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

How to Reuse Code, Really

A Guide to Creating Reusable Code with C++

Peter Andreae, Robert Biddle, Ewan Tempero

Technical Report CS-TR-93/6
September 1993

Abstract

Despite nearly three decades of study in the area of code reuse programmers are *still* writing code that they and others have written many times before. In this paper, we examine a set of programming language features that are generally accepted as necessary to providing effective reuse of software. The set of features includes macros, procedures and functions, inheritance, polymorphism, overloaded functions and operators, and templates. Our goal is to determine exactly how each of these features benefit the production of code. This will enable us to extract general principles that can be used as guidelines for developing software more suited to reuse. We also discuss common misunderstandings of some of these programming language features and, in doing so, provide some practical hints on how to effectively use these features.

Publishing Information

This paper appeared in the Proceedings of the Thirteenth New Zealand Computer Society Conference, Auckland, New Zealand, August 18-20 1993. Pages 236–260.

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 (whether from another part of the same program, another program, or from some library), we still experience "deja-vu" — knowing 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 context. Writing code that can be safely reused in a wide variety of contexts is difficult.

This paper is an attempt to address the problem of writing reusable code. It outlines a coherent selection of techniques for writing code so that it can be reused in many places. Most of these techniques reduce to the appropriate use of particular language features found in many modern (and some not so modern) programming languages.

We intend this compendium to 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 and C++ 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 reinventing 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 to reuse are useless if the code fragments were not written to be reusable outside very specific contexts. The goal of the paper is to help professional programmers write reusable code, and thus providing a starting point for solving the more general problem of programming without re-invention.

1.1 Why Reuse Code?

The goal of reusing code is to avoid repeating the same or similar code fragments in different places by writing it once, in one place, and invoking it from several different contexts. By "context", we include both different places in a program (different lexical contexts with different data elements), different times during the execution of a program (with different values in the same data elements) and different programs.

The advantages of writing reusable code go beyond avoiding unnecessary writing. Firstly, any time code is written, mistakes are made: avoiding the writing means avoiding more mistakes. Secondly, even if no actual mistakes are made in re-writing, 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 each 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 then similar code merely hinting at similar purpose.

Overall, code reuse allows greater leverage in programming, and promotes program integrity; in turn, this allows easier maintenance and results in more understandable programs.

There are several dimensions along which techniques for reusing code fragments vary:

- The kind of code fragment we want to reuse.
  It may be a fragment of executable code specifying actions to be performed, a fragment of declaration code specifying data structures, or a combination specifying data structures and actions associated with those data structures.

- The way the fragment should be parameterized.
  If the fragment should have exactly the same effect whenever it is invoked (for example, signaling an immediate fatal error), then the fragment does not need to depend on the context at all. Most often, however, the fragment must have a slightly different effect when invoked from different contexts. The different techniques use different ways of making the effect of the fragment depend on the context. The different techniques vary in how easy it is to invoke the fragment — some methods place tighter constraints on the contexts from which the fragment may be invoked.

- The kind of support for modularity provided by the programming language.
  The general principles underlying good modularity require that any kind of unit (such as a code fragment that will be reused in several contexts) should have a well defined "interface" specifying how information can be passed between the unit and the surrounding context, and the program should be prevented from violating that interface by disallowing either the context or the unit to affect one another in any other way (for example, for the unit to affect the contents of data structures in the context that it is not explicitly permitted to). A programming language will provide different levels of support for specifying interfaces and enforcing "integrity constraints" for different reuse techniques.

In the sections that follow, we discuss these issues as they relate to a number of code reuse strategies, beginning with very small sections of code, and continuing to those potentially much larger. We start by looking at reuse of executable code and data declarations separately, then bring the two together while discussing the advantages of encapsulation, and proceed to elaborate various ways of getting the maximum reusability from encapsulated data types.

2 Executable Statements

2.1 Repetitive Structures

The most primitive way to reuse executable code is simply to use flow control to route execution through the same statements more than once. This may be done as part of a constrained repetitive structure, such as the `while` , `do`, or `for` constructs, or done in an unconstrained way with statement labeling and the `goto` statement.

This is so commonplace we don't usually think of it as code reuse, but the principle is the same: avoiding duplication by invoking the same piece of code in different contexts. The effectiveness of the reuse, however, is obvious if one considers unwrapping the loop code into multiple copies of the code fragment.

While reusing code in this way is fundamental, we usually don't want the code to do *exactly* the same thing each time. However, each use has the potential to do different things because the lexical context of the code fragment may include data elements whose values may be different on each invocation of the fragment. If the code refers to these data elements, then different values may yield different actions, either implicitly (for example, with array subscripting) or more explicitly with selection statements (`if`, `switch`) inside the code itself.

In the following code fragment, we use the C `for` structure to repeat some code, but because the value of the variable `i` changes, each repetition is distinct.

```
for (i = 0; i < num_accounts; i++) {
  total += accounts[i];
  if (accounts[i] < MIN_VALUE)
    too_small++;
}
```

This dependence on the names of variables in the lexical context is also a limitation to this technique: the code fragment can only be reused in another lexical context that has the same variables and must be copied to another context using an editor — a dangerous method of reuse that can introduce bugs and does not support later modification well. Furthermore, the language does not, and cannot, provide any support for preventing unwanted side-effects, either of the context on the loop code or the loop code on the context, since such side-effects are how the method works.

2.2 Procedures

The most common technique for reusing executable code is to package up the code as a procedure or function, and then call it from several places. This requires us to detail the code separately from any single place where it might be used and give it a name, but allows us to use the code from a variety of places with ease by merely using the name. This saves the space of repeated code, and the name can also play a helpful documentary role in suggesting the overall purpose of the code. Even more importantly, all the advantages of integrity apply, and the programs become easier to maintain, more understandable, and easier to develop further.

Sometimes we do wish a function to do exactly the same thing each time we call it, but that is rare. More usually, 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 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 the context, or by having the data or functions passed as explicit parameters.

Referring directly to names in the context has the same advantage and the same limitation for functions as with code within loops: we don't have to bother specifying or specially handling the data in the call to the function but we are limited to contexts where the data is correctly set up, and the language cannot provide any integrity protection.

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.

In the following fragment, code to sum the elements of an array has been made a function that takes the array as an argument, and returns the sum. The function can then be called from different locations to sum different arrays.

```
int sum_accounts(int num_accounts, int accounts[]) {
  int i;
  int total;
  for (i = 0; i < num_accounts; i++)
    if (accounts[i] > 0)
      total += accounts[i];
  return total;
}

void account_analysis(void) {
  //...
  s_total = sum_accounts(south_number, south_accounts);

  n_total = sum_accounts(north_number, north_accounts);
  // ...
}
```

The advantage of passing data 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. Functions that implement many complex algorithms (for example, binary search, graph algorithms, bisection ...) must perform a slightly different action depending on the kind of element in the main data structure (array, graph, or whatever). The kind of element will vary from context to context, but most of the action of the algorithm is identical in all contexts.

A clean way of implementing a function for this kind of algorithm that can be used in many contexts is to make the function take function parameters that specify the action in any 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. We will discuss more powerful ways of addressing this problem later.

3 Data Declarations

3.1 Repetitive Structures

The most primitive way to reuse data specification code is simply to use types derived from a single declaration. This may be done by in a constrained way by declaring an array, or in an unconstrained way by declaring a pointer.

This is so commonplace we don't usually think of it as code reuse, but the principle is the same: avoiding duplication. The array and the pointer allow us structures involving the same underlying data type.

In the following code, the `struct` is detailed once, but two variables and an array are declared with the same specification.

```
struct {
 int number_accounts;
 int accounts[MAX_ACCOUNTS];
} south_accounts, north_accounts, agencies[MAX_AGENCIES];
```

The limitations of this simple form of data specification are severe: different forms of the same structure must be declared at the same time, and as part of the same declaration.

Unlike the situation with executable code, we usually *do* want the data declaration to be exactly the same each time. There are some provisions for data structures to be similar, but not exactly the same, by using the `union` structure or the anonymous (`void *`) pointer. Also, because of the way arrays are passed as parameters in C, it is possible for functions to regard all arrays of the same type as compatible, though of different size. However, all these are very limited kinds of flexibility.

3.2 Defined Types

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

This requires us to detail the declaration separate from any single place where it might be used, but allows us to use the declaration in a variety of places with ease.

As shown below, defined types can include structures and arrays, and allow the detailed declaration to be named once, and then used to declare data wherever necessary.

```
typedef
struct {
 int number_accounts;
 int accounts[MAX_ACCOUNTS];
} Portfolio;

Portfolio Wellington_Investments;
Portfolio Auckland_Liabilities;


typedef char Address[CHARS_PER_LINE][LINES];

Address destination;
Address return_address;
```

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 in suggesting the overall purpose of the data structure.

Type names created with `typedef` have no particular flexibility or ability to be customised. In fact, 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.

4 Encapsulation

So far we have discussed how particular sequences of executable statements or data declarations may be reused. However, it proves a very useful idea to consider *sets* of data and functions as a suitable unit for reuse. The key idea is that such sets of data and functions must be regarded as bound together, *encapsulated*, with strict controls about access to the constituent elements.

4.1 Encapsulation and Procedures

Functions allow detailed code to be represented by abstract names that can be used to access the detailed code from many locations. However, the explicit access by name at particular locations may obscure the fact that functions may relate to their environment *implicitly* as well. There are two aspects to such implicit relationships, and both have significant consequences detrimental to code reuse.

Firstly, the detailed code of functions is allowed to, and may depend upon, access to the external environment. Functions may call other functions and use external data. This means that it is not enough to acquire the function, and access it by name. Statically, some other named functions, data, or types may also need to be available. Dynamically, some data may need to be in some appropriate state. In C, many of these requirements may not be explicitly stated in a way that can be easily seen: access to external data may be implicit. This is a weakness in the language, and while comments clearly stating access to the external environment are useful, they cannot be relied upon.

Secondly, some implementation details of functions are easily accessible from their external environment. For example, where functions depend on external data, other functions may also inspect and manipulate the same data. This means that the necessary state of external data relied upon for the correct behaviour of functions may be manipulated by other code. This manipulation may interfere with the necessary state relied upon for the correct behaviour of some functions. Even if it doesn't do that, it introduces a dependency into the code — no longer can the function be reused solely on its own specifications.

The C programming language offers several strategies for coping with these problems. For functions that only use external data 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 so must share visibility of the structure and cannot use internal static data. For example, a single buffer might be accessed by functions to store and retrieve 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 grouping external static data together in a file 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 use of these techniques, functions and necessary data may be encapsulated together, allowing explicit reuse without the dangers of any implicit dependencies.

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 char data[MaxLength];
static int front, rear;
static int length;

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

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

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

int DeQueue(char &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. There are *ad hoc* ways around this restriction (using tokens or anonymous pointers) but they have other limitations.

4.2 Encapsulation and Data Types

Like functions, defined types also allow code fragments to be represented by abstract names that can be used to access the code from many locations. However also like functions, the explicit access by name at precise locations may obscure the fact that defined types may also relate to their environment implicitly as well. Again, there are two aspects to such implicit relationships, and again both have significant consequences detrimental to code reuse.

Firstly, the detailed code of defined types allows and may require knowledge of their external environment: types may be built of other types and constants. Some other named constants or data types may also need to be available.

Secondly, implementation details of defined types are easily accessible from their external environment. Where defined types are built upon other types, other code may make use of these subordinate types. This may interfere with the necessary state of the data of a defined type. Again, this introduces dependencies into the code — no longer can the defined type be reused solely on its own specifications.

The most important addition to C made by 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 may be called from any location to access the data structure forming a strict interface, 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 interface. This means that types may be detailed and given a name, and that name may be used to declare data elsewhere — but use of such data is restricted to a certain interface. As a result, use of such data can rely on the integrity of the data being guaranteed by the encapsulation, so allowing safe reuse free from possible interference. From the *other* perspective, definition of such data types can rely on use of the type not depending on details of the implementation, so the implementation may be changed as desirable without invalidating any use of the data type. Moreover, this allows any use of the data type to benefit from improvements to the implementation of the data type without need for any change in the code that uses the data type.

In this way, 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 advantages of the encapsulation.

Following is a class declaration for the kind of FIFO queue described earlier. Both the functions and the data 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. Furthermore, it allows the program to have many instances of `Queue`.

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

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

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

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

5 Reusing Encapsulated Types

Encapsulated data types are more reusable because of the guarantees that they provide: nothing from the external environment can affect the implementation details of such a type except through a specified set of operations (the interface), and the implementation details depend on nothing in the external environment.

As we have already seen, the reusability of code depends on the dependencies that must be met to use it and the degree to which it can be customised. Encapsulated data types reduce the dependencies to just the interface. This means that the reusability of an encapsulated type depends on the customisability of the interface.

In this section, we will discuss the important issues involved in customizing the interface of an encapsulated type. We will discuss language features specifically designed to make interfaces of types more flexible (composition, parameterized types, derivation) and features that support the implementation of these types (overloading and polymorphism). We will also discuss an important class of encapsulated types, namely container types.

5.1 Composite Types

The ability to define new encapsulated data types allows a programmer to name and specify the interface and implementation of a new type in one place, and use that name and specification for data in many other places. The new type can be used directly to declare data, and the functions and operations of the type may then be used to manipulate the data in accordance with the specified interface.

For data of a defined type, the interface consists of the specifications of the public member functions that alone may manipulate the implementation. The implementation consists of the private member data, and the internal workings of the member functions which manipulate the member data.

A defined type may have private member data of the types built-in to the language, but it may also have member data that is itself a defined type. In this way, new *composite* types may be defined by reusing other defined types. The implementation of the new type will use the interfaces to the other types.

This is a method for easily creating new types by combining existing types. A common situation for applying the method concerns a type being developed that has a number of properties or attributes. The new type will then be composed of types appropriate to each of the attributes. For example, for an application that manages Customer records, the Customer type might include data corresponding to the name, address, and the current balance of the customer. If types appropriate for names, addresses, or money amounts are already defined, they can be reused, so making Customer a composite type. The operations of the Customer type will also reuse the operations of the component types to manipulate the attributes of the Customer type. Entire programs can be written in this way, with higher-level composite types defined in terms of lower-level types.

The strategy is quite straightforward, but enables two classic benefits of code reuse: less code to write, and easier code to maintain when changes are made.

5.2 Container Types

A common concept in writing software is a structure holding a set of items which allows those items to be "removed" and other items "inserted". Such structures are commonly called *containers*, and their software representation *container types*. The interface to a container type makes one or more of its attributes visible (the attributes that are the items in the container). That is, there are operations that allow those attributes to be accessed or changed.

For example, consider a pair of integers, such as (3,65), used to represent two-dimensional coordinates. This pair can be thought of as a container holding two integers. The corresponding type, `integer_pair`, has two attributes representing the two integers. The interface to `integer_pair` allows a client to access and modify both attributes. Thus the client could determine that the first integer has value 3 or change the coordinates to be (4,66).

A container type is like a composite type that allows clients to directly access the components. In fact, it is not necessarily the case that the container type is implemented with its attributes as components. For example, in the `integer pair`, the two integers may in fact be represented as a `string` such as "(3,65)". However the interface of the `integer pair` will allow the individual integers to be accessed or modified.

The most familiar kind of container type is the *collection*. A collection is a grouping of homogeneous items, a common example being the *array*. Since many computer applications 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 applications and so is valuable for reuse.

Despite this, it is usually the case that when an application requires a collection type, even if someone has already implemented that same type for another application, that type will be reimplemented. This is because the existing type cannot be customized for the new application.

5.3 Parameterised Types

Implementing container types, particularly collection types, requires further support than just type encapsulation. The main problem, one that any professional programmer who has had to implement a general container type is familiar with, is the type of the items in the container. In traditional programming languages, such as C, the type of the items in the user-defined container type, such as a `queue`, is fixed at implementation time. All instances of the queue described earlier can only contain characters because that is how the queue was implemented. This is in contrast with the language-provided *array* container type, for which the item type does not need to be specified until instantiation time, the time at which the array is being used.

Fixing the item type at implementation time reduces the reusability of the code. If a queue of customers is needed in another application, the existing queue of characters cannot be used, despite the fact that that implementation contains all the important details of any queue implementation. At best, the character queue implementation can be copied and appropriate substitutions from character to customer can be made. This is a tedious and error-prone process. It also results in two similar but different pieces of code that are likely to lead to maintenance headaches in the future, either because they become inconsistent, or because of the effort involved to ensure they *do not* become inconsistent.

One solution to this problem to use a generic name for the type, such as `itemtype`, instead of a particular type. Then, when the code is needed, `itemtype` can be set to the appropriate value via a `typedef`, for example,

```
typedef char itemtype;
```

This will work quite well in many situations. However it will not work if it is necessary to have two queues holding different types in the *same* application.

Alternatively, we could specify the item type to be an anonymous type, such as C's `void *`. This allows items of any type to be put in the queue, so two queues can be created for the application — one used for characters and the other used for customers. However this method sacrifices integrity checks that may be possible because it also allows putting both characters and customers in the *same* queue.

The best solution to this problem is for the language to allow the specification of container types without having to specify the item type. This is typically done by elevating the status of types so that *type variables* may be used in the definition of the interface. Then, when the container type with a particular type of item is needed, the item type is assigned to the type variable. Types whose interfaces include type variables are called *parameterised types*.

C++ provides parameterised types by way of *templates*. Below is the template version of the circular FIFO queue introduced earlier. As before, it specifies `data`, `front`, `rear`, and `length` as private data members and `Queue`, `Enqueue`, and `Dequeue` as public member functions. What is different is that it also specifies that `Item` and `MaxLength` are parameters to the `Queue` type; `Item`, which defines the type of the allowable items in the queue, may be any language or user-defined type, whereas `MaxLength`, the maximum number of items in the queue, must be an integer.

```
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;
};
```

The example below shows 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(42);
cust_queue.Dequeue(newcustomer);
```

Thus this implementation gives good control over those details of circular queues that clients most want to vary, namely the type of the item and the size. This was achieved by making these two properties of a queue be parameters in the interface, increasing the customisability of the `Queue` type. Increasing the customisability corresponds directly to being able to use the `Queue` in more contexts.

5.4 Overloading

One property of queues is that they make no assumptions about the type of the items that they hold. This is not the case with all container types. For example, consider *sets*. One of the operations for sets is a test for membership in the set. The following is an implementation of set of integers, which uses an array to hold the elements currently in the set. The `membership` operation must scan through the array until it either finds the element or has checked every element in the set.

```
class Set {
public:
  // Other operations omitted
  int membership(int element) {
     for (i = 0; i < length; i++)
       if (data[i] == element) return 1;
         return 0;
   }
private:
  // 'Set' implemented with an array.
  int data[MaxLength];
  int length;
}
```

If we also wanted a set of floats, or characters, or strings, or Customers, then we do not want to have to write the same piece of code again and again, so we make it a template.

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

The problem with this implementation is the requirement that `Item` must allow the operator `==` to be applied to it. For non-built-in types, this is not usually the case. This is an example of a dependency between the parameter of the parameterized type and the implementation of that type. Such dependencies are very subtle; if not dealt with adequately, they can lead to code that is not as reusable as it might be.

A 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;
  }
```

Apart from having to write *more* code (namely `equals` for the built-in types), it is really just changing the problem because now we need several different functions, all named `equals`. However 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*.

```
  int equals(int a, int b) {
    return a == b;
  }

  int equals(Customer a, Customer b) {
    // code for testing equality between customers
  }
```

Ideally, the user-defined types that will be used with sets should be able to define `==`. The major advantage of this is that user-defined types can use a form of notation (namely operators) that have well-understood semantics. C++ also provides this ability, referred to as *operator overloading*, as shown below:

```
int operator==(Customer a, Customer b) {
// implementation details
}
```

Now the template version of `Set` can be implemented on the assumption that the type of its items understand `==`, which, thanks to overloading in C++, is every type that it is sensible to have sets of. This essentially eliminates the dependency that was causing problems above.

5.5 Polymorphism

Templates (parameterized types) are a *static* form of customization. There is flexibility in the exact form the type takes at a particular place in the program, but the decision as to what that form is must be made at compile time. For some situations, this is too soon.

It is not uncommon to have a set of types that are similar but different in detail. The similarities are that they all have the same interface but the implementations of some of the operations may be different for different types. This means that while they are all effectively the same type (having the same interface), they behave differently at run-time. This means that the actual code that gets executed is specified dynamically.

For example, suppose there are three kinds of customer in the management of Customer records: business, account and mail order. The three kinds involve different discounts — the business customer gets a bulk 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.

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:

```
// A member function of customers.
// "kind" is a private data member of customers.
void PrintPrice(Article a) {
  switch (kind) {
    case business:
      ...
      break;
    case account:
      ...
      break;
    case mail_order:
      ...
      break;
  }
}
```

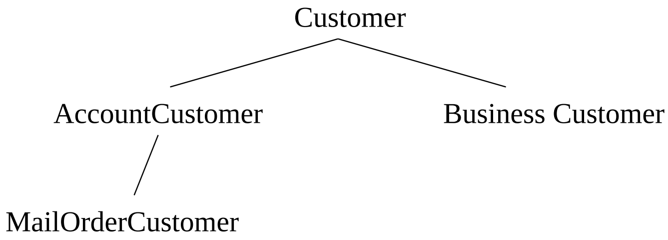
Similar structures might appear in other customer operations. As the management required for the customer type becomes more sophisticated, these switch structures become more unwieldy. Techniques such as decomposition into sub-operations (for example, `ComputeBusinessPrice`) improve the situation but 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.

The solution to this is to apply the principle of locality and create separate implementations of each kind of customer. This means that the information about one kind of customer, such as the *business* customer, is kept in one place. Any function that expects a customer argument, will accept all three implementations, but when an operation is called on any particular entry in the container (eg `PrintPrice`), the actual operation called will depend on the kind of customer. This is illustrated below:

```
void PrintReceipt(Customer *c, Article a) {
  c->PrintAccountInfo();  // Prints account number,
                          // name and address
  a.PrintName();
  c->PrintPrice(a);
}
```

There is now no need for an explicit `switch` construct; this is automatically handled by the language. Should a new kind of customer be created, the only change required by the application software is a new implementation of the customer type. This change will not affect *any* of the existing software in any way.

This notion of getting a different actual function being called at run time is what is usually called *polymorphism* (or more properly, inclusion polymorphism). This *dynamic* form of customising provides great flexibility in how code is (re)used. C++ achieves this by placing its types (classes) in a *hierarchy* and declaring the functions to be *virtual*. In this hierarchy, a child type may be used anywhere the parent type is expected. For example, suppose the types discussed in the example above are related as follows:



Then both calls to `PrintReceipt` below are legal.

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

Here, polymorphism has meant that the `PrintReceipt` function does not have a static dependency between the type of the formal parameter and the type of the actual parameter. As always, reducing dependencies results in more reusable code.

5.6 Differential Programming

We introduced parameterised types as a way to customise the interface of a container type. Now let us consider a form of customisation that is more sophisticated. In the previous section, we described how to take advantage of the fact that there might be more than one implementation of the customer type, but it is not clear how to manage these multiple implementations. The main issue is that while different implementations behave differently in some cases, they also share some behaviour. For example, all customers will have an account number and an account name. To reimplement those parts of each type is contrary to the goals of reusable code.

To make the example more concrete, suppose the account customer has the attributes *account number*, *account name*, *billing address*, and *current balance*, and operations for printing and updating those attributes. The mail-order customer has the same attributes and operations, but in addition will have a delivery address and an operation for printing that address for the package. 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.

None of the techniques we have discussed so far cover this case. What we want to do is take an existing type (for example the cash customer type) modify selected parts and/or add new parts to create a new type. This is known as *differential programming*, because only the *difference* between what exists and what is needed gets implemented. By managing the code this way, we both reduce the amount of code that has to be written and reduce the amount of maintenance the code requires, because changes to the shared code will be reflected in all implementations.

One way to achieve this is to use composition. Below is the account customer type as implemented in C++. For brevity, only the class definitions are shown. Note that this implementation is composed of other user-defined types, such as `Money` and `AccountNumber` and so is itself a composite type.

```
class AccountCustomer {
public:
  // Create a new account customer with name "n" and address "a"
  AccountCustomer(String n, String a);
  void PrintAccountInfo();
  Money CurrentBalance();
  void Payment(Money m);
  Money PrintPrice(Article a);
private:
  Money balance;
  AccountNumber accnum;
  String name;
  String billing_address;
}
```

Now here is the mail-order customer type implementation that uses the new account, `AccountCustomer`.

```
class MailOrderCustomer {
public:
  // Create a new account customer with name "n", billing address "a"
  // and mailing address "m".
  MailOrderCustomer(String n, String a, String m) :
     ac(n,a) // Initialise the Account Customer part
  {
    // 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 ommitted
    ...
  }

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

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

This clearly allows the reuse of the code written for `AccountCustomer`. It provides access to the `AccountCustomer` operations through the use of “wrapper code”, that is, code that just passes on the arguments to the appropriate `AccountCustomer` operation and returns any relevant results. While this does the job, the wrapper code does require some maintenance and it also makes it difficult to see the relationship between `AccountCustomer` and `MailOrderCustomer`. Recall from the previous section that there are situations where it would be convenient to be able to treat both these as different implementations of the same (more abstract) type. Using composition, this is not possible.

C++ provides a more flexible form of differential programming, namely its *inheritance* mechanism. The relationship between the two types `MailOrderCustomer` and `AccountCustomer` (namely, a `MailOrderCustomer` *"is-a"* `AccountCustomer`) can be stated directly by making `MailOrderCustomer` a *derived type* of `AccountCustomer`. This is shown below.

```
class MailOrderCustomer : public AccountCustomer {
public:
  // Create a new account customer with name "n", billing address
  // "a" and mailing address "m".
  MailOrderCustomer(String n, String a, String m):
    AccountCustomer(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 composite version and also has the advantage that it makes the relation between the two types explicit. This is, in fact, how C++ specifies the class hierarchy described in the previous section.

With differential programming, it is possible to customise the interface and implementation of an existing type; the interface can be changed by adding new operations and the implementation can be changed by replacing the existing implementation of an operation with a new implementation.

6 Macros

The previous techniques allow a programmer to write many kinds of code fragments that can be reused in different contexts. Some of the techniques allow considerable customisation so that the reused fragment has a different effect in different contexts, but the common code is written just once. However, there will be situations where the same piece of code is used in several places in a program (or in different programs) but it is not possible to fit it into any of the above frameworks because the kind of customisation required cannot be expressed.

For example, breadth first search arises in a number of different applications — traversing trees, searching graphs, some optimisation problems, and AI problem solving. The kind of space that is traversed varies from explicit tree data structures to implicit networks represented only by a function for calculating the neighbours of a node on the fly. However, the structure of the breadth first search is the same in all cases, and we would like to be able to reuse this code. Two pieces of code using breadth first search are given below: one prints the nodes in an explicit tree; the other searches for the shortest sequence of moves to get to a winning board position. Both use the generic queue interface described earlier. The shared code is obvious, but could not be captured satisfactorily in a function, even a generic function with function parameters.

```
Queue Q;
Q.EnQueue(root);
while (Q.DeQueue(node)) {
  node->print();
  for (i = 0; (i < node->num_children) ; i++) {
    neighbour  = node->child[i];
    Q.EnQueue(neighbour)
  }
}
```

```
Queue Q;
Q.EnQueue(Initial_Board(game_number));
while (Q.DeQueue(node))  {
  if (Winning_Board(node))
    return node;
  else {
    printf("Considering board:");
    board->print();
  }
  for (i = 0; i < Max_Moves; i++) {
    neighbour = Construct_Board(node, move[i]);
    Q.EnQueue(neighbour);
  }
}
```

One approach for reusing this code is to use an editor — copy the code from one place to another, then edit it to provide the appropriate customisation for the new context. This is generally better than rewriting the code since it is usually faster, and is slightly less likely to introduce bugs. However, it doesn't provide the advantages of writing the common code just once — updates or bug fixes in one context are not automatically transferred to the new place.

For maximum reuse, we would like to be able to write a definition of `BREADTH_SEARCH` in one place and reuse it in the following way by providing just the fragments that are different in the two cases:

```
BREADTH_SEARCH(root,
  node->child[i],
  node->num_children,
  node->print())
```

and

```
BREADTH_SEARCH(Initial_Board(game_number),
  Construct_Board(node, move[i]),
  Max_Moves,
  if (Winning_Board(node))
    return node;
  else {
    printf("Considering board:");
    board->print();
  }
)
```

We can achieve this using a 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.

```
#define BREADTH_SEARCH(start_node, neighbour_node, \
                       num_neighbours, action) \
{
  Queue Q;\
  int i;\
  Q.EnQueue(start_node);\
  while (Q.DeQueue(node)) {\
    action;\
      for (i = 0; i < num_neighbours; i++) {\
        neighbour = neighbour_node;\
        Q.EnQueue(neighbour);\
      }\
  }\
}
```

In this particular example, the code that is being reused is fairly small, not very complicated, and has few opportunities for introducing bugs. One advantage of the macro is in the clarity of the code — the reader can much more easily determine that this piece of code is doing a breadth first search rather than having to read the code carefully to determine this. For more complex patterns there are more opportunities for introducing bugs if the code is rewritten each time, and reusing code with a macro may decrease the likelihood of errors considerably.

The problem with reuse by macros is that they give no kind of encapsulation or protection, nor can they be checked independently of their invocation. However, macros are invoked at compile time, so that the compiler can check that the macro expansion is valid.

Macros should generally be thought of as a last resort for code reuse — if the language provides an alternative way of achieving the reuse, then use that; if that is not possible, then a carefully created macro may be better than cutting and pasting with an editor.

7 Conclusion

We began this discussion of reusable code by looking at small code fragments, and then progressed to larger units where the advantages of reuse become more significant. Even in the smallest ways of creating reusable code, some key ideas become clear:

- We identify code in one place so we can reuse it from other places;

- We maximise the ways to customise the code to make it more useful;
- We minimise the dependence between the code and the context in which it is used to preserve its reusability.

We initially discussed executable statements and data declarations separately, but because larger units of code involve both, we then discussed how both executable code and data can be encapsulated together. Encapsulated types, at first glance, seem to be the ultimate in reusability. They reduce the dependency between the implementation of a type and the external environment in which it is used to an explicit set of allowable operations: the interface.

However, as we showed, this is not the full story. There can be two further forms of dependency between the implementation and its environment that are not provided for by encapsulated types. The first is that the type may be some form of container whose implementation does not depend on the actual type of the items kept in that container. For this case we need parameterized types to remove that dependency. And even that is not enough. It is possible that the implementation of the parameterized type may require a specific behaviour (operation) of the item type. This dependency can be removed by allowing the overloading of names, particularly operator names, so that any item type that may be used with the container type can be given the needed behaviour. Finally, it may be the case that the actual function to be executed cannot be fixed at compile time. In this case, polymorphism provides a way to provide the necessary flexibility.

The features described above reduce the requirements that must be met before an encapsulated type can be used. However they do not address situations where the type as is not sufficient as provided. It must also be possible to customize existing types to create new types, without introducing new dependencies. Composition and differential programming give this ability.

We have described a set of tools and techniques with which a programmer can construct code that can be reused in a wide variety of contexts, thereby obtaining the benefits of code reuse. We intend to use this as a basis for developing a general model of the nature of reusable code.

Bibliography

The standard reference for C++ is Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, second edition, 1991. For a description of the design principles for C++ see Bjarne Stroustrup, A history of c++:1979-1991, in *Proceedings of the Second ACM SIGPLAN History of Programming Languages Conference*, pages 271–298, April 1993.

Below is a list of alternative views of software reuse.

Grady Booch. *Software Components with Ada.* Benjamin/Cummings, 1987.

Grady Booch. *Object oriented Design with applications.* Benjamin/Cummings, 1991.

Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

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

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

Rajendra K. Raj and Henry M. Levy. A compositional model for software reuse. *The Computer Journal*, 32(4):312–322, August 1989.