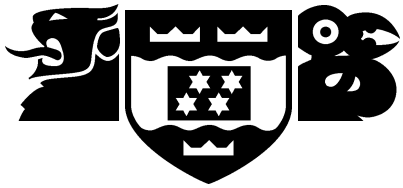
VICTORIA UNIVERSITY OF WELLINGTON



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

Understanding Code Reusability: Experience with C and C++

Peter Andreae, Robert Biddle, Ewan Tempero

Technical Report CS-TR-93/12
December 1993

Abstract

Despite nearly three decades of study in the area of code reusability, programmers are *still* writing code that they will not be able to use again. In this paper, we examine a set of C and C++ programming language features usually considered to be relevant to code reuse: functions, defined types, macros, composition, generics, overloaded functions and operators, and polymorphism. Our goal is to determine exactly how each of these features should be used to produce more reusable code, and so develop an understanding of the nature of reusability. We also discuss common misunderstandings of some of these programming language features and, in doing so, provide some practical advice for more effective programming.

Publishing Information

This report supersedes report CS-TR-93/6.

1 Introduction

Computer programming is a creative process, and necessarily involves invention; too often it involves re-invention. Some re-invention is inevitable, as it is impossible to know about all programs ever written. Of course, we would like to be knowledgeable about relevant and important programming to avoid "re-inventing the wheel". However, even in situations where we know all about code already written, where that code is written in the same language, and we have actually found a piece of code that we would like to reuse, we still experience a kind of "deja-vu". We know that we have written this code before, but are unable to escape writing it again because the code does not quite work for the new situation. Writing code that can be easily reused in a wide variety of situations is difficult.

It is important to distinguish between two distinct issues: the process of how to reuse code, and the process of how to write more reusable code. Existing work in this area has generally been on code reuse, [5] and work that does address reusable code is often more to do with design, [7] or does not actually talk about reusable *code* [3].

This paper is an attempt to address the problem of how to write reusable code. We look at how particular language features found in many modern (and some not so modern) should be used to make code more reusable. Our goal is to help professional programmers write reusable code, thus providing a starting point for addressing the more general problem of programming without re-invention.

We hope this compendium will provide practical advice for practical programming. To this end we consciously avoid theories of program design and idealized programming languages. Instead, our discussion will focus on language level issues and will use existing languages. The languages we have chosen are C [4] and C++ [8] because of their widespread use and availability, and because they represent two related points in the evolution of programming languages. However, knowledge of these languages is not necessary to understand our main points. In fact, many important aspects of both languages will not be mentioned as they do not directly contribute to the production of reusable code.

Writing reusable code is just one part of programming without re-inventing code, but we believe that it is a crucial issue: the best design techniques for encouraging reuse of existing code, and the most effective tools for finding relevant code fragments, are useless if the code fragments were not written to be reusable outside very specific situations.

1.1 Why Reusable Code?

The goal of creating reusable code is to avoid repeating the same or similar code fragments in different places by writing it once, in one place, and invoking it where it is needed. The advantages of writing reusable code go beyond avoiding unnecessary writing. Firstly, any time code is written, mistakes are made: avoiding the re-writing means avoiding more mistakes.

Secondly, even when no actual mistakes are involved, it is still so easy to allow inconsistencies between different versions. Inconsistencies can be irritating anyway, but as code is modified, minor inconsistencies can become much worse. When code must be modified, repeated code must be sought out and changed in the same way. Where code is specified in only one place but used in several others, it can be changed in only one place and have the change take effect in several others.

Thirdly, even if code is not being written, not being modified, but is simply being read and understood, code reuse is a significant advantage. Not only is there less code to read and understand, but there is an additional documentary effect: the connection between several places is made clear and certain, rather than similar code merely hinting at similar purpose.

1.2 Outline

Much of what we describe here, and what has appeared in the literature [1], comes from hard-won experience from using the language features we are studying. While these lessons are valuable in their own right, they are even more valuable if they can be used to give a better understanding of what contributes to better reusability. The next section is a first attempt at developing a framework for investigating reusability.

Section 3 looks at techniques for creating reusable code that are well established in many languages, as represented in the C programming language. While these techniques may be considered very basic, it is helpful to review them both as validation of our framework, and as the foundation for understanding more sophisticated techniques. Following that, section 4 discusses encapsulation. This is a concept that should be considered established, given that it was available in early languages such as CLU [6], but is not as widely used as it should be.

While this paper is mainly concerned with creating reusable code, it is useful to be aware of the code reuse mechanisms available to ensure there is no undesirable interaction between these mechanisms and the language features being used. This is discussed in section 5.

Section 6 presents our discussion on how to use certain language features to create more reusable code and finally, section 7 presents our conclusions.

2 Understanding Reusability

When talking about reusable code, it is convenient to make a distinction between the piece of code that is being reused — *the code fragment* — and the programs in which the code fragment is being reused — *the contexts*. Often (though not always) the code fragment is written at an earlier time than a context in which it is being reused. Because this paper is concerned with the reusability of code rather than the process of reusing code, we will focus on the writing of the code fragment, with the writing of contexts in the future, rather than focussing on the writing of the context.

When talking about a code fragment, it is common to distinguish the interface of the fragment from the implementation. We use *interface* to refer to all the ways in which the code fragment and the context can affect or depend on each other, and *implementation* to refer to all the details of the code fragment that are not part of the interface and are therefore independent of the context.

For example, consider a fragment containing a single function and a variable declared outside the function that is used to record state information between calls. The interface would include the function name, the types of its arguments, the type of value returned (if any), and the state variable (even if the designer *intended* the variable to be an implementation detail of the fragment). The interface would also include any variables or functions defined outside the fragment, but referred to from inside the fragment. We also consider the interface to include any additional constraints on the arguments of the function, such as a requirement that an array of characters must be in sorted order, since these constraints arise from dependencies between fragment and context that must hold for the fragment to behave correctly.

We refer to those parts of the interface that are declared in the code to be part of the interface (parameters of functions, public operations and variables of classes, *etc.*) as the *explicit interface*, and the parts of the interface that a compiler could determine are part of the interface, even though they are not declared to be, as the *implicit interface*. We refer to the other parts of the interface, which can only be stated in comments (if they are stated at all), as the *informal interface*.

The designer of the fragment may not intend later programmers to depend on all parts

of the interface. We find it useful to distinguish between the *contract* and *non-contract* parts of the interface: contract parts of the interface are what the fragment designer intends programmers to use to invoke the fragment and what programmers may rely on; non contract parts of the interface are what fragment designer intends to be implementation details that programmers should not exploit.

Note that "interface" is sometimes used in the narrow sense to refer to only the explicit, contract, parts of the interface; we find this restricted usage to be less helpful than our wider sense.

2.1 Dimensions of Reusability

The reusability of a code fragment is a concerned with how well it can be reused in different contexts. There is not a single measure of reusability, since there are a variety of different factors that determine how well a code fragment can be reused. Firstly, all the factors that contribute to the usability of code — efficiency, correctness, robustness, clarity, *etc.* — also contribute to reusability. We have also identified three additional factors that are particularly important for the reusability of code: Generality, Safety, and Ease of Use.

One cannot apply absolute scales to these factors, so they cannot be used to compute an absolute reusability measure. In many cases, one fragment may be more reusable by one factor but less reusable by another factor. These situations represent trade-offs where the programmer must make a decision.

2.1.1 Generality

A code fragment is more general to the degree that it allows a programmer to customise its behaviour to suit the particular context the programmer is working on. Generality contributes to reusability because it allows a code fragment to be reused in more different contexts than a less general code fragment. There are several ways of making a fragment more general. For example:

- Introduce implicit parameters by moving names in the fragment into the implicit interface. These names (for example, variables or function names) will then be defined, or at least accessible from, outside the fragment, enabling customisation by the context.

- Introduce explicit parameters by replacing fixed components of the fragment by formal parameters in the explicit interface. These fixed components may be constant data values, variable names, function names, types, collections of statements, or (in the case of macros) arbitrary pieces of code.

- Add alternatives to the code in the fragment, and introduce new parameters to select between the alternatives. Such parameters are often referred to as *flags*. This is a more constrained way of generalizing than replacing a piece of the code of the fragment by a variable to be given a value by the context since it merely allows the context to choose between a fixed set of alternatives rather than specifying an arbitrary piece of code.

- Reduce constraints on parameters, both explicit or implicit, to allow a greater range of possible values. Generalising the type of a parameter is one way of reducing the constraints. For example, a function that computes a numerical function might be generalized by extending the type of one of its parameters from integer to any kind of number. A fragment implementing a sequence abstraction might be generalized by increasing the range of possible types for the elements of the sequence.

2.1.2 Safety.

A code fragment is more safe to the degree that it could not contribute to errors in the final system. Safety contributes to reusability because it allows a programmer to rely on the fragment more than on a less safe fragment. More importantly, the simplest ways of obtaining more general fragments lead to unacceptably unsafe code. A running theme of this paper is to demonstrate how various language techniques are able to retain or increase safety at the same time as providing increased generality. The distinctions made above between interface *vs.* implementation, and contract *vs.* non-contract are important for evaluating safety.

There are three different kinds of safety that we are concerned with:

Integrity: A fragment has integrity to the degree that the fragment and the context cannot interfere with each other's correct operation. This means that the context and fragment should not be able to change either the data or the actions of the other except by using the contract part of the interface. Examples of fragments with reduced integrity include the following:

- A fragment that keeps some internal state in global variables that the context can modify directly, resulting in undefined behaviour of the fragment.
- A fragment that refers to a function name that it assumes has been defined in one way, but could be redefined by the context.

Independence: A fragment is independent to the degree that any program using the fragment will still work if either the fragment or the context is modified, as long as the contract interface of the fragment is unchanged. Independence requires that the fragment and the context may not depend on any part of each other except by using the contract interface. For example, the context should not be able to depend on the way a fragment has implemented its internal data structures, nor should the fragment be able to depend on data structures or functions in the context, other than those specified in the contract.

Achieving both integrity and independence require that the compiler be able to check for possible interference between the fragment and the context. This requires that as much as possible of the contract interface is made explicit, that the non-contract interface should be as small as possible, and that the compiler should enforce the interface.

Preventiveness: A fragment is preventive to the degree that it prevents a programmer from misusing the fragment in ways that are likely to lead to errors (in the context). For example, a fragment that implements a sequence abstraction using a list of `void*` would allow the context to construct sequences of mixed types. This is conducive to errors where the context inserts an item of one type, and then treats it as another type when removing it. A fragment that implemented the same sequence abstraction in a different way that required all the items in one sequence to be the same type would prevent this kind of error and would therefore be more preventive.

2.1.3 Ease of Use

The final factor concerns the subjective measure of how easily programmers are able to reuse the fragment in their program. For example, fragments that require the programmer to set up elaborate structures or require very complex invocations will generally be less easy to reuse than fragments with very straightforward interfaces. The consistency and completeness of the interface will be significant components, as will the design of the parameterisation of the fragment.

Although we consider this factor to be important for reusability, it is concerned with design more than the programming language techniques. Since this paper addresses programming language techniques to support the creation of reusable code, we will not address ease of use in this paper.

3 Established Techniques

Code reusability is not a new idea, and some techniques were developed early in the history of programming. In this section, we discuss three established techniques: functions, user defined data types, and macros. Our aim is to develop better a understanding of how these techniques achieve reusability, and how various aspects of these techniques affect the degree of reusability obtainable.

3.1 Functions

Probably the most common (and historically successful) technique for creating reusable executable code is to package up the code as a function or procedure, allowing it to be called from several places. This also saves the space of repeated code, and the name can play a helpful documentary role. Even more importantly, all the advantages of reusability apply, and the programs become easier to maintain, more understandable, and easier to develop further.

The most primitive way to reuse executable code is simply to use explicit flow control to route execution through the same statements more than once. This is so commonplace we don't usually think of it as code reuse, but the principle is similar: avoiding duplication by invoking the same piece of code from different contexts. Unlike an explicit branch, a function call also specifies a return address. The function definition and call facilities are really simply extra support that enhance both generality and safety, and so reusability.

Sometimes we wish a function to do exactly the same thing each time we call it. More usually, however, we want the actions of a function to depend on the context in some way. We can accomplish this by using expressions and selection statements involving data values from the context, and by calling other functions defined in the context. The function definition can access the relevant data and functions from the context in two ways: by referring directly to the names of variables or functions defined in any available context (implicit interface), or by having the data values or functions passed as parameters (explicit interface).

Referring directly to names defined in the context is both an advantage and a limitation. The advantage is that we need not bother specifying the data in the call to the function, and so the code is easier to use. The limitation is twofold. Firstly there is a limit on generality, because the reusability is limited to contexts where the data is correctly set up. Secondly there is a limit on safety, because direct use of names from the outside context will immediately compromise independence between the context and the function code, and potentially compromise integrity.

Parameters require both specification and special handling, but they allow us to customise the behaviour of a function without any particular context of external data being necessary. Accordingly, customisation with parameters generally affords wider reuse by widening the set of contexts from which the function may be called. Furthermore, if the function does not refer to any data in the context other than what is passed in the parameters, the language provides some integrity protection by ensuring that the function cannot have any side-effects on data in the context that is not passed in the parameters. This widens the set of contexts from which it is safe to call the function.

The advantage of passing data values explicitly in parameters is well understood and generally used, but fewer programmers seem to use *function* parameters to achieve the levels of reuse that they could. A general implementation strategy to make more general code is to make the function take function parameters that specify the action in a particular context. For example, a binary search can take a function parameter that specifies how to compare two elements of an array. This approach is limited in C because any functions passed as arguments must all take arguments of the same type. Generality can be increased by using functions with parameters specified to be the anonymous pointer `void*`, but this approach disables preventive safety. We will discuss more powerful ways of addressing this problem later.

Despite the advantages of the explicit interface of parameters, the implicit interface of names used from the outside context is still widely used. Use of global data, for example, can make function calls simpler. Moreover, calls from one function to another function also constitute an important part of the implicit interface, a part that is often under-emphasised. Sometimes a function appears reusable, but it depends on calling other functions that are not so easy reuse.

Early definitions of the C language made no provision for a calling context to explicitly declare the types of parameters of a function it was calling. Moreover, language implementations seldom detected inconsistencies between calls, and where functions were compiled separately, types of parameters were seldom checked either at link or run time. All this compromised the safety of function calls significantly. C++, and more recent definitions of C, allow explicit declaration of the types of parameters in a call (the *prototype* facility), and more recent implementations of the languages check function parameter types as part of the linking process. C and C++ both allow constructs where safety can not be guaranteed (for example, the anonymous pointer), but these constructs are explicit, and can be treated with the caution they deserve.

3.2 User Defined Types

The most common approach for creating reusable data declarations in C is to package up the declaration with a `typedef`, and then use the name as a type name in further declarations.

This generality requires the programmer to detail the declaration separate from any single place where it might be used, but allows use of the declaration in a variety of places with ease. This is useful because detailed type declarations do not have to be repeated when other data of the same type are declared. Moreover, by declaring data with the type name, no possibility for accidental difference is allowed, and the name can also play a helpful documentary role.

Data declared with a `typedef` type name are indistinguishable from data declared directly with the same specifications as the `typedef`. In both cases, the details of the type structure are directly accessible. The safety is limited, therefore, because the internal structure is completely accessible and cannot ensure integrity or independence.

However, types created with `typedef` can be specified in such a way that increases generality. Instead of specifying the new type using built-in types, it may be specified using other user defined types. In the example below, the `Queue` type is specified using the `Item` type, which must be defined by the context.

```
typedef
struct {
  Item data[MaxLength];
  int front, rear;
  int length;
} Queue;
```

As a result, instead of being limited to using the type for queues of, say, characters, the generality allows the declaration to be reused for queues of any other type as well. All that needs to be done is to specify the `Item` type differently, and the `Queue` type uses that definition. The example above is made even more general by the typical C use of a macro `MaxLength` to specify the size of the array.

This form of customisation is implicit, similar to use of a global variable in a function, but still very useful because the increased generality allows the type declaration to be used in more contexts. We shall return to this issue in the next section. The approach has a significant limitation: it can only be used once in the scope of the type definition. Two queues based upon different item types can never be used together.

3.3 Macros

We have discussed the established techniques for creating reusable executable code and reusable data declarations, but there is an established technique for creating reusable code of any kind: the macro. Essentially, a macro specifies a (parameterised) piece of text that can be inserted anywhere in a program as if copied by an editor, but the piece of text is written in just one place so that changes to the macro definition are automatically propagated to all the places that it is used.

Macro processing essentially deals with the lexical surface of a program, and so may be done by software quite separate from the programming language being used. In the case of C and C++, the macro pre-processor is usually a separate piece of software from the compiler, but programmers quickly rely on it always being available and applied before compilation.

Because macros involve only the surface structure of a program, their body and parameters can allow extreme generality, involving executable code, data declaration, or even incomplete code fragments. However, this flexibility means that there is no assurance of safety. Macros may destroy the integrity of the invoking context, and details of their expansion may result in unhelpful dependencies. Moreover they cannot be checked in even rudimentary ways independent of particular invocations. Macros are invoked at compile time, so syntactic checking of macro expansions is done, but even then, error diagnosis and recovery is very difficult. All these characteristics usually mean macros are used as a last resort to create reusable code.

However, this is an important role in the creation of reusable code. One aspect of this role is the fostering of language evolution. The C language, for example, did not originally have constructs for either constants or enumerated types, but similar facilities using macros were very widely used to create more reusable code. More recently, many C++ implementations lacked the template capability, and programmers made extensive use of macros instead. As it turns out, C and C++ have frequently evolved to provide facilities that allow reusable code to be created with more safety than macros can ensure, and so the reliance on macros for several aspects of programming has diminished.

4 Encapsulation

So far we have discussed how particular sequences of executable statements or data declarations may be made reusable, and also the very general capabilities of macros. However, it proves a very useful idea to consider *sets* of data and functions as a reusable unit. For such sets of data and functions to be made reusable yet *safe*, they must be regarded as *encapsulated*: bound together with strictly controlled access to the constituent elements.

4.1 Encapsulation of Data with Functions

There are very common situations that illustrate the desirability of encapsulation of data with functions. Most involve functions that use data to maintain some state from call to call. There are two safety issues involved: integrity and independence. Integrity is at risk from interference by the context, where the data maintained by functions may be changed to an invalid or inconsistent state. Independence is at risk from the context accessing the data directly, so depending on its exact representation. Without encapsulation, the data is an implicit part of the interface, and subject to both these threats.

The C programming language offers two facilities for encapsulating data with functions. For a function that uses external data only to provide "memory" from one call to the next, we may instead use data internal to the function but specified `static`. By default, data declared internal to a function is *automatic*, and is newly allocated on each call, and de-allocated on return. Internal static data is also visible only to the function, but has a lifetime that spans calls, so providing memory from one call to the next.

Some functions work together in management of a data structure, and must share visibility of the structure, so cannot use internal static data. For example, a single buffer might be accessed by functions for both storing and retrieving data. For such cases, external data is necessary, but it too may be specified `static`. C programs may consist of several files as separate compilation units. By default, all external data is visible across all compilation units. External *static* data, however, is only visible within its own compilation unit. By encapsulating external static data together in a unit with the functions that must use it, the functions may still be called, but the external data they require may not be interfered with or depended upon by the context.

By use of these techniques, functions and necessary data may be encapsulated together, allowing reuse, yet safe from the dangers of interference and dependence.

The following unit contains functions to manage a FIFO queue implemented by circular use of an array. The array must be external to any function because it must be accessible by both the enqueue and dequeue functions. So that other code outside the unit may neither interfere with nor depend upon its structure, the array (and supporting variables) are declared `static`.

```
static Item data[MaxLength];
static int front, rear;
static int length;

Init_Queue() {
  front = rear = 0;
  length = 0;
}

int EnQueue(Item x) {
  if (length == MaxLength)
    return 0;
  data[rear++] = x;
  rear %= MaxLength;
  length++;
  return 1;
}
```

```
int DeQueue(Item &x) {
  if (length == 0)
    return 0;

  x = data[front++];
  front %= MaxLength;
  --length;
  return 1;
}
```

Notice that because this approach encapsulates the actual data with the functions, it is limited to managing one instance of a structure. For example, the above unit only provides *one* queue. The generality of such fragments is therefore severely limited. There are *ad hoc* ways around this restriction (using tokens or anonymous pointers) but they are not very safe, and have other limitations.

4.2 Encapsulation of User Defined Data Types with Functions

As outlined earlier, the C `typedef` mechanism does enable the creation of reusable type declarations, but offers little more. In particular, the structure of data so declared is as accessible as if it had been separately detailed. As a result, any context using such data may directly access the internal structure.

In a situation where data is encapsulated in a compilation unit and never directly accessed from any context, the accessibility of type structure is irrelevant. However, some situations require that data be passed between an outside context and an encapsulated unit. An outside context must then know the type of the data, for code generation and type-checking, but if it can also access the internal structure then integrity and independence are compromised. Safety is then diminished, and reusability of the compilation unit is limited.

The most important addition made by C++ to C, the notion of *class*, allows us to overcome these problems. Like a `typedef`, a `class` specification allows a detailed data declaration to be given a name, and allows that name to be used as a type when declaring data elsewhere. However, a class allows the internal structure to be specified either `public` (as is always the case with a `typedef`), or `private`, in which case access is limited to a specified set of functions. These functions may also be specified `public`, in which case they form the explicit interface and may be called from any location to access the data structure, or specified `private` and so provide a service layer beneath the public interface.

In this way, the definition of a type may be encapsulated with an interface set of functions. The type may be used to declare variables anywhere, but access to the implementation details is restricted to a strict explicit interface. Encapsulation of types allows explicit reuse of defined types, without the dangers of any implicit interference or dependence. Moreover, because the encapsulation comes with the type, any data declared with the type has the safety advantages of the encapsulation.

Following is a class declaration for the kind of FIFO queue described earlier. Both the functions and the data type are encapsulated together in the class. Because a class creates a type, the name of this class, `Queue`, can be used to declare queues wherever necessary.

In contrast to the fragment in the previous section, this fragment allows the program to have many instances of `Queue`. The generality comes from having a user defined type; the safety comes from encapsulation. However, like the simple `typedef` example given earlier, instances of the queue in the same scope must all have the same kind of item, limiting the generality.

```
class Queue {
public:
  Queue() {
    front = rear = 0;
    length = 0;
  }

int EnQueue(Item x) {
  if (length == MaxLength)
    return 0;
  data[rear++] = x;
  rear %= MaxLength;
  length++;
  return 1;
}

int DeQueue(Item &x) {
  if (length == 0)
    return 0;
  x = data[front++];
  front %= MaxLength;
  --length;
  return 1;
}

private:
  Item data[MaxLength];
  int front, rear;
  int length;
};
```

5 Reusing Encapsulated Types

Once an encapsulated type has been created, there must be some way for the programmer to use it. An obvious way is to declare variables of the new type. It would also be of much greater use to be able to create new types by reusing the existing ones. There are two general ways in which this can be accomplished: composition and differential programming. Both methods produce new types without repeating code available in existing types and so provide all the benefits available through having reusable code. In fact, these methods apply equally well to non-encapsulated types, but the results can be unsafe.

5.1 Composition

Encapsulated types must have some way to represent the abstraction they describe. This representation may consist of a data structure described by types built-in to the language. However the representation of the new type may also use other encapsulated types, that is, it is *composed* of other encapsulated types.

For example, consider an application that manages customer records. Customers might have attributes such as account number name, address and current balance. If types appropriate for account numbers, names, addresses, or money amounts are already defined, a customer type can be more easily developed by composing these existing types together. An implementation is showed below. For brevity, only the class definitions are shown.

```
class AccountCustomer {
public:
  CreateAccountCustomer(String name, String address);
  void PrintAccountInfo();
  Money CurrentBalance();
  void Payment(Money m);
  Money PrintPrice(Article a);
private:
  AccountNumber accnum;
  String accname;
  String billing_address;
  Money balance;
};
```

This method has been taught in introductory programming courses as a way to create programs for so long that it is usually completely overlooked that it reuses code. Nevertheless, composition provides the greatest benefits for reusing code.

5.2 Differential Programming

A common situation is when the programmer requires a type that is similar to, but not exactly the same as one provided. As discussed in the introduction, copying the source for the type provided and making the necessary modifications is undesirable, so what is needed is some way to modify selected parts or add new parts to the existing type to create a new type. This is known as *differential programming*, because only the *difference* between what exists and what is needed has to be implemented.

A solution that is available in most languages is that of composition. The new type is created with the existing type as one attribute and access to the operations of the existing type is provided through the use of "wrapper code", that is, code that just passes on the arguments to the appropriate operation and returns any relevant results.

For example, suppose there are two kinds of customer in the management of customer records: account and mail-order. The mail-order customer has the same attributes and operations as account customers but will have in addition a delivery address and an operation for printing that address. Having implemented the account customer type, it should only be necessary to implement the details corresponding to the delivery address of a mail-order customer to get the mail-order type. Below is such an implementation using composition.

```
class MailOrderCustomer {
public:
  CreateMailOrderCustomer(String n, String a, String d)
  {
    ac.CreateAccountCustomer(n,a);
    // MailOrderCustomer initialisation
    ...
  }

  void PrintAccountInfo() { ac.PrintAccountInfo(); }

  Money CurrentBalance() { return ac.CurrentBalance(); }

  void Payment(Money m) { ac.Payment(m); }

  Money PrintPrice(Article a) {
    // Implementation specific to MailOrderCustomer omitted
    ...
  }

  void PrintDeliveryAddress() {
    // implementation omitted.
    ...
  }

private:
  AccountCustomer ac;
  String delivery_address;
}
```

The result reuses the code written for `AccountCustomer`. It also makes it quite clear that `MailOrderCustomer` is a "bigger" type than AccountCustomer.

C++ provides a more flexible form of differential programming through its *inheritance* mechanism. Inheritance allows new types to be specified as *derived* from existing types. This means that the new type has all the code of the existing code and it can be accessed directly, rather than through wrapper functions. The new type can also replace existing functions or add new ones. This is shown below:

```
class MailOrderCustomer : public AccountCustomer {
public:
  CreateMailOrderCustomer(String n, String a, String d) {
    CreateAccountCustomer(n,a); // Initialise the Account Customer part
    ...
  }

  Money PrintPrice(Article a) {
    // Implementation specific to MailOrderCustomer
   ...
  }

  void PrintMailingAddress() {
    // implementation omitted.
    ...
  }
private:
  String mailing_address;
}
```

This is clearly a much more concise implementation than the composition version. It also reduces the maintenance required since, in the composition version, the wrapper code has to be written and there can also be inconsistencies when a new operation is added to `AccountCustomer`; that operation is not automatically added to `MailOrderCustomer`.

6 Reusable Encapsulated Types

Just because a type is encapsulated does not necessarily mean that it will be usable for any situation other than the specific one it was created for. In this section, we describe techniques for using certain language features to make an encapsulated type more reusable by increasing its generality while maintaining safety. These techniques are most effective on a particular (but very large) variety of encapsulated type: container types.

6.1 Container Types

A common concept that appears in software is a structure that is a repository for one or more items. Being a repository means that the structure allows the items to be directly manipulated: in particular the items can be removed from the structure and either returned or replaced by other items. Such structures are commonly called *containers*, and their software representation *container types*. To provide the behaviour described, the interface to the container type must provide operations that allow its contents to be accessed or changed.

For example, consider a `point` type that consists of a pair of integers corresponding to an x-coordinate and a y-coordinate. This pair can be thought of as a container that holds

two integers. The interface to `point` must allow the programmer to inspect and modify the individual integers. Thus for the point value `(3,65)`, the programmer could determine that the x-coordinate is `3` or change the y-coordinate to `66`.

A container type is like a composite type that allows the programmer to directly access some attributes. In fact, it is not necessarily the case that the implementation of the container type directly represents its contents as attributes. For example, the `point` type may actually represent the pair of integers as a string, such as `''(3,65)''`. Nevertheless, the interface of `point` will allow the individual coordinates to be inspected or modified.

For an example of a non container type, consider a "statistics" type. Its purpose is to gather statistics on properties of another type, such as `observation`. Although instances of `observation` will be passed to the statistics type, it is likely that the statistic computed (e.g., the mean) will not allow those instances to be recovered.

Container types and Reusability

As discussed in section 2, to make a type more reusable the type must be made more general while at the same time maintaining safety. However this is not specific enough — it does not say what to make general or how to achieve generality safely.

Container types provide a large part of the solution. Most types created for an program either are container types by conscious design, or can be made into container types without sacrificing design goals. Also, it is clear how to make container types more general: increase the number of kinds of things they can hold.

Thus, not only do container types clearly identify a large well-understood category of types for creating reusable code, they also have quite specific guidelines for how their reusability can be increased. We believe the importance of the relationship between container types and reusable code has not been sufficiently realised.

Collections

The most familiar kind of container type is the *collection*. A collection is a container of homogeneous items, a common example being the *array*. Since many computer programs require the management of a group of relatively homogeneous pieces of data, collections are an often-implemented container type.

Many different kinds of collection types have been developed. The variations are due to different access disciplines, such as FIFO (queue) versus LIFO (stack), or maintaining different relationships between the items in the collection, such as a particular order (sequence) or the relationship between a key and a value (maps or lookup tables). Each variation has a well-understood abstract behaviour and appears in a large number of programs, and so is worthwhile making more reusable.

Despite this, it is surprisingly often the case that when an program requires a collection type, even if someone has already implemented that same type for another program, that type will be reimplemented. This is because the existing implementation is not sufficiently general for use in the new program.

In the remainder of this section, we will use collection types as our main focus for explaining how to make container types more general, and so more reusable.

6.2 Generic Types

Implementing collection types requires further support than just type encapsulation. A problem that often arises when trying to reuse an existing user defined collection type is that the type of the item it can contain is not what is wanted. This usually means that the type has

to be reimplemented. This is in contrast with the language-provided *array* collection type, for which the item type does not need to be specified until the time at which the array is being used, that is, instantiation time.

As discussed earlier, one solution to this problem to use a generic name for the type. When the code is needed, that name could be set to the appropriate type, using a mechanism such as C's `typedef`. This will work quite well in many situations. However it will not work if it is necessary to have several collections holding different types in the same program.

Alternatively, the item type for the implementation could be an anonymous type, such as C's `void*`. This allows items of any type to be put in the collection, so several collections can be used in the same program. However this method is unsafe because it also allows items of different types to be put in the same collection. For example, suppose that previous version of a program only put customer numbers in the queue but it now has been decided to put the customer type in the queue. In the process of making the change it would be very easy to miss one use of customer numbers in the code, and chaos would ensue. This form of generic type is unsafe. Specifically the problem is that of preventiveness: the programmer is not prevented from using the type incorrectly.

The best solution to this problem is to have the language support the type of the contained type being a *parameter* in the interface of the collection type. Such types are called *parameterised* types.

C++ provides parameterised types by way of *templates*. A template defines a family of types, where each member of the family differs by the value of the template's parameter(s). The parameter may be a type or a value. Below is the template version of the circular FIFO queue introduced earlier. Two parameters have been introduced (by just the addition of the `template` clause): one is the type of the items in the queue and the other is the maximum size of the queue.

```
template <class Item, int MaxLength>
class Queue {
public:
  Queue() {
    front = rear = 0;
    length = 0;
  }

int EnQueue(Item x) {
  if (length == MaxLength)
    return 0;

  data[rear++] = x;
  rear %= MaxLength;
  length++;
  return 1;
}

int DeQueue(Item &x) {
  if (length == 0)
    return 0;

  x = data[front++];
  front %= MaxLength;
  --length;
  return 1;
}

private:
  Item data[MaxLength];
  int front, rear;
  int length;
};
```

Below are the declaration and use of two different queues: one, a queue of at most 10 characters and the other, a queue of at most 100 customers.

```
Queue<char,10> ch_queue;
Queue<Customer,100> cust_queue;

ch_queue.Enqueue('a');
cust_queue.Dequeue(newcustomer);
```

Parameterised types are the result of generalising the fixed-type collections but doing so in a safe way — there is no way a non-customer can be enqueued in a queue of customers.

6.3 Overloading

Queues make no assumptions about the type of the items that they hold. This is not the case with all collection types. For example, consider a set which includes a test for membership as an operation. The following is an implementation of a (template) set that uses an array to store the items. The `membership` operation must scan through the array until it either finds the element or has checked every element in the set.

```
template <class Item> class Set {
public:
  int membership(Item element) {
  for (i = 0; i < length; i++)
    if (data[i] == element) return 1;
      return 0;
   }
  // Other operations omitted
private:
  Item data[MaxLength];
  int length;
}
```

The problem with this implementation is the set implementation is not as general as it first appears. Part of its implicit interface is the requirement that `Item` must allow the equality operator, `==`, to be applied to it. For non-built-in types, this is not usually the case. This is a subtle point but the result is code that is not as reusable as it might be.

One solution to this problem is to use a function, such as `equals`, instead of using the `==` operator:

```
  int membership(Item element) {
    for (i = 0; i < length; i++)
      if (equals(data[i],element)) return 1;
        return 0;
  }
```

This requires having many definitions of `equals`, one for any type that will appear in sets, and also requires having to write *more* code (namely `equals` for the built-in types). In C, this is a problem because it does not allow the same name to be used for more than one function.

C++ allows this, provided the functions can be distinguished by the types of their arguments. Since every version of `equals` will take two parameters of the type being compared, they will all have arguments of different types. This is referred to as *overloading*.

Ideally, the user-defined types that will be used with sets should be able to define the operator `==`. The advantage of this is that user-defined types can use a form of notation (namely operators) that have well-understood semantics. It also means that there is no need to write functions like `equals` for built-in types. C++ also provides this ability, referred to as *operator overloading*.

Now the template version of `Set` can be implemented on the assumption that the type of its items understand `==`. This increases the generality of the set implementation, without compromising safety.

6.4 Polymorphism

It is not uncommon to have a set of types that are similar, in that they have the same interface, but different in detail, in that the implementation of some operations is different. Thus, while they appear to have the same type (because their interfaces are the same) they behave differently at run-time because the actual code that gets executed is determined dynamically.

For example, now suppose there are three kinds of customer in the management of customer records: mail order, account, and business. The three kinds involve different discounts — the business customer gets a bulk discount, the account customer gets a frequent customer discount, and the mail order customer gets a different discount from account customers because the paperwork is different. This means that the `PrintPrice` function for the customer type must behave differently for the different kinds of customers if the following function is to behave as required.

```
void PrintReceipt(Customer *c, Article a) {
  c->PrintAccountInfo();
  a.PrintName();
  c->PrintPrice(a);
}
```

The typical implementation technique is to include a `kind` attribute for customers. The various operations, such as `PrintPrice`, must then examine this attribute to determine the appropriate behaviour. The code typically takes on the following structure:

```
void PrintPrice(Article a) {
  switch (a.kind) {
    case business:
      ...
      break;
    case account:
      ...
      break;
    case mail_order:
      ...
      break;
  }
}
```

Similar structures might appear in other customer operations. As the management required for the customer type becomes more sophisticated, these switch structures become more unwieldy. Techniques such as decomposition into sub-operations (for example, operations such as `ComputeBusinessPrice`) improve the situation but do not help with the switch code itself. The problem becomes most apparent when a new kind of customer gets added (e.g., employee, which gets staff discounts). Now the implementor must add the employee case to every place that checks the kind of customer, a process well-known to be error-prone.

This problem can be avoided if the different kinds of customers are implemented as separate types, so that all information relating the implementation of one kind of customer is kept in the same place. This will reduce the problems relating to maintenance but now the question is how to avoid writing multiple versions (one for each customer kind) of `PrintReceipt`. The problem is that now each kind of customer is a different type, and C++ requires that

the types of values passed to a function agree with the types of the formal parameters of that function.

The key observation here is that the three different kinds of customer are all specialised versions of more general "customer", one that only supplies simple `PrintAccountInfo` or `PrintPrice` operations. The different kinds of customers are related by the *is-a* relationship: a business customer is a customer, an account customer is also a customer, and a mail-order customer is an account customer.

If a type "B" is a type "A" then values of type B may be supplied wherever values of type A are expected, and furthermore, if values of type B are supplied then B's semantics will be used, not A's. This property is *polymorphism*. If a programming language has support for polymorphism, then functions such as `PrintReceipt` may be made more general, because they will now accept values of more types as their arguments. C++ provides polymorphism through the use of `virtual` member functions and the inheritance hierarchy.

Notice that polymorphism is only useful if there is some doubt as to the exact type of the value being manipulated. The only time such doubt can arise is when the value has just been removed from a container. This indicates a strong relationship between containers and polymorphism.

To continue the example, both calls to `PrintReceipt` below are now legal and, in each case, the appropriate version of `PrintReceipt` will be called.

```
AccountCustomer* ac;
BusinessCustomer* bc;
Article purchase;
// ...
PrintReceipt(ac, purchase);
PrintReceipt(bc, purchase);
```

This increase in generality has been achieved without any loss of safety. The compiler will only allow calls to `PrintReceipt` if the first argument is-a (either directly or indirectly) customer, and because it is a customer it must have the operations `PrintAccountInfo` and `PrintPrice`.

7 Conclusion

In this paper, we have been concerned with creating more reusable code. We have explained known successful techniques and investigated why they are successful. In an attempt to explain the success, we describe a framework for examining code reusability.

An important factor in increasing code reusability is generalising it, whether by creating some implicit form of customisation, introducing parameters, or reducing constraints on the parameters. But just increasing generality does not guarantee greater reusability: the generality must not be at the expense of safety. The important kinds of safety are integrity, independence, and preventiveness.

The most common technique for creating reusable code is the function. Functions can be further generalised by adding more parameters without sacrificing safety. On the other hand, macros, although providing a flexible form of generalisation, do not necessarily create reusable code because of their lack of safety.

A widely discussed, but perhaps insufficiently widely used, technique is encapsulation. Encapsulated types are more reusable due to the safety they provide. Two methods for creating new types by reusing existing types are composition and differential programming. The first is widely available although seldom discussed as a way to reuse code. The second

is wildly popular (indeed, over-sold) as a method to reuse code. However, we believe its limitations have not been sufficiently investigated.

Container types seem to offer the category most likely to be productive when improving the reusability of code. Such types are everywhere (particularly collection types) and it is clear how to make them more general: increase the number of kinds of items they can hold. We discussed three techniques for generalising container types: parameterised types, overloading, and polymorphism. None of these techniques compromise safety. We note that these techniques are considered by some to be different forms of polymorphism [2] and speculate that a better understanding of polymorphism may prove productive.

The ultimate goal is improving software productivity. The potential of code reuse has been examined for several decades, but with only small gains. We believe that part of the reason for this is that code is not being written for reuse. Furthermore, we believe that language support is a necessary component of successful reusable code. To fully evaluate existing or proposed language features, a better understanding of the principles of code reusability is essential.

References

[1] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems.* Benjamin/Cummings, 1987.

[2] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[3] T. Capers Jones. Reusability in programming: A survey of the state of the art. *IEEE Transactions on Software Engineering*, SE-10(5):488–494, September 1984.

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

[5] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[6] Barbara Liskov. A history of CLU. *ACM SIGPLAN Notices*, 28(3):133–148, March 1993. The Proceedings for the Second ACM SIGPLAN History of Programming Languages Conference.

[7] B. Meyer. Reusability: The case for object-oriented design. *IEEE Software*, pages 50–64, March 1987.

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