the same code more than once, where the only difference is the type of some things. For example, the code for `AccountList` would differ from the code for `TransactionList` only by the use of `Account` instead of `Transaction`. We can avoid this duplication in C++ using the `template` construct.

```
template <class T> class List {
public:
  void insert(T item);
  ...
};
```

In this example, the type that goes into the `List` class is left unspecified. When we need an actual list of something, we need to instantiate the template, by specifying the type of the items in the list.

```
List<Transaction> audittrail;
List<Account> bank;
```

C++ may not be the "best" object-oriented language, but it does OK

There are many claims that C++ is a "bad" language. It is true, it has some problems. Many of them are due to C++'s C ancestry, such as weird syntax, use of a pre-processor, and the file-based nature of program management. Stroustrup acknowledges them — he calls them "second-order defects", that is, they are a nuisance, but any half-way competent programmer can easily learn to deal with them. There are also complaints about the size of C++. Again, this is partly due to C++'s C ancestry. However there are also a number of features that are there to allow the creation of efficient code (such as inlining). This does make C++ more complicated, but complexity is the cost of high-performance in any engineering discipline.

Another major complaint is that C++ does not have garbage collection (automatic management of dynamic memory). Stroustrup deliberately excluded garbage collection because he was concerned about the time and space overheads. Perhaps the biggest problem is the lack of hand-holding that C++ provides. Stroustrup did his best to avoid dictating how to write programs by adopting a "policy free" design. The consequence is that programmers may create problems for themselves though not understanding how some features interact. This is something that must be taken into account when choosing to use C++.

There are costs associated with changing to C++

The main problem with C++ is that it is a bit of a moving target. The language is still evolving to some degree, and so there are incompatibilities between C++ compilers from different vendors. However there is a standard being developed and the vendors are tracking it, and so there will be agreement in time. In the meantime, care must be taken to avoid depending upon non-standard features.

Another problem is the lack of training material for beginners. There are a huge number of books that claim to teach C++, however they generally tend to concentrate on the syntax and obvious points. At the moment, the best bet is probably Stroustrup's book [7]. It is not the greatest teaching book in the world, but it is relatively complete and so makes a useful reference to have available.

There are benefits associated with changing to C++

Although there are differences between compilers, at least they exist. There is more likely to be a C++ compiler for any hardware platform than any other language, and in many cases, the compiler you need is easily available. Despite the poor quality of training material, at least it exists, and the sheer volume of books means that there is a chance of finding something appropriate (the training material for other languages isn't much better). Furthermore, there is more C++ experience in New Zealand than there is experience with any other object-oriented language. Finally, there are books that will help the transition from novice to experienced C++ programmer (for example, [6] and [2]).

The best way to learn C++ is to learn C++

Because C++ is an object-oriented language, it is sometimes suggested that the best way to learn it is to first learn some other "purer" object-oriented language. However, despite their similarities, the mindsets required to use other object-oriented languages are sufficiently different that the benefit of learning another language does not justify the cost. Because C++ is based on C, it is tempting to think the best way to learn it is to learn C first. However C and C++ have quite different mindsets and so the only benefit in learning C would be in learning the syntax (which could also be taught as C++ syntax). This isn't to say there is no benefit from learning another language before learning C++. C++ is a sufficiently complicated language that it can be less stressful to learn the basic principles of programming in a simpler language (for example, Pascal or a subset of C).

Given the similarity between C and C++, anyone who knows C can pick up the main differences fairly quickly (just by reading this paper for example). People who don't know C will take a bit longer, but if they can already program then it is "just" a matter of learning a new language. However there is quite a difference between knowing the syntax of a language, and being an effective programmer with that language. This is even more true of object-oriented programming languages.

Making effective use of the concepts of object technology requires a completely different mind-set than traditional programming techniques. Estimates for learning to be a good C++ programmer (that is, with the object-oriented mindset) vary between 3 and 18 months.

The hardest part of learning C++ is learning how to identify the classes that are needed and how objects in the classes interact to solve the problem. If the classes have already been identified, their implementation can be tackled by a novice C++ programmer. This means that an effective way to train people is to use "mentors", people who already have experience using the technology, and can pass the experience on to novices. In this way, people new to C++ can still be productive, and at the same time can acquire the experience needed to become effective object-oriented programmers.

This does not mean that novices should receive no training other than what they learn from their mentors. It is a bad idea to just train your best programmers with the expectation that they will then do all the training — your best programmers will end up spending all their time teaching rather than programming. Everyone must have some training in the basics. The novices can then better appreciate what their mentors are doing, meaning they can learn by watching the experienced people do their jobs.

There are costs associated with adopting object technology

There are two main costs associated with adopting object technology: run-time performance and start-up costs[1].

The run-time costs are due such things as the dynamic lookup required for polymorphism and the fact that object-oriented design tends to require a large number of function calls. A language that does static type-checking (such as C++) can reduce the dynamic lookups to just the necessary ones. A sophisticated compiler can mitigate the cost of a large number of function calls.

The start-up costs involve training of personnel, acquiring the necessary tools, and building the right culture within the organization.

The training of personnel is the most time-consuming and thus the most expensive. However it must be done properly if the benefits of object-oriented programming are to be realized. It is not enough to just hand out books (although reference material must be available), it is not enough to just put people on a 3-day course (but it does provide a kick-start), and it is not enough to just train the best people and expect them to teach the others (better to use them as mentors).

The tools required depends on the nature of the organization. They usually consist of reference materials, CASE tools, and object-oriented libraries.

The right organizational culture requires a commitment by the whole organization to adopt the new technology. This can be difficult during the transition period as old systems still have to be maintained. There will also be a desire to avoid completely throwing out the old systems, which requires interfacing them with the new way of doing things. There will also be a period of time while the new systems are developed. Finally, new habits have to be adopted to get the most benefits of object technology, particularly in the creation of reusable code.

There are benefits associated with adopting object technology

There are several benefits to adopting object technology over other software development techniques. Object-oriented programming provides a better way to translate our familiarity with the real world into a coherent software development process. This means that we can usually build applications faster, the resulting applications tend to better meet the customers' requirements, and if they don't, the cost of maintaining the applications is more predictable. Object-oriented programming also encourages the creation of reusable code. While the initial code may cost more to develop, these costs can be recovered in reduced development time and maintenance costs in future applications.

Getting the benefits of object technology requires good management

Just using an object-oriented language like C++ will not yield the benefits of object technology. Personnel have to be trained to look for, recognize, and exploit the analogies most appropriate to the problem at hand, and they must be given the opportunity to do so. This is what a manager must know about C++.

References

[1] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, second edition, 1994.

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

[3] Richard P. Feynman. *QED: The strange theory of light and matter.* Princeton University Press, 1985.

[4] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[5] Andrew Koenig and Bjarne Stroustrup. C++: As close to C as possible — but no closer. *The C++ Report*, July 1989.

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

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

[8] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.