Generic Programming

Last updated 15 December 1998
Objectives and Meta-Outline

Part I: Fundamental Concepts

Part II: Algorithms on Algebraic Structures

Part III: Algorithms on Sequential Structures

Part IV: Data Structures and Containers

Part V: Programming Languages and Software Engineering

Appendices

Appendix B: Axioms and Key Principles

Appendix C: Summary of Code

Bibliography (content)

Send comments about this page to Jim Dehnert, Alex Stepanov, or. John Wilkinson.

Generic Programming: Objectives and Sketch

Last updated 16 March 1998

Objectives

This book is being derived from a class on generic programming first taught by Alex Stepanov at SGI in 1997. It is intended to be a practical guide and motivation for practicing programmers, in the form of the original class, based on extensive examples drawn from the C++ Standard Template Library. The approach is the progressive development of a variety of important generic examples, with attention to the underlying history, complexity, and performance.

Issues:

- *This is fundamentally a software engineering book. It should be quite appropriate for an advanced programming course. It might also be a reasonable text for classes on algorithms and data structures. Should that be an objective? It wouldn't be a traditional algorithms text, since we don't want to do rigorous complexity analysis.*

- *Do we want to discuss and/or develop NUMA iterators somewhere?*

Part I: Fundamental Concepts

Chapter: Preliminaries

The book's objectives, genesis, and contents. General motivation -- reusability and performance. A

simple motivating example, e.g. swap. Background definitions. C++ requirements? Conventions.

Chapter: Fundamentals

Interrelation of equality, assignment, and construction. The notion of regular types. The notion of parts. The taxonomy of objects and their parts. Swap algorithm. Generic optimizations on regular types. Ordering relations. Strict weak ordering. Total ordering. Min and max algorithms.

Chapter: Software Components

Taxonomy of software components. Function objects. Function object adapters. Binding and composition. Components traits. Categories and category dispatching.

Part II: Algorithms on Algebraic Structures

The material in this part is mathematical in nature, and may be skipped without significant effect on the remainder of the book.

Chapter: Arithmetic

Power. Russian peasant algorithm. Fibonacci numbers. Fiduccia's algorithm for linear recurrences. GCD. Euclid's algorithm. Stein's algorithm. Extended Euclid's algorithm. Continued fractions.

Chapter: Algebra and Number Theory

Algebraic structures: monoids, semigroups, groups, rings, fields. Finite groups: order of group, subgroup, element, and generators. Euler's theorem. Modular arithmetic. Prime numbers and primality testing. Miller-Rabin's algorithm.

Chapter: Encryption

The RSA encryption algorithm.

Part III: Algorithms on Sequential Structures

A focus of this part is how data access capabilities (iterators) affect the available algorithms and complexity for various problems, and how one can adapt the algorithms used to the capabilities available at compile-time instead of making more expensive choices at runtime.

Chapter: Iterators

Linear search algorithm (find). Copy algorithm. Iterator axioms and models. Iterator categories: input, output, forward, bidirectional, random access. Iterator traits and compile-time dispatch. Iterator adapters. Reverse iterator. Stride iterator.

Chapter: Permutations

What is a permutation? Classification: Index-based, value-based, and comparison-based. Cycle decompositions. The computational complexity of a permutation.

Chapter: Index- and Value-Based Permutation Algorithms

Reverse algorithms. Memory-adaptive algorithms. In-place rotation: Gries-Mills algorithm, 3-reverse algorithm, and GCD-cycles algorithm. Stability of permutations and pseudo-permutations. Remove and stable remove. Partition algorithms: Hoare partition, Lomuto partition, stable partition. Random shuffling and random selection. Lo-Ho algorithm. Reduction, parallel reduction, and binary-counter reduction. NUMA-iterators.

Chapter: Sorting And Related Algorithms

Binary Search. Lower and upper bound algorithms. Merging and set operations. In-place merging. Memory-adaptive merging. Stable sorting. Priority queues and partial sorting. Musser's introsort. Binomial forests.

Part IV: Data Structures and Containers

Chapter: Data Structure Principles

Data structures and their iterators. Container requirements.

Chapter: Sequential Data Structures

Sequences. Vectors. Lists. Dequeues.

Chapter: Associative Data Structures

Associative containers.

Part V: Programming Languages and Software Engineering

Compiler optimization issues. Why don't we use inheritance? Limitations of C++. Concepts as a linguistic entity. Compiling generic components. Science is a necessary foundation of engineering.

Send comments about this page to Jim Dehnert or Alex Stepanov.

Home 

Generic Programming

Part I: Fundamental Concepts

Chapter 1: Preliminaries

Last updated 26 March 1998

They came [to Plato's lecture on the Good] in the conviction that they would get some one or other of the things that the world calls good: riches, or health, or strength. But when they found that Plato's reasonings were of mathematics their disenchantment was complete.

(Aristoxenus, Harm. El. 2.30, trans. Ross)

Section 1.1: Introduction

Genesis and contents. Objectives and motivation: reusability and performance.

Section 1.2: A Motivating Example -- Swap

Section 1.3: Background Definitions

Section 1.4: C++ Requirements

The underlying algorithmic language we use requires only very simple constructs, namely if, while, do

while, function call, return, and block statements, along with the operations (functions) required by the algorithms we are implementing.

Develop the STL pair structure as an example.

Section 1.5: Conventions

There are a number of conventions we will follow throughout this book.

Parts of the book are mathematical or discuss analogies with mathematical concepts, but are not critical to understanding of the material. We set those parts in a smaller font to distinguish them.

The use of code examples, both good and bad, is critical to our exposition. We will distinguish incorrect code by setting it in red.

Implementation Queue

The following are points to be integrated somewhere:

- *Although this book is based on C++, we will often discuss the development of language features over time, using examples from previous languages. However, this is not a book about the history of programming languages, and we choose our example languages for their familiarity to a large number of programmers, and not necessarily because they represent the first or most important occurrence of the feature being discussed.*

- *There are two kinds of complexity propositions: requirements on the operations in the concepts and algorithmic guarantees of algorithms defined on the concept. (Of course, both of these have worst case and average case kinds.)*

Send comments about this page to Jim Dehnert or Alex Stepanov.


Generic Programming

Part I: Fundamental Concepts

Chapter 2: Fundamentals

Last updated 10 April 1998

---

§ 2.1 -- Introduction

Generic programming is based on dividing programs into components, for example key data structures and algorithms, which can be matched with other components satisfying a minimal set of assumptions. Its success depends upon minimizing the set of necessary assumptions, while still providing enough guarantees that the programmer of a generic component can make effective choices in its implementation.

This chapter discusses a minimal set of guarantees, and provides some illustrative examples. These guarantees involve the most basic operations of a programming language, and should seem quite intuitive to practicing programmers. However, they involve questions which we seldom think about explicitly, and require careful thought and understanding precisely because they will be assumed by programmers with no knowledge of the components satisfying them.

Many books contain a chapter on "fundamentals" or "background" which is review material for many readers, and can safely be ignored or skimmed if it looks familiar. That would be a mistake here. Although the subjects are superficially familiar, the details are important. Indeed, it might be useful to revisit this chapter after reading the later material which depends on it, since some of the implications will be clearer at that point.

Interfaces

Interfaces are a critical aspect of good software engineering. We are acutely aware of many of the interfaces we deal with, from the operations defined for the types we use, to the parameters required by functions we call, to the data and procedural interfaces exported by library packages we use.

Interfaces are contracts between the provider of a feature and a client. Naively, they are restrictions: A type interface provides some operations, but not others. We can call a function only with the supported parameter types. To use a library package, we must satisfy certain restrictions on program state when we call its routines. Less obviously, but much more important, the interface and its restrictions free the feature provider to provide a much more effective implementation because it need not be concerned with unspecified (and even unanticipated or unanticipatable) capabilities or situations. Similarly, it frees the client by excluding undesirable side effects from the feature provider.

There are many considerations in designing a good interface. An ideal interface should provide the functionality expected by the client, in an intuitive way, without sacrificing performance. It should also result in client code which is easy to understand for maintenance, which will often result from the other requirements. Often this ideal is difficult or impossible to achieve, when these requirements are in conflict, or when multiple clients have differing needs. In such cases, making the best tradeoff can be difficult, requiring excellent judgment or intuition. This leads many to believe that programming is as much art as science. Indeed, although experience provides principles which may be used for these tradeoffs, and we will suggest a number in this book, judgment remains important.

However, it is often possible to objectively identify good and bad interfaces. A clear understanding of which clients are important, of what their requirements are, and of what the implementation needs, can make many choices quite clear. We will see many such cases in the examples in this book, and the remainder of this chapter will discuss some of the most fundamental ones.

Programming *Concepts*

Generic programming's central premise is that we can decompose the programming task into orthogonal components, and write each component with no more assumptions about the others than are necessary. We will discover that certain sets of assumptions occur over and over. We will call these sets of assumptions *concepts*.

More concretely, a concept describes an interface between algorithms and types. It describes a *base* type, *affiliated* types, operations on these types, and properties of the types and operations. Concepts are not simply types. Whereas a type is a concrete specification of the representation of a set of values, along with the full set of operations on those values, a concept is an abstract specification which imposes a minimal set of restrictions on its base type, in terms of the affiliated types and operations, so that any type with affiliated types and operations satisfying those restrictions can be used with algorithms interfacing with that concept. This hierarchy is analogous to the biological concepts of an individual animal (value), its species, which is a specific description of animals with the same characteristics as the individual (type), and its genus, which is a more abstract description of a variety of species with similar characteristics (concept). Thus, we can say that a type (along with its affiliated types and operations) is an instance of a concept in the same sense that *dog* is an instance of *mammal*.

Although we define concepts in terms of a set of

types, operations, and their properties, it is often more convenient to specify a concept in terms of other concepts. Thus, we may create a concept from another by *refining* it (adding one or more affiliated types, operations, or properties), or by *abstracting* it (removing affiliated types, operations, or properties from its definition).

> *Concepts are analogous to theories in mathematical logic, including the fact that there are two quite different ways of specifying one. A theory in mathematical logic may be specified by giving the set of theorems that it satisfies; analogously, a concept may be specified by giving the set of programs which work correctly with it. Alternatively, a theory may be specified as the set of models which satisfy it, or a concept may be specified by giving the set of programming components (e.g. data structures) which implement it.*

Concepts provide us with the means of distinguishing generic components from concrete program components. Whereas a concrete component defines its interface in terms of types for its function parameters and results, or data structure members, a generic component defines its interface in terms of concepts, and may be instantiated by replacing the concepts with specific types and operations that satisfy the properties of the concept, allowing it to be used with any other components having interfaces satisfying the concept(s).

Abstract Type Interfaces

Beyond the obvious interfaces in programs, there are others that we don't usually think about. This is sometimes because they are so pervasive that we take them for granted, for instance the basic syntax and semantics of our programming language. Or it may be because they are so subtle or intuitive that it doesn't occur to us to think about them, such as the precise semantics of operations like assignment and equality testing.

The data types we define in our programs interact with all the other components we use. To achieve maximum benefit, our generic components must work as well with these user-defined types as with the built-in C++ data types. In fact, they should work the same way, since writing multiple versions of code is what we are trying to avoid in generic programming. That we can contemplate doing so is due to the fact that newer programming languages like C++ allow us not only to define new data types, but also to define all of the standard operations for those types. This is a powerful capability, but using it effectively requires that we be very clear about the interface requirements for these fundamental operations. This is necessary even though these are operations which we intuitively understand (or think we do) without thinking about them, e.g. assignment.

An important issue in matching the semantics of these operators to the operators of the language's built-in types arises from the specification of the standard operators on user-defined types as functions. The appropriate definition of an operator requires understanding not only what effects must occur, but also what the best parameter/result function interface is. Moreover, because generic program components consist largely of function definitions, our desire to match the behavior of user-defined types to that of built-in types also means that their parameter passing behavior should match. In C++, this means that our new types should work like built-in types when they are passed either by value, i.e. copied into the

formal parameter so that formal parameter modification does not affect the actual argument, or by reference, i.e. passing the actual parameter's address so that the formal becomes a new name for the actual.

This chapter addresses these fundamental assumptions about types and operations on them. We will derive a number of requirements, which are satisfied by the built-in types of C++, and which must be satisfied by user-defined types to be fully useful in generic programs. As with all interface specifications, these requirements may seem restrictive. But they are fundamental to our ability to accomplish anything on a generic basis. Furthermore, there is a side benefit to our attempt to match traditional type semantics. It gives us a much sounder basis for reading and understanding programs, since programs using the traditional primitive operators will behave as we expect.

It is interesting to contrast our approach with the traditional abstract data type paradigm. The ADT approach suggests that a new type should be defined with all of the operations and algorithms relevant to it, and then it can be used as a component of more complex modules. Object-oriented programming elaborates on this approach, providing language mechanisms (inheritance) for incorporating simpler types into more complex types, as well as propagating the operations defined on them. By contrast, generic programming takes a more top-down approach. We first develop an interesting algorithm or data structure (container), then identify the minimum requirements for the operations it needs on the underlying types, and finally apply those requirements to specify usable types, or concepts.

§ 2.2 -- Fundamental Operations

We have suggested above that a clear specification of fundamental operations on types is important to generic programming. Which operations are fundamental? In this section, we will identify several, all of which have default C++ definitions for all built-in data types and reasonable extensions to user-defined types. As we introduce them, we will also begin to discuss their required semantics. As we do so, a couple of common threads will emerge.

First is an emphasis on operators doing what we intuitively expect. We have all heard arguments about how important it is to be able to read code and understand it in order to maintain it. Generic programming amplifies this concern -- the writer of a generic algorithm cannot know about the implementations of the types and operators he uses, since such an algorithm is intended to work with a wide variety of types.

The second is a concern that we be able to apply various natural optimizations to generic code using these fundamental operators. Again, we want to optimize our generic algorithms without being concerned about whether the operators really have the semantics we expect.

Assignment

The assignment operator, e.g. x=y, copies the value of an object y to another object x. To understand this

apparently simple statement a bit better, consider the examples in the adjacent table.

A	`x = x;`	`;`
B	`x = y;` `v = x;`	`x = y;` `v = y;`
C	`x = y;` `f(x);`	`x = y;` `f(y);`
	original	optimized

Example A copies the value of variable x to itself. We intuitively expect this to be a useless operation, and require that we be able to remove it, since a programmer seeing it in a generic code fragment will naturally do so. This implies that the assignment may not perform any necessary functionality besides the copy.

Example B copies the value of variable y to x, and then copies x to variable v. This should be equivalent to copying v directly from y, i.e. the intermediate copy should not affect the final result. This optimization allows the two copies to proceed in parallel, and may allow the first copy to be removed if x is not otherwise needed.

Example C is similar to example B, if we assume f is a function taking a value (copy) parameter of the same type as x and y. In the original form, this code copies y to x, and then copies x to the formal parameter of function f. Again, we should be able to simply copy y to the formal parameter, allowing the two copies of y to proceed in parallel, and possibly allowing elimination of the first copy.

In C and C++, assignment also returns a reference to the target object, and we will assume this as well, although it is not critical to our development. The default C++ assignment operator for constructed types recursively assigns each component. For components that do not have user-defined assignments, this will ultimately do a bitwise copy of components of built-in types.

Some programming languages have treated assignment as copying a reference to y instead of its value -- we explicitly prefer the value semantics of C++. However, we anticipate a later discussion by pointing out that the value to be copied for a complex object requires careful definition.

Equality

The equality operator, e.g. a==b, compares two objects of the same type, and returns a Boolean (true/false) value indicating whether they are equal. There is no default C++ equality operator for constructed types.

Generic algorithms often use the equality operator, and they expect somewhat more than the definition above. First, the equality operator should define what mathematicians call an *equivalence relation*, that is, it should satisfy the following conditions, for any a, b, and c:

- a == a (reflexive property)
- if a == b, then b == a (symmetric property)

- if a == b and b == c, then a == c (transitive property)

Next, consider the following code fragment:

D	`if ( a == b ) {`
	`  if ( a == b ) {`
	`  statement1;`
	`  }`
	`  statement2;`
	`}`
		`if ( a == b ) {`
		`  statement1;`
		`  statement2;`
		`}`

	original	optimized

As programmers, we naturally expect that the inner equality test is redundant, and may be removed. Generic components should be able to make the same assumption. Hence, we require that the equality operator have the semantics that multiple equality tests on the same unmodified objects should return the same result, and that they do not have side effects which prevent their removal.

C++ also provides an inequality operator, e.g. a!=b, which is closely tied to the equality operator. For built-in types it is defined as the negation of the equality operator, and STL (but not the base C++ language) provides this default definition as a template for user-defined types without an explicit user-defined inequality operator. This is the behavior we expect from mathematics, and we require it of user-defined inequality operators. That is, a user should be able to write either a!=b or !(a==b) without worrying about whether they are equivalent. Consider another code fragment, similar to Example D above:

E	`if ( a == b ) {`
	`  if ( a != b ) {`
	`  statement1;`
	`  }`
	`  statement2;`
	`}`
		`if ( a == b ) {`
		`  statement2;`
		`}`

	original	optimized

In this case, we again expect the inner comparison to be redundant, and require that we be able to

remove it along with statement1. That is, we want the same conditions on reproducibility and side effects as for the equality operator, as well as the condition that they produce inverse results.

Constructors

When a new object is created, it occupies memory, and contains an initial value. In many programming languages, the initial value is undefined, or sometimes a fixed binary value, typically zero. Later languages have allowed the programmer to provide an initial value with some limitations, for instance that it must be constructed from compile time constants. C++ provides a completely general mechanism, allowing the definition of one or more operators, called constructors, which given zero or more parameter values, calculate the initial value. In fact, they may do arbitrary calculation. For example, a constructor for an object which controls a physical device might include code to force the device to a known state.

Constructors are used in a number of contexts where objects are created. One of these, of course, is data declarations, but there are others where we intuitively just expect a copy to be made, for instance when an actual parameter is copied to a formal value parameter in a subprogram call, or when the return value of a function is copied out.

Constructors which copy a single parameter to a new object of the same type are called copy constructors, and they are our principal concern here. We expect them to have behavior similar to the assignment operator, since both copy values. For example, consider a modification of assignment Example B above, where T is a type name.

This example differs from example B only in that it copies the value of variable y to a newly declared variable x using a copy constructor instead of to a previously declared x using assignment. We expect it to behave like the assignment example, and expect to be able to perform the same optimizations.

F	`T x (y);` `v = x;`	`T x (y);` `v = y;`
	original	optimized

As for assignment, the requirement that copy constructors copy the values of complex objects will be discussed with more precision later. C++ provides default copy constructors, which it defines to recursively perform copy construction of each member of a composite object.

Destructors

Destructors are an inverse operator for constructors. That is, when an object goes out of scope in C++, a destructor is called to do any necessary cleanup of its memory, or any other necessary final processing such as turning off a controlled device, before the object is deallocated. Our only requirement for destructors is that they clean up the same object that is constructed or copied by constructors and assignment. The significance of this seemingly simple requirement will become clearer when we discuss

the parts of complex objects later.

As an aside, we note that C++ also allows the explicit invocation of constructors and destructors. This is useful when it is necessary to separate the allocation and deallocation of memory from the construction and destruction of the objects which occupy it, for instance when a vector of objects is allocated as a unit and then initialized element by element.

The requirements for destructors are simpler than for the other fundamental operators in the sense that, since it is not valid to call a destructor twice for the same object, we need not be concerned with how they behave when redundant.

Comparisons

Many data types support a natural ordering, and C++ provides comparison operators for querying that ordering, namely <, <=, >=, and >. It provides default definitions for the operators on all built-in types, and allows user definition of them for user-defined types. Note that defining a default ordering for all built-in types automatically implies a natural ordering for composed types, given by composing the orderings of the components lexicographically.

Important generic algorithms, for instance sorting and searching, are based on ordering relations, and are naturally defined to use these operators. Although it is not necessary to require their definition and use, it is convenient to use them if available, and we require that they behave reasonably if defined. This means for our purposes that they define what mathematicians call a *total order* on the underlying type, that is for any a, b, and c of that type:

- either a<b, a==b, or a>b (and only one of these)
- if a<b, then b>a (anti-symmetric property)
- if a<b and b<c then a<c (transitive property)

We will address the axiomatization of orderings more precisely in Section 2.7.

These axioms again suggest several optimization possibilities, similar to the equality operator cases:

G	`x = y;`
	`if ( x < y ) statement1;`
	`if ( x > y ) statement2;`
		`x = y;`

H	`if ( x < y ) {`
	`  statement1;`
	`} else if ( x > y ) {`
	`  statement2;`
	`} else if ( x == y ) {`
	`  statement2;`
	`}`
		`if ( x < y ) {`
		`  statement1;`
		`} else if ( x > y ) {`
		`  statement2;`
		`} else {`
		`  statement2;`
		`}`

I	`if ( x < y ) {`
	`  if ( y < x ) statement1;`
	`  statement2;`
	`}`
		`if ( x < y ) {`
		`  statement2;`
		`}`

	original	optimized

Example G reflects our expectation that only one of the three relations <, >, or ==, holds between two values, and example H is another case of that rule. Example I reflects the anti-symmetric property, i.e. that if a<b then it cannot be true that b<a. All of these examples again exhibit what should be a familiar rule by now -- these operators must not have side effects which would invalidate an immediate re-test, or make it impossible to eliminate them when redundant.

General Principles

We have proposed a number of restrictions above on the fundamental operators we expect to use in writing generic program components. It is worth re-visiting our justification. These restrictions are not part of the C++ language definition, indeed C++ allows much greater leeway in defining these operators for user defined types.

However, our desired properties are not entirely without precedent. They are valid for all built-in types in C++, and indeed in all mainstream production programming languages. Moreover, these properties are heavily used for optimization in modern compilers, and virtually any programmer would be astounded to find them violated for the built-in types. Because the C++ language definition fails to make these guarantees, it is difficult for optimizers to make the optimizations we suggest automatically, but we can expect programmers to make them without giving the question much thought. Proposing to do generic programming, where there is typically no way to inspect the underlying operators, but where efficiency is vital, makes little sense.

In addition, in the case of the comparison operators, we have well-established mathematical tradition

behind the total ordering axioms we require. We should always remember that our programs, in addition to specifying actions for a computer, will typically be read and modified by other programmers. As mathematicians have known for centuries, the symbols we use to communicate ideas are arbitrary, but by following conventions we can eliminate most of the time spent decoding, and focus our energy on the ideas being expressed.

Another observation is worthwhile here. In mathematics, and the other sciences, the formal articulation of properties is always preceded by an intuitive understanding based on observation. Mathematicians generally develop a theory by first observing mathematical facts, then postulating theorems to explain them, and finally formalizing the axiomatic basis required for formal expression and proof.

Our work on underlying assumptions for generic programming is analogous to this. We have observed extensively the assumptions made by programmers and by compilers about simple type behavior. We have also observed that writing efficient generic programs requires that we make similar assumptions about the user-defined types we support. This book is an effort to present the axioms which embody these assumptions, and to demonstrate that a large body of extremely useful, general, and efficient code can be built on the resulting framework. Ultimately, the validity of these ideas must be judged based on the utility of the results.

---

§ 2.3 -- Interactions Between Fundamental Operators

We continue our discussion of requirements on abstract data types by investigating the relationships among equality, assignment, construction, and (where defined) destruction operators. We address these in the context of C++ semantics, with the extended assumptions from the preceding section, although the issues are common to all programming languages which support construction and destruction of user data types.

To begin the discussion, consider an example. We will look at a variable-length vector type, which consists of a current length member and a pointer to the actual data vector. We will assume that the data vector is owned by the header type, i.e. that distinct objects do not share data vectors.

```
class IntVec {
        int* v;  // Data vector
        int n;   // Length of vector v
public:
        IntVec (int len) : v(new int[len]), n(len) {}
        ~IntVec () { delete [] v; }
        int operator== (IntVec& x) { return v == x.v; }
        int& operator[] (int i) { return v[i]; }
        int size() { return n;}
};
```

This looks like a reasonable definition, on the surface. It contains a constructor which, given a desired vector length, initializes an object with the requested length and a pointer to a vector which it allocates with that length. It also has a destructor which deallocates the data vector, an equality operator which compares the vector pointers, an indexing function, and a size function.

However, it is not a good type definition, for a number of reasons. Can you see why? First, think about passing an object of this type by value to a function.

In C++, an argument passed by value is copied to the local parameter by invoking the copy constructor. Since the type doesn't provide a non-default copy constructor, the C++ default is used: the data members are copied bitwise. As a result, we don't actually have a full copy of the vector in the function -- our local parameter is sharing the data vector. Contrary to expectations, modifications of the parameter's data vector will also affect the original. Worse still, let's consider what happens when we call the most trivial of functions:

```
void foo ( IntVec ) {}
```

The body of this function does nothing. However, the C++ semantics of passing a parameter by value requires that the IntVec actual parameter be copied to the formal parameter using the copy constructor, with the result mentioned above that the formal parameter contains a data vector pointer which is a copy of the actual parameter's data vector pointer. After doing nothing, the function returns to its caller. But before it can do so, because the formal parameter is going out of scope, its destructor must be called. This will deallocate the data vector which it shares with the formal parameter, leaving it with a dangling pointer to unallocated memory. Thus, it is not possible to pass our IntVec objects by value to a function without destroying them.

Clearly then, we must provide a copy constructor:

```
class IntVec {
        int* v;  // Data vector
        int n;   // Length of vector v
public:
        IntVec (int len) : v(new int[len]), n(len) {}
        IntVec (IntVec& x);
        ~IntVec () { delete [] v; }
        int operator== (IntVec& x) { return v == x.v; }
        int& operator[] (int i) { return v[i]; }
        int size() { return n;}
};

IntVec::IntVec ( IntVec& x ) :
         v ( new int[x.size()] ),
         n ( x.size() )
{ for ( int i = 0; i < size(); i++ ) (*this)[i] = x[i]; }
```

This copy constructor allocates a new data vector for the copy, and copies the source object's data vector into it. This now does what we expect for a copy, and illustrates our first requirement:

> Copy 1: A type representing values that are not entirely contained within a single C++ class object must have a non-default copy constructor that copies the entire value.

This is our first encounter with an issue which will arise frequently. What is logically an object need not correspond exactly to a C++ type. When we speak of an object, we mean not only the primary C++ variable to which we refer explicitly, but also any additional components which are part of the logical object. Even with this intuitive understanding, however, the phrase "entire value" above will require deeper discussion later.

Let us return to our example. Is IntVec correct now? Consider a test of a simple *swap* function:

```
template <class T>
inline void swap (T& a, T& b) {
       T tmp = a;
       a = b;
       b = tmp;
}

template <class T>
bool test_of_swap (T& a, T& b) {
       T oldA = a;
       T oldB = b;
       swap(a, b);
       return a == oldB && b == oldA;
}
```

Will our IntVec definition pass a test_of_swap? This routine saves copies of the original parameters, calls swap on them, and then tests the resulting values against the saved copies. Consider carefully what each of the steps will do with our definition, and then try the following test program.

```
void initialize_IntVec ( IntVec& v, int start )
{
  for ( int i = 0; i < v.size(); i++ ) v[i] = start++;
}

main() {
  IntVec a(3);
  IntVec b(3);
  initialize_IntVec ( a, 0 );
  initialize_IntVec ( b, 1 );
  if ( test_of_swap (a, b) )
    printf ( "test of swap - passed\n" );
  else
    printf ( "test of swap - failed\n" );
}
```

This fails -- why? Let us identify precisely what happens, statement by statement.

The two IntVec declarations create two dynamic vectors of length 3, a and b. The calls to

initialize_IntVec then initialize them, a to {0,1,2} and b to {1,2,3}. Since the dynamic data vectors for a and b do not have variable names, let us call them av1 and bv1, with the digit 1 indicating that these are the first copies created.

We then call test_of_swap. It begins by creating copies of a and b using the copy constructor, for use later in determining whether swap worked correctly. These copies, oldA and oldB, point to new copies of the data vectors, which we will call av2 and bv2.

Next our test program calls the swap function. It copies a to a temporary named tmp, again using the copy constructor because it does so in the declaration of tmp. In doing so, it again copies the data vector for a, and we will call the new copy av3. At this point, we have three copies of a's data vector, and two copies of b's data vector, all at different addresses.

Now, the swap function assigns b to a, and tmp to b. It uses the built-in C++ assignment operator, which simply assigns the objects component by component, in this case copying the length member and the data vector pointer. Therefore, after the assignments, a will point to bv1, and b will point to av3. When we return from swap, the test "`a == oldB`" compares the data vector pointers, that is, it compares the address of bv1 to the address of bv2, and fails.

It should be clear what we need to do to solve this problem, which is due to the fact that our type has *remote* parts, i.e. parts which are not contiguous to the primary record. The principal is as follows:

> Equality 1: When comparing the values of two objects of a type with remote parts, the equality operator must ignore the pointers to the remote parts, and compare the parts themselves.

Another way of viewing this requirement highlights the interaction between copy operations (constructors and assignment) and the equality operator:

> Equality 2: When comparing the values of two objects of a type with remote parts, the equality operator should compare only the values of parts which are copied by copy operations, ignoring parts which serve only to locate the remote parts, and which always have distinct values in distinct objects.

As we shall see in the next section, equality must deal with additional subtleties as well, but this is adequate for our current example. It gives us a new definition of the IntVec equality operator:

```
bool IntVec::operator==(IntVec& x) {
    if ( size() != x.size() ) return false;
    for ( int i = 0; i < size(); i++ ) {
        if ((*this)[i] != x[i]) return false;
    }
    return true;
}
```

With this new equality, the test above will probably pass. However, there is still a subtle bug. Can you identify it? Consider the following revised test program, which repeats the test, and extend the analysis we did above:

```
main() {
    IntVec a(3);
    IntVec b(3);
    initialize_IntVec ( a, 0 );
    initialize_IntVec ( b, 1 );
    if ( test_of_swap (a, b) )
        printf ( "test of swap - passed 1\n" );
    else
        printf ( "test of swap - failed 1\n" );
    if ( test_of_swap (a, b) )
        printf ( "test of swap - passed 2\n" );
    else
        printf ( "test of swap - failed 2\n" );
}
```

This modified test will probably fail. To see why, led us go back to the state at the end of the first call to the swap function. Recall that b points to av3, i.e. the copy of the a vector which was created to initialize tmp. When we exit the swap function, what happens to tmp? Since it is a local variable of swap, it goes out of scope and its destructor is called. In this case, the destructor deallocates its data vector, which also happens to be the new data vector of b. As a result, we return from swap with b pointing to deallocated memory for its data vector. Although it will probably "pass" the test "`b==oldA`" because that memory has not yet been re-allocated and modified, it will only be a matter of time until b's value is apparently changed by a seemingly irrelevant event, and the program fails. In the case of our modified test, that irrelevant event is likely to be the allocation and initialization of a new tmp variable in the same memory during the second call to swap. However, it is highly dependent on the implementation's memory allocation policies, and the dangling pointer to invalid memory could continue to "work" indefinitely if

the deallocated memory is left unused.

Once again, the correction is obvious once we understand the problem. We must define an assignment operator which copies the data vector just as the copy constructor does.

> Assignment 1: A type representing values that are not entirely contained within a single C++ class object must have a non-default assignment operator that copies the entire value.

However, there is a complication in the assignment case which does not occur in the constructor case. Suppose we simply use the copy constructor:

```
IntVec& IntVec::operator= ( IntVec& x )
{
    new (this) IntVec(x);
    return *this;
}
```

This code will certainly copy the value of x, including its data vector, to the left-hand side assignment operand. But what happens to the previous value of that operand? It will simply be overwritten, which is incorrect if it is a type with a destructor. Therefore we must first call the destructor:

```
IntVec& IntVec::operator= ( IntVec& x )
{
    this->IntVec::~IntVec();
    new (this) IntVec(x);
    return *this;
}
```

Even this is not quite correct, however. What will happen on the assignment "a=a"? We will destroy the old value of a, before attempting to copy the value. To solve the problem, it is necessary to avoid the destructor call for a self-assignments. These observations give us two more conditions on assignment:

> Assignment 2: Assignment operators for types with non-trivial destructors must destroy the old value of the left-hand side before copying the value of the right-hand side.

Assignment 3: The destructor call required by Assignment 2 must be avoided when assigning an object to itself.

Once we meet these conditions, we finally have a good definition:

```
IntVec& IntVec::operator= ( IntVec& x )
{
    if ( this != &x ) {
        this->IntVec::~IntVec();
        new (this) IntVec(x);
    }
    return *this;
}
```

That is, before calling the destructor and constructor, we must first verify that the source and destination objects are different. If not, nothing needs to be done.

Observe that this is a generally valid model for assignment -- it will always work. There are simpler cases which do not require the explicit test, for instance where there are no non-default destructors. However, this is correctly viewed as a special case optimization, and not as and exception to the validity of the general definition. It is unfortunate that this is not the default C++ definition -- in its absence, non-default assignment operators are required much more often than should have been necessary.

Let us finish up our discussion of the IntVec example by presenting its full, correct definition:

```
class IntVec {
   int* v;      // Data vector
   int n;       // Length of vector v
public:
   IntVec (int len) : v(new int[len]), n(len) {}
   IntVec (IntVec& x);
   ~IntVec () { delete [] v; }
   int operator== (IntVec& x);
   int& operator[] (int i) { return v[i]; }
   int size() { return n;}
};
```

```
IntVec::IntVec ( IntVec& x ) :
    v ( new int[x.size()] ),
    n ( x.size() )
{
    for ( int i = 0; i < size(); i++ )
        (*this)[i] = x[i];
}

IntVec& IntVec::operator= ( IntVec& x )
{
    if ( this != &x ) {
        this->IntVec::~IntVec();
        new (this) IntVec(x);
    }
    return *this;
}

bool IntVec::operator== ( IntVec& x )
{
    if ( size() != x.size() ) return false;
    for ( int i = 0; i < size(); i++ ) {
        if ((*this)[i] != x[i]) return false;
    }
    return true;
}
```

We now have enough information to summarize the required operations on what we will call *regular types*. Except where otherwise specified, we will assume that all types with which we deal are regular.

Table: Required Operations on Regular Types

(syntax for a type named 'Regular')

Operation	C++ Syntax (invocation)	Result type	Conditions, comments
Default constructor	Regular a; Regular()	n/a	Creates a valid object
Copy constructor	Regular a = b; Regular a(b);	n/a	Postcondition (a == b)

Copy constructor	new (&a) Regular(b)	n/a	Postcondition (a==b); Constructed in memory at &a
Reference	&a	Regular&	May pass parameter by reference
Destructor	&a->~Regular()	n/a	Destroys all parts of object
Assignment	a = b	Regular&	assert (&a == &(a = b))
Equality	a == b	bool	assert (a == a) a==b implies b==a a==b && b==c implies a==c &a==&b implies a==b
Inequality	a != b	bool	a != b iff !(a == b)

---

§ 2.4 -- Equality and Parts of an Object

In the previous section, we discussed the necessary relationships between the equality, assignment, construction, and destruction operators, but we glossed over their precise definitions. The correct definition of equality and copy operators is intimately tied to the concept of the parts of a data structure. We shall discuss these concepts in detail in this section.

Primitive and Composite Types

We call a type *primitive* if it is directly supported by the underlying computer hardware, or if it has built-in language support similar to that for hardware types. We call a type *composite* if it is composed of member types, which we call its *parts*. Furthermore, the parts may be composed of parts themselves, which we call *indirect* parts of the original types. We apply the same terminology to objects as to the types of which they are instances.

For example, integer, pointer, and floating point types are primitive in C++. Our variable-length vector of the previous section is composite, composed from the length member and the data vector members, and a list data type would be composite as well.

For primitive objects, the definitions of the equality and copy operators are easy -- we simply compare or copy the objects' values. For composite objects, however, there are more possibilities to choose from. Even for record types (e.g. C structs), the problem is not difficult -- a member-by-member definition works well for these operators. But for more complex data structures, such as lists or trees, the question can become quite difficult. Our objective in this section is to determine how we must define these

operators to give all composite types the same value semantics as primitive types.

Development of Composite Types

Composite types have developed slowly over the history of programming languages. The first high-level programming language, Fortran, supported primitive data types (e.g. INTEGER and REAL), and arrays as the only method of creating composite types. Arrays were extremely important in allowing large numbers of data items to be processed, but their capabilities were quite limited. The most obvious restriction limits their component parts to be homogeneous, i.e. all of the same type. Just as important, Fortran avoided choosing a correct definition of the equality and assignment operators for arrays by not defining them at all.

The next step was to allow composite types with heterogeneous parts. These were called *records* in Pascal and Ada, *structs* in C, and eventually *classes* in C++. Pascal supported neither assignment nor equality operators on such types and C supported only assignment; not until C++ did a mainstream language provide default definitions of the key operators, and even then not equality.

Even record types, however, do not allow self-contained definition of complex data structures such as lists or trees. Because such structures change dynamically as programs execute, providing static definitions for them as single structures would be difficult. Instead, the approach is to define their parts as class types, and to connect the parts using pointers. When an object has parts which are allocated elsewhere, we called them *remote* parts. Even C++ does not attempt to properly define equality and assignment for objects with remote parts, although it provides the mechanisms for user definition.

Equality of Composite Types

It is our objective to provide definitions of equality and copy operators which will make even variable size data structures look and feel like primitive types or simple records. However, it is these complex objects which most stress our intuition. In order to produce consistent operator definitions, based on a clear understanding of the parts of an object, it is useful to consider some of their history in mathematics and philosophy.

First, let us look at the meaning of equality. In mathematical logic, two values of a given type are considered equal if and only if any function defined on that type returns the same result when applied to either value. Naively, this definition matches our intuition quite well -- it is natural to assume that any function will produce equal results when applied to equal values. Unfortunately, there are two serious problems with this definition.

First, it is not a good operational definition of equality, since we cannot very well try all possible functions and compare their results. Fortunately, there is another definition which does produce a workable procedure for equality testing, and corresponds to how we would naturally compare two objects by hand. This is to verify that the two objects have matching parts, with the same structure

connecting the parts, and with the corresponding parts having equal values.

Second, the mathematical definition is not even a valid one for programming objects or indeed for physical objects. Recall that we want two objects to compare equal after after we copy the value of one to the other. However, in most programming languages we can define functions based on the addresses of objects. In the case of two distinct objects, their addresses are not equal, and such functions can return different results for objects with equal values.

For a more subtle example of the same problem, consider our variable-length vector from the previous section. If we copy one vector to the other, and then try to test equality by comparing each component of the objects, we will correctly determine that the lengths are equal, and that the data components of the vectors are equal, but we will also determine that the pointers in the vector headers are unequal because they point to different copies of the data values. Thus we can see that even our alternate definition of equality will not work for objects containing pointers between their components, unless we treat those pointers differently from other parts. Clearly they are different from other parts of the data object, and ought to be ignored in equality comparisons. How can we wore precisely characterize this distinction?

To improve our intuition, let us redirect our attention from mathematics to the physical world. Consider a chair. It has a variety of physical characteristics, such as its color, weight, arm style, number of legs, etc. Another chair might have precisely the same physical characteristics, and we would consider them to be equal. We would hold this opinion whether they were next to one another at a table, or on opposite sides of the world. That is, the location of the chair has no bearing on our view of its "value." Nor does the location of its arm or its leg.

These observations provide the basis for the most important distinction between parts of an object. We can see that in addition to the essential parts of an object, which carry its value, there are other parts which serve only to keep track of the locations of non-adjacent parts, e.g. pointers between the records which comprise a complex data structure. We will call these latter parts *structural links*, because they provide structural connections between the parts of an object which actually hold its value. They must not be considered in equality comparisons. As we saw earlier, they should also not be copied -- rather the remote parts to which they point must be copied, and the new link will point to the copied parts.

Relationships Among Objects

The distinction between essential parts of an object and its structural links is not the only relevant distinction, however. Let us return to our chair example. One chair might have a person sitting in it, while the other holds a sleeping dog. Although these facts might be viewed as attributes of the chairs, we would still consider the chairs to be equal. That is to say, the content of a chair, although it is clearly associated with the chair, is not viewed as an essential part of the chair's value.

This situation is common in programming as well. For example, an insurance company database containing records of insurance policies would probably associate with each policy a reference to

another record for the customer holding the policy. However, it is unlikely that someone comparing two policies for equality would want to take the policyholder into consideration.

There is a more compelling consideration, though, involving the destruction of objects, where it is important to to be clear about where one object ends and another begins. We will be writing generic code which destroys objects, and if we want it to work correctly we must follow consistent rules, which we can depend upon without knowledge of the specific type. Although one could conceivably treat each data structure as a unique case, with specialized protocols for construction and destruction, and make this work for a custom program, generic programming requires more discipline.

When we destroy an object by calling its destructor, we want to know that we have destroyed the entire object, and that we have not destroyed part or all of another object. We can deal with the destroyed object being part of a larger object -- indeed, we will destroy complex objects in general by destroying their parts -- but the destroyed object may only be part of a single object. To put this another way, we do not allow the sharing of parts between objects. If two objects share a part, one must be part of the other.

Similarly, to avoid problems with our paradigm of destroying an object by destroying it parts, we must avoid circularity among objects. That is, an object cannot be part of itself, and therefore cannot be part of any of it parts.

These requirements, inspired by the needs of destructors, provide us with the second important distinction between parts. A part is not an essential part of the value of an object if its purpose is simply to provide a link to a distinct object. Thus, in our insurance company example above, the customer link is not an essential part of the insurance policy object.

This leaves us with three categories of object parts:

- *Essential parts* are components of the object's value.
- *Structural links* connect remote parts of an object.
- *Relationship links* connect related but distinct objects.

We can use these categories to finally resolve how the fundamental operators must treat composite objects. Equality should compare the essential parts of an object, following structural links to compare remote essential parts, but not comparing the structural links themselves. Moreover, equality should normally ignore the relationship links.

Philosophical Background

*We can find these distinctions made by philosophers as far back as Aristotle, who viewed objects as consisting of **form** and **matter**. What he called an object's form is the fundamental pattern for an object, which is analogous to the concept of type in programming languages. For example, all chairs would have the same form. What Aristotle called matter is the actual physical instantiation of the object, which is analogous to the concept of value in programming languages, and is central to our search for valid copy and equality operators.*

*Much later, in the late middle ages, John Duns Scotus added **haecceity** (from Latin "hic" - this) as the principle of individuation, i.e. that which distinguishes between two identical chairs or*

Copy operators (i.e. assignment and copy constructors) should copy the essential parts, including remote essential parts found by following structural links, linking the remote parts in the new object with new structural links. Again, copies should normally ignore relationship links. Finally, destructors should destroy all parts of an object, following structural links to destroy remote parts, but not following relationship links to distinct objects.

< to be continued >

An ADDRESSABLE part is a part to which a reference can be obtained (through public member functions)

houses. Although there have been extensive metaphysical arguments about whether matter or haecceity constitutes the proper principle of individuation, in the case of computers the trichotomy is quite clear:

- *type <=> form*
- *value <=> matter*
- *address <=> haecceity*

Attributes related to haecceity, such as location (address), name, and relationships to other objects, are at the root of our failures of intuition. Because they reflect the distinctions between objects, rather than distinctions between values, they play no legitimate role in the value-based semantics of copy constructors, assignment, and equality testing.

An ACCESSIBLE part is a part of which the value can be determined by public member functions.

Every addressable part is also accessible (if a reference is available, it's trivial to obtain the value).

An OPAQUE object is an object with no addressable parts.

An ITERATOR is an object which refers to another object, in particular, it provides operator() returning a reference to the other object.*

Two non-iterator objects are equal when all the corresponding non-iterator accessible parts are equal

*Two iterators are equal when they refer to the same object (i == j iff &*i == &*j)*

An implicit function AREA is defined for all objects

For the primitive objects the area is equal to sizeof()

For the composite objects the area is equal to the sum of the areas of its parts

An object is FIXED SIZE if it has the same set of parts over its lifetime

An object is EXTENSIBLE if not fixed size

A part is called PERMANENTLY PLACED if it resides at the same memory location over its lifetime

(Knowing that a part is permanently placed or not allows us to know how long a pointer which points to it is valid)

An object is called PERMANENTLY PLACED if every part of the object is permanently placed

An object is called SIMPLE if it is fixed size and permanently placed

For a regular object the complexity of all the fundamental operations (constructors, destructors, assignment and equality) is linear in the area of the object.

Two objects are equal if they have the same number of ESSENTIAL parts and their corresponding essential parts are equal.

A function is REGULAR if it returns equal results for equal values of arguments.

Question: Why is it important for an optimizer to know if a function is regular?

---

§ 2.5 -- Swap Algorithm

In this section, we will develop a simple example: a function which swaps two values. It is a simple example of a generic algorithm, but more importantly, it is a key function for a number of generic algorithms, and appropriate tailoring to the characteristics of the type on which it operates can make a significant difference in the performance of generic algorithms such as sorting.

First, consider how we would normally swap two integer values:

```
int a,b;
...
int tmp = a;
a = b;
b = tmp;
```

This code fragment simply creates a temporary variable initialized to the first value, copies the second value to the first variable, and finally copies the temporary variable back to the second variable. It is code which we have all written many times, and it is as efficient as possible. In fact, a clever compiler may avoid doing any data movement beyond reading the old values of the variables a and b, and writing the new values.

What does this code fragment assume? It requires that the copy constructor initialize tmp to the value of a, and that the two following assignments copy the right-hand-side values to their left-hand-side variables. Since we have placed these requirements on any regular type T, we can easily turn this code fragment into a generic function for any such type T:

```
template <class T>
inline void swap (T& a, T& b) {
       T tmp = a;
       a = b;
       b = tmp;
}
```

This function is straightforward. The parameters are passed by reference, since they must be modified. Otherwise, this swap template requires no comment in the general case. It will very efficiently swap the values of a and b by copying each to the other, using a temporary intermediate.

In particular cases, however, it may be extremely inefficient. Why? Think about our variable-length integer vector type in section 2.2. There, the above code will carefully copy not only the vector length, but also the vector data. However, that is unnecessary. We can accomplish the same semantics by simply swapping the pointers to the vector data. Since those pointers are private members of the class, we can implement swap as a member function, and then invoke it from a global function with the above interface.

```
class IntVec {
   int* v;      // Data vector
   int n;       // Length of vector v
public:
   ...
   void swap ( IntVec & b )
   {
      swap ( this->v, b.v );
      swap ( this->n, b.n );
   }
};

inline void swap ( IntVec &a, IntVec &b )
{
  a.swap ( b );
```

```
}
```

Observe that we swap the data members n and v of the class using our generic swap function above, since we can't do better.

The lesson here is that for a type with remote parts, the swap function can be implemented more efficiently than the general template by simply swapping the pointers to the remote parts instead of copying the remote parts. When implementing a type intended to be used with generic algorithms like sorting, it is well worthwhile to provide such a tailored swap function.

A less obvious lesson is that it is useful to recognize cases like swap where tailored versions will be much more efficient than a general version. In such cases, we can provide a template for the general version, while describing an interface that providers of new types can use to give us a more efficient version when that is beneficial. Following this principle, the algorithms in the C++ Standard Template Library consistently use the swap function when swapping values, rather than in-line code, in order to take advantage of such specialized implementations.

---

§ 2.6 -- Generic Optimizations on Regular Types

We discussed the optimization requirements of individual operations in section 2.2. Here, we want to give examples of a couple of traditional optimizations (e.g. CSE, copy propagation), and point out that regular type semantics allows us to apply them at a higher level, in spite of the implementation of the fundamental operations as functions.

---

§ 2.7 -- Ordering Relations and Comparison Operators

In our description of fundamental operations above, we described basic requirements on the standard C++ comparison operators <, <=, >=, and >. Since many algorithms are highly dependent on ordering relations and the comparison operators which represent them, and are sometimes sensitive to subtle distinctions between different possible ordering relations, we will provide a careful characterization in this section.

Ordering Relations

What is an ordering relation? In its most general form, it is a transitive binary relation on a set of values. That is, if we write "a prec b" for a pair (a,b) satisfying the ordering relation, then the following property

holds:

Order 1: a prec b **and** b prec c **implies** a prec c (*transitive*)

This is a fairly weak axiom, and most ordering relations of interest satisfied additional important properties. We shall discuss several of them in the following paragraphs.

The first such property is *strictness*. A strict ordering is one in which a value does not precede itself. That is, the following axiom also holds:

Order 2: **not** a prec a (*strict*)

The standard "less than" relation on numbers is a strict order. By contrast, "less than or equal" is not strict.

Another important property produces a *total* ordering, that is one in which any pair of values is either equal or ordered by the relation:

Order 3: **for all** a, b: a=b **or** a prec b **or** b prec a (*total*)

These possibilities are not exclusive. Both the standard "less than" and "less than or equal" relations on numbers are total. An ordering which is not total is *partial*.

Finally, there are ordering relations which are extremely important in programming, and which behave much like total orderings but are not total. Consider what is required to sort a set of employee records by salary. In a company of any significant size, it is likely that there will be employees with equal salaries. Assuming that we don't care how they are ordered in the sorted list, we want an ordering relation which treats the employee records with equal salaries as though they were equal, but is otherwise a total ordering based on the salary component.

Such an ordering relation is called a *weak* ordering. More precisely, it divides the set being ordered into disjoint subsets which are equivalent with respect to the ordering (i.e. two elements in the same subset are ordered the same relative to all other elements), and induces a total ordering on the subsets. Weak orderings may be either strict or non-strict.

Note that although any ordering determines a decomposition of the set being ordered into equivalence classes of elements which are ordered the same relative to other elements, the set of equivalence classes need not be totally ordered in general. The weak ordering constraint is a fairly strong one.

Comparison Operators

Now that we have a rudimentary understanding of possible ordering relations, we can turn our attention

to the comparison operators which implement them. As we noted in section 2.2, C++ defines the standard comparison operators <, <=, >=, and > for built-in types, as do almost all programming languages. These operators implement a total ordering on the built-in types: < and > are strict, while <= and >= are non-strict. Furthermore, these operators have a well understood relation to one another:

- a>b *iff* b<a
- a<=b *iff* ! b<a
- a>=b *iff* ! a<b
- either a<b, a==b, or a>b (and only one of these)

These relationships show that we could implement all four operators in terms of the < operator. In fact, STL does this, providing templates for the three other operators which produce the correct functions given a reasonable definition of the < operator. Any of the four operators could have served as the base operator in this sense, but a specific choice had to be made to avoid ambiguity.

Since the composition of components in generic programming depends on equivalent behavior from the same operators (i.e. those with the same name) on different types, we require that any types defining these standard comparison operators define them with the same properties. This implies that they implement a total ordering. Given the templates for the other operators in STL mentioned above, a user defining a type need only define an appropriate < operator.

For a composite type, a suitable < operator can be defined lexicographically. That is, the first member fields are compared using their < operators, if they are equal the second fields are compared, and so on. This is the same approach typically used to compare character strings.

The STL algorithms which depend on orderings use these standard comparison operators by default, with these assumptions. As we observed above, however, what we want is often not a total ordering. The same STL algorithms normally also provide for use of an alternate ordering relation, in the form of a comparison operator. They consistently require that this operator implement a strict weak ordering. The choice between a strict and a non-strict ordering is arbitrary, in these sense that either one would produce working, efficient algorithms. However, one must be chosen, because some of the algorithms require subtle differences in implementation depending on the choice. In fact, some STL algorithms might not terminate given a non-strict comparison operator.

It is important to make one further point about comparison operators. The performance of many algorithms is dominated by the cost of comparisons. Furthermore, some of those algorithms make different choices for the cases a<b, a==b, and a>b. Typically, a comparison function can distinguish these three cases just as easily as it can produce a Boolean result for one of the standard two-way operators. In cases where the comparison is expensive, for example character strings, using a three-way comparison operator which returns a negative, zero, or positive integer can produce a significant performance improvement. We shall see several such cases in this book.

---

§ 2.8 -- Min and Max Algorithms

In this section, we are going to discuss one of the more common generic applications of the ordering relations from the preceding section, namely the min and max functions. In doing so, we will encounter a number of important programming issues.

First, we will thoroughly investigate a simple template for the min function. For this simplest form, we need only assume that we have an operand type for which the "<" operator provides a strict total order. Recall from section 2.2 that this requirement is satisfied by all C++ built-in types, and that it is easily satisfied for any user-defined type simply by defining "<" lexicographically.

```
template < class StrictTotalOrderable >
inline StrictTotalOrderable& min (
    StrictTotalOrderable& x,
    StrictTotalOrderable& y )
{
    return y < x ? y : x;
}
```

Let us work our way through this example line by line. In the template header, we specify a template parameter "StrictTotalOrderable." This reflects our requirement that the operand type be totally ordered by "<". Of course, the name carries no implicit meaning to a C++ compiler, so there will be no compiler enforcement of our requirement. It would be ideal to if the language allowed us to specify a *concept* template parameter, i.e. a type or class along with the additional requirements we wished to impose, such as a "<" operator in this case. However, this ideal will need to wait for a future language design.

We begin the second line by declaring our function *inline*. This is an easy choice, since the body of the function will seldom be more code than a call would have been, and will always be faster. Inlining the call will provide the compiler with maximum information for optimization. More interesting in this line is the return type. Why do we return a reference instead of a value? This is because we want to be able to do the following:

```
min(x,y) = 5;
```

This statement assigns the value 5 to whichever of x and y contained the smaller value, regardless of what that smaller value was. In general, this treatment allows us to use min to identify which of two variables contains the smaller value and use that knowledge directly in contexts where we do not care what the actual minimum value is. Of course, this usage is not valid in cases where the operands are not

valid C++ lvalues (i.e. assignable objects), but our definition still works for valid usages in such cases.

The next lines are our formal parameter declarations. Again, the interesting question is why we pass the parameters by reference rather than by value. There are several reasons. They may be large objects, and will promptly be passed to the "<" operator -- there is no reason to do extra copies. In an inline function, the reference parameters give the compiler the most flexibility for optimizing its treatment, since they do not require the semantics of a full copy. Since this function does not modify it parameters, there is no reason to demand copy semantics. Finally, of course, in order to return a reference to the smaller actual parameter, we must pass a reference rather than a value in the first place.

The last line of interest is the function body. It is straightforward, raising no obvious questions, but it does reflect a subtle decision. Why don't we write it as:

```
return x < y ? x : y;
```

The answer concerns how we want this function to behave in the case where the two operands are equal. In particular, we want it to consistently return the first operand in that case -- this will often prevent unnecessary activity in the caller in those cases.

Next, we must understand what this simple min function cannot do. The first problem is a result of the behavior of the C++ type system. Consider what happens when x and/or y are const objects, a case which will often occur inside functions with const parameters. In this case, the template class parameter "StrictTotalOrderable" will match the base type (without the const attribute) of the min parameter, and the compiler will object that we cannot pass a const value as an actual parameter to a non-const formal parameter. In order to deal with this, we need to define an alternate form of the template:

```
template < class StrictTotalOrderable >
inline const StrictTotalOrderable& min (
    const StrictTotalOrderable& x,
    const StrictTotalOrderable& y )
{
    return y < x ? y : x;
}
```

Observe that this version is completely identical to the base version except for the const attributes. If either of the actual parameters is const, this version will be matched instead. Because we returned a reference to one of the actual parameters, the result type must also have the const attribute.

The second inadequacy of the original version is a more general generic programming issue. The templates above use the operator "<" -- this is the standard definition and by far the most common

interpretation for numeric types, but for user-defined types we are much more likely to be interested in alternate orderings. The ordering we wish to use may not meet our requirements for "<". In particular, it may not be a total ordering. For instance, we often need to compare two structures on the basis of a subset of their fields, producing an ordering in which they may compare as equivalent because those fields are equal, even though other fields are not. Even if we want to use a total ordering, there may be more than one total ordering of interest, and the current one may not be defined with the built-in operator "<".

For all of these reasons, we need a version of the template in which the comparison operator is itself a parameter. This new comparison operator does not need to define a total ordering -- a strict weak ordering will do. Therefore, we will rename the template formal parameter for the operands, although this has no real semantic effect in C++.

```
template < class StrictWeakOrderable,
           class StrictWeakOrderingRelation >
inline StrictWeakOrderable& min (
    StrictWeakOrderable& x,
    StrictWeakOrderable& y,
    StrictWeakOrderingRelation rel )
{
    return rel(y,x) ? y : x;
}

template < class StrictWeakOrderable,
           class StrictWeakOrderingRelation >
inline const StrictWeakOrderable& min (
    const StrictWeakOrderable& x,
    const StrictWeakOrderable& y,
    StrictWeakOrderingRelation rel )
{
    return rel(y,x) ? y : x;
}
```

This pair of templates is completely analogous to the preceding pair, and all of the earlier comments apply. Only the second template formal parameter and the third function formal parameter are new, and lead to new observations. First, we note that the new parameter is declared with a new class type, StrictWeakOrderingRelation, and used like a function. This will typically be what we call a function object, which we will discuss more thoroughly in the next chapter.

Second, observe that the new parameter is passed by value. Why? As a function object, the required

function address will usually be implicit in its type, and the object itself will often have smaller size than a pointer. We want to avoid forcing the caller to load a larger pointer, and the callee to dereference it, in those cases where it is not inlined.

Now that we have a full set of min function templates, it is straightforward to produce the corresponding max function templates -- we leave this as an exercise to the reader. The only tricky aspect is that we place the same requirement on max as we did on min, namely that it must return a reference to the first parameter if the two are equivalent.

We will close the discussion of min and max operators by commenting on several of their properties. It is trivially verified that

a == max(a, a)
a == min(a, a)

More importantly, min and max are associative operators, i.e.

min (a, min(b,c)) == min (min(a,b), c)
max (a, max(b,c)) == max (max(a,b), c)

Finally, these are almost commutative operators, with two exceptions. These exceptions may or may not be important for any particular use, so we will note them explicitly. As long as the relational operator used induces a total ordering, as does "<", they will be commutative in terms of the result returned, but not in terms of the results' addresses. That is:

min (a, b) == min (b, a); but
& min (a, b) != & min (b, a); if a==b

If the ordering is not total, the operators are not even commutative in terms of values, in the case of two operands which are equivalent but not equal. For example, considered the case of a pair-of-integers operand type, where the relational operator compares the first element of the pair and ignores the second. In this case, we have:

max (make_pair(1, 2), make_pair(1, 1), compare_first()) == (1,2)

but:

max (make_pair(1, 1), make_pair(1, 2), compare_first()) == (1,1)

In spite of these exceptions, however, the min and max operators may be treated as commutative for many purposes.

To conclude this section, we will take a brief look at another algorithm which is quite similar to min/max, and can use them in its implementation. It is *median*, which finds the median, or middle, element of a set. Specifically, we will look at a function of three values which returns a value that is neither larger nor smaller than the other two. (This rather odd statement, of course, is necessary because equal or equivalent operands may mean that there is no strictly middle value.) Our interest in this function, other than its similarity to the min/max functions we have just studied, results from its use in other algorithms we will see later, such as efficient implementation of quicksort.

We will impose the same requirements on median as we did on min and max. In particular, for the base version, we require a "<" operator inducing a strict total ordering, and if more than one of the operands is equivalent to the median value, we want to return the first. With these requirements stated, here is a template for median:

```
template < class StrictTotalOrderable >
inline StrictTotalOrderable& median (
    StrictTotalOrderable& a,
    StrictTotalOrderable& b,
    StrictTotalOrderable& c )
{
  if (a < b)
    if (b < c)
      return b;
    else
      return max(a,c);
  else if (a < c)
    return a;
  else
    return max(b,c);
}
```

We will not analyze this function as we did min, but we encourage the reader to do so as an exercise. Moreover, the reader should attempt to define the same alternate forms as were necessary for min and max. Finally, implementing a function to return the median of 5 values is an interesting exercise.

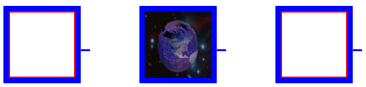
Implementation Queue

The following are points to be integrated somewhere:

- *The main problem with formal methods in CS was not that they were formal, but that they were not dealing with real problems. You have to build formalizations of large set of useful things.*
- *Complexity is part of specification.*
- *What is a type? It is an attribute of a location.*
- *Check that all the points from Rune's homework #1 from 10 April are covered in the section developing int_vec.*

22 July class

Send comments about this page to Jim Dehnert or Alex Stepanov.



Generic Programming

Part I: Fundamental Concepts

Chapter 3: Software Components

Last updated 10 April 1998

---

§ 3.1 -- Introduction

Generic programming is based on the idea that we can break programs down into components, and write many of the components in a general way that allows them to be mixed and matched with a variety of other components. We implement this composition by defining the set of interfaces (i.e. operators and functions) assumed by a component, and then providing those interfaces in other components. In the preceding chapter, we discussed our assumptions about the standard operators and their interfaces. However, few interesting components can be written practically using only standard operators. Rather, we must often specify interfaces at a higher level.

Using higher level interfaces provides us with a corresponding increase in the power available from our components, but it also makes it easy to define interfaces which are so specialized that it becomes difficult to use them in a variety of contexts. We can minimize this effect by clearly understanding the purposes served by components, and by designing component interfaces to cleanly satisfy one of a small set of such purposes. This perspective is the core of this chapter, which will discuss a taxonomy of software components, and describe effective techniques for developing and using them in C++.

Before we proceed with the presentation of this taxonomy, it is important to point out what we are not trying to do. The past 25 years have seen attempts to revolutionize programming by reducing all programs to a single conceptual primitive. Functional programming, for example, made everything into a function; the notions of states, addresses, and side effects were taboo. Then, with the advent of object-oriented programming (OOP), functions became taboo; everything became an object (with a state and associated methods).

Far from solving the problems of programming, these reductionist approaches have been distracting and ultimately harmful. Effective programming uses a variety of paradigms, because different parts of the problem have different requirements. For example, an array and a binary search should not be reduced to

a single, fundamental notion. The two are quite different. An array is a data structure -- a component that holds data. A binary search is an algorithm -- a component that performs a computation on data stored in a data structure. As long as a data structure provides an adequate access method, you can use the binary-search algorithm on it. Only by respecting the fundamental differences between arrays and binary searches can efficiency and elegance be simultaneously achieved.

This observation leads directly to the orthogonal decomposition of component space which is at the core of generic programming. We want to understand the differences between components. Only by understanding these can we identify the meaningful similarities which are the basis of our taxonomy. This is the fundamental philosophical failure of the reductionists. By starting with the claim that everything is a function, or an object, and ignoring their differences, they discard all knowledge. A definition that fits everything provides no interesting information. The attempt to reduce everything to a single category is also methodologically wrong. It is as if mathematicians were to start with axioms, and then see if any of the resulting theorems were interesting. Instead, mathematicians start with likely theorems, attempt to prove them, and finally identify their axioms based on the requirements of of their proofs. Similarly, programming methodologies should begin with interesting algorithms, understand them well, and only then attempt to define interfaces that will make them work. *(I don't actually like this analogy. It's pretty weak.)*

Another important failing of the language design community has resulted from a fundamental misunderstanding of addresses and pointers. Many brilliant computer scientists have observed problems with pointers, and come to the conclusion that they should be avoided. Functional programming was motivated partially by the desire to work with values instead of addresses. Object-oriented programming attempts to minimize pointer use by providing access only to the object and not directly to its members. The Ada programming language limited pointer use to objects in the heap, forbidding their use with global (static) and stack objects.

But in fact, pointers are very good. We don't mean dangling pointers. We don't mean pointers to the stack. But the general notion of pointer is a powerful tool, universally used. In programming, as in the real world, we are usually interested in identifying specific objects rather than identifying an object with a particular value. The address of the object is how we identify it. Sometimes we can use the notion of a variable name in lieu of an explicit address, but complex data structures and algorithms to do not lend themselves to explicit naming of each component, depending on the use of anonymous pointers or indices to navigate a uniform space like a vector or a tree.

It is incorrectly believed that pointers make our thinking sequential. That is not so. Without some kind of address we cannot describe any parallel algorithm. If we attempt to describe an addition of n numbers in parallel, we need to be able to talk about the first number being added to the second number, while the third number is added to the fourth number. Some kind of indexing is required. Addresses are fundamental to our conceptualization of any algorithm, sequential or parallel.

Rather than attempt to avoid pointers, we will abstract their useful properties, developing them carefully

in Part III. By abstracting them, we can actually provide a more disciplined usage, which helps to avoid some of the problems which led to the misguided attempts to eliminate pointers.

---

§ 3.2 -- Taxonomy of Software Components

Taxonomy of Types and Data Structures

Taxonomy of Algorithms

---

§ 3.3 -- Function Objects

Writing and using general-purpose algorithms depends on our ability to take the operations defined on an arbitrary type and map them to the specific operations required by a generic algorithm. For example, consider a sort algorithm. Its most fundamental requirement is an operation which compares two elements of the data structure being sorted, and orders them. Although the default "<" operator is often appropriate for this purpose, it is by no means the only interesting ordering operator. Thus, we need a way to provide a specific function with the name expected by the generic algorithm.

Even in C, we can pass a pointer to a function as a parameter to another function, but this is not as general as we would like. It is useful to be able to specify not only the code to be used for a function, but also some state. For example, if we are passing a function which adds two numbers modulo some base, we need to associate the base value with the function. We call this construct a *function object*, because it is an object which incorporates a function's code along with the state data which it requires. The alternatives, i.e. putting the state in global variables referenced by the function, or writing a distinct version of the function for each state of interest, are clearly less desirable.

In C++, our preferred technique for defining and using function objects is derived from the treatment of template functions. When C++ sees a function call which matches a template, it automatically instantiates the template with the required template parameters. However, it must be able to derive those parameters' types from the types of the function call parameters. Consider the following code fragment:

```
template < class T, class Compare >
void sort ( T* first, T* last )
{
  Compare comp;
  ...
}
```

The intent of this code fragment is that an instantiation of the template specifies, as template parameters, the type T of the values to be sorted, and the type Compare of a function object which can be used for comparisons. This type Compare can then be used inside sort to declare the function object *comp* that is actually used for comparisons. This approach has the apparent advantage that the call to sort need not pass a parameter for the Compare function object type.

However, if the application calls this template function for sort, the compiler cannot deduce the desired Compare type from the parameter list, which specifies only the first and last positions to be sorted. Instead, then, we use a declaration like the following, passing the function object as an explicit parameter:

```
template < class T, class Compare >
void sort ( T* first, T* last, Compare comp )
{
  ...
}
```

Although this version requires an additional parameter to the sort function, this is less of a concern than it seems. If the Compare class contains no data members, serving only to identify the comp function it encapsulates, an effective compiler can ignore the parameter in the instantiation of the sort function. (This is also why we generally pass function objects by value instead of by reference -- the object itself is generally small enough that avoiding a copy is not worth the cost of dereferencing a pointer.) More importantly, as we shall soon see, the comp function object may contain data members which cannot be derived from its type, in which case the first version of sort above would be unable to make use of it. For both of these reasons, generic function should generally include formal function object parameters for any function on which it as generic dependencies, unless the function type can be unambiguously derived from the other parameters.

With this context in mind, let us consider the appropriate interface for the Compare function object itself. We could require that the Compare class have a comparison function as a member function with a particular name. But C++ gives us a means to avoid adding another name to the specification -- we can define an application operator, which allows us to simply treat the function object's name as the function name:

```
// Define a comparison function using <
template < class T >
struct less
{
  ...
  bool operator() ( const T& x, const T& y ) const
  {
    return x < y;
  }
};

// Instantiate and use a function less:
int i[100];
...
sort ( i, i+100, less<int>() );
```

This simple function object exhibits some interesting points. First, notice how we pass the function object in our call to sort. The syntax, *less<int>()*, specifies an instantiation of the *less* class for type T as int with the string "less<int>", and the parentheses following it indicate the invocation of a default constructor to create a concrete object of that type as the actual parameter.

Next, consider the declaration of the formal parameters to operator() as *const T&*. While we could pass them by value, this could be much more expensive for large types T, or for types T with expensive copy constructors. This is particularly true in cases where the "<" operator for T ends up referencing a small part of the object. On the other hand, if the type T is a simple type without copy constructors, the compiler is free to actually pass it by copying the value if that is less expensive. With an ideal compiler, therefore, we get the best of both worlds.

To deal with less than ideal compilers, in important cases, it may still be desirable to provide specializations of the function object for common simple cases. For example, we might specialize for *int* as follows:

```
// Specialize template struct less for int:
template <>
struct less<int>
{
  ...
  bool operator() ( int x, int y ) const
  {
    return x < y;
  }
};
```

In this template specialization in scope, an instantiation of *less<int>()* in our call to sort will produce the specialization, passing the parameters by value, instead of instantiating the more general template and possibly passing them by reference.

The examples we have considered so far associate no state with the function object -- the function code requires none. This is not always the case, however. In fact, much of the power of function objects comes from the ability to include state with the function, without passing additional parameters and thereby changing the interface.

As a demonstration, let us consider a different sorting problem. Some algorithms in graphics, for instance convex hulls, start by sorting a list of points in the order of the angle of a line connecting a particular point with each of them in turn. This sorting problem involves a comparison operator with state in the form of the distinguished point. If points are represented as pairs of floats, we might start with the following definition:

```
// Define a point:
struct point
{
  float x_coord;
  float y_coord;

  // Construct a point (x,y):
  point (void) : x_coord(0.0), y_coord(0.0) {}
  point (float x, float y) : x_coord(x), y_coord(y) {}
};
```

Obviously, this is a very simple definition. We represent a point simply as its x and y coordinates in the plane, and provide a default constructor as well as a constructor which creates a point from its coordinates. The default copy constructor and assignment operator, which will simply copy the

coordinate members, are satisfactory for our purposes. In order to provide a full regular type, as we specified in the previous chapter, we should provide global operator== and operator< functions as well, but we will not need them for this simple example, so we omit them here. Observe that this definition is really an instance of the STL pair template. The struct definition above could be replaced simply by:

```
typedef pair<float,float> point;
```

Doing so is simpler, and would automatically provide the missing functions. We provide the above definition just to be explicit.

Now consider how to sort a set of these points in order of their angles from a central point. First, we define a function which returns the angle between the line connecting them and the x-axis:

```
// Angle in radians between points p1 and p2:
inline float angle (const point& p1, const point& p2)
{
  return atan2 (p2.y_coord-p1.y_coord, p2.x_coord-p1.x_coord);
}
```

Finally, we are ready to define our comparison operator, and use it to sort a vector of points:

```
// Define a less-than function based on the angle:
template <>
struct less<point>
{
  point origin;

  // Construct a less function object:
  less<point> (const point& p) : origin(p) {}

  // Compare two points based on their angle:
  bool operator() (const point& p1, const point& p2) const
  {
    return angle (origin, p1) < angle (origin, p2);
  }
};

...
// Sort a list of points relative to origin (x,y):
float x, y;
point origin (x, y);
```

```
less<point> comp (origin);
point p[100];
...
sort ( p, p+100, comp );
...
```

Next, present an example of a function object with state. One possibility is a less than function defined on points in a plane, comparing the distance of two points from a fixed third point; This fits well into the sort framework we have already introduced. A second possibility is a function which tests whether its parameter value is less than some fixed value. This could be used with the find function. It provides the basis for a discussion of the binder function object adapters, and could lead into, or fit into, the next section.

---

§ 3.4 -- Function Object Adapters

---

§ 3.5 -- Binding and Composition

---

§ 3.6 -- Component Traits

---

§ 3.7 -- Categories and Category Dispatching

---

§ 3.8 -- Primitive Recursive Functions

Implementation Queue

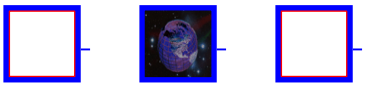
The following are points to be integrated somewhere:

- *Specialization: there are two levels of meaningful function specialization. The first, from functions on concepts (i.e. template functions) to functions on types, is supported by C++, and used extensively in STL. The second, from functions on types to functions on literal values, is not supported by C++. (See language definition chapter.)*

  Any specialization must satisfy the constraint that removing it from a program, if the program remains valid, does not change the results of the program, although it may become less efficient (slower, more resource consumption). That is, a specialization may be richer than the more general component it specializes, but it may not be semantically different.

  (Specializing classes is not supported in C++.)

Send comments about this page to Jim Dehnert or Alex Stepanov.



Generic Programming

Part II: Algorithms on Algebraic Structures

Chapter 4: Arithmetic

Last updated 17 June 1998

---

§ 4.1 -- Introduction

Send comments about this page to Jim Dehnert, John Wilkinson, or Alex Stepanov.

§ 4.1 -- Power: The Russian Peasant Algorithm

An operation of great utility, one that is supported directly by many languages, is that of raising a number to a positive integer power. In this section we discuss this algorithm generically.

For any set with a binary operation *, we can define a power function inductively by

power(x, 1) = x
power(x, n) = power(x, n-1) * x if n > 1

For a general binary operation, however, this power function is not very interesting: if, for example, we defined it with the operation applied in the other order, we would in general get an entirely different operation. The power function becomes interesting when we require the operation to be *associative*. If * is associative, we can show easily by induction the familiar *law of exponents*:

power(x, m+n) = power(x, m) * power(x, n)

As we shall see, this property not only makes the power function interesting; it also makes it possible to compute it efficiently.

For a generic power algorithm, it is clear that their are two types involved: the type of n should be some type that can represent natural numbers, and the type of x should be of some type supporting an associative binary operation. Mathematicians call such a set a *semigroup*. This suggests the interface

```
template<class SemigroupElement, class Integer, class SemigroupOperation>
SemigroupElement power(SemigroupElement a,
                       Integer n,
                       SemigroupOperation op) {
  ..............
}
```

(Here as always the template parameter names are merely mnemonics, intended to suggest requirements on the template arguments.)

It is, however, a little more convenient to add an additional requirement on the semigroup, namely that it possess an *identity*, that is an element *e* such that

$$op(x, e) = op(e, x) = x \text{ for every } x.$$

Such a semigroup is known as a *monoid*, and we are not really losing significant generality by restricting ourselves to monoids, since it is a mathematical fact that we can always adjoin an identity element to any semigroup.

This gives the interface

```
template<class MonoidElement, class Integer, class MonoidOperation>
MonoidElement power(MonoidElement a,
                    Integer n,
                    MonoidOperation op) {
  ..............
}
```

Of course only the names have been changed!

We now proceed to try to develop an algorithm.

Let's start by initializing the result:

```
template<class MonoidElement, class Integer, class MonoidOperation>
MonoidElement power(MonoidElement a,
                    Integer n,
                    MonoidOperation op) {

  MonoidElement result = identity_element(op);

  ................

  return result;

}
```

This initialization accomplishes something very important: if a0 is the initial value of a and n0 is the initial value of n, we have established the equality op(result, power(a, n)) = power(a0, n0). If we can reduce n to zero keeping this relation invariant, we will be sure of having a correct algorithm.

This suggests the following straightforward solution:

```
template<class MonoidElement, class Integer, class MonoidOperation>
MonoidElement power(MonoidElement a,
                    Integer n,
                    MonoidOperation op) {
  MonoidElement result = identity_element(op);

  while (n != 0) {
    result = op(result, a);
    --n;
  }

  return result;
}
```

This is correct, but not very efficient: it has linear complexity in n and would be quite unsatisfactory for very large values of n. Note that this algorithm uses only the inductive definition of the power function. It does not exploit the law of exponents, and could be used perfectly well for arbitrary binary operations. Applying the law of exponents, however, allows a major improvement. The identity

$$power(x, m+n) = op(power(x, m), power(x, n))$$

when m is equal to n gives

$$power(x, 2n) = square(power(x, n), op)$$

where we define square(x, op) to be op(x, x).

This means that if n is even, we can maintain the invariant with a much greater reduction in n: we can square a and halve n. This observation leads to the improved algorithm

```
template<class MonoidElement, class Integer, class MonoidOperation>
MonoidElement power(MonoidElement a,
                    Integer n,
                    MonoidOperation op) {
  MonoidElement result = identity_element(op);
  while (n != 0) {

    if (is_even(n)) {
      square(a, op);
      halve(n);
    }
    else {

      result = op(result, a);
      --n;
    }
  }
  return result;
}
```

This version has logarithmic complexity. Probably we're not going to achieve dramatic improvement from this point, but let's see if we can squeeze it a bit.

Note that in the *else* branch, once we have reduced n by 1 we know it is even, so we have a redundant test in

the next iteration. We might as well fold it in now. This gives us

```
template<class MonoidElement, class Integer, class MonoidOperation>
MonoidElement power(MonoidElement a,
                    Integer n,
                    MonoidOperation op) {
  MonoidElement result = identity_element(op);
  while (n != 0) {
    if (is_even(n)) {
      square(a, op);
      halve(n);
    }
    else {
      result = op(result, a);
      --n;

      if (n != 0) {
        square(a, op);
        halve(n);
      }

    }
  }
  return result;
}
```

Now if n is odd, the combination

```
--n;
halve(n);
```

is equivalent to just

```
halve(n);
```

so we can simplify this to

```
template<class MonoidElement, class Integer, class MonoidOperation>
MonoidElement power(MonoidElement a,
                    Integer n,
                    MonoidOperation op) {
  MonoidElement result = identity_element(op);
  while (n != 0) {
    if (is_even(n)) {
      square(a, op);
      halve(n);
    }
    else {
      result = op(result, a);

      if (n != 1) {

        square(a, op);
        halve(n);
      }
    }
  }
  return result;
}
```

And the comparison against 1 can be eliminated by separating out the last iteration, provided we are a little careful: if n is not zero initiallly, then repeated halvings must take it through 1. So let's treat the case n == 0 separately at the beginning:

```
template<class MonoidElement, class Integer, class MonoidOperation>
MonoidElement power(MonoidElement a,
                    Integer n,
                    MonoidOperation op) {

  if (n == 0) return identity_element(op);


  MonoidElement result = identity_element(op);

  while (n != 1) {
```

```
    if (is_even(n)) {
      square(a, op);
      halve(n);
    }
    else {
      result = op(result, a);
      square(a, op);
      halve(n);
    }
  }

  result = op(result, a);

  return result;
}
```

But we can do better if we note that whenver n is non-zero and even, it is wasteful to check for both n non-zero and n even, since repeating halving must eventually make n odd. Applying this observation gives us

```
template<class MonoidElement, class Integer, class MonoidOperation>
MonoidElement power(MonoidElement a,
                    Integer n,
                    MonoidOperation op) {
  if (n == 0) return identity_element(op);

  MonoidElement result = identity_element(op);
  while (n != 1) {
    if (is_even(n)) {
      square(a, op);
      halve(n);
    }
    else {
      result = op(result, a);

      do {
        square(a, op);
        halve(n);
      } while (is_even(n));

    }
  }
```

```
    result = op(result, a);
    return result;
}
```

But now the inner loop always terminates with n odd. If we could get n odd at the beginning of the first iteration, we could eliminate the is_even test altogether. But we can use our previous observation again to guarantee this:

```
template<class MonoidElement, class Integer, class MonoidOperation>
MonoidElement power(MonoidElement a,
                    Integer n,
                    MonoidOperation op) {
  if (n == 0) return identity_element(op);

  MonoidElement result = identity_element(op);

  while (is_even(n)) {
    halve(n);
    square(a, op);
  }


  while (n != 1) {
    result = op(result, a);
    do {
      square(a, op);
      halve(n);
    } while (is_even(n));
  }

  result = op(result, a);
  return result;
}
```

Now either the first iteration of the outer loop or the instruction following the loop has a multiplication by the identity. This can be eliminated if we assign a to result just before the loop and move the multiplication to the end of the loop:

```
template<class MonoidElement, class Integer, class MonoidOperation>
MonoidElement power(MonoidElement a,
                    Integer n,
                    MonoidOperation op) {
  if (n == 0) return identity_element(op);

  while (is_even(n)) {
    halve(n);
    square(a, op);
  }

  MonoidElement result = a;

  while (n != 1) {
    do {
      square(a, op);
      halve(n);
    } while (is_even(n));

    result = op(result, a);
  }

  return result;
}
```

§ 4.3 -- Greatest Common Divisor

The most important fact about the greatest common divisor d of two integers x and y is that there always exist integers u and v for which

$$ux + vy = d \quad (1)$$

In some applications, finding these integers is more important than finding the gcd. In particular, if gcd(x, y) = 1, then finding u and v satisfying (1) solves the problem of finding a multiplicative inverse for x modulo y or for y modulo x.

Fortunately, solving this problem is not much more difficult that finding the gcd, and we can extend the Euclidean algorithm to provide a solution at very little additional cost. We call this the *Extended Euclidean*

Algorithm.

Given x and y, we want to find u and v such that

```
(1)   ux + vy = d
```

where d is the gcd of x and y. An obvious approach is to replace d by a variable r, find some initial value of r for which

```
(1)   ux + vy = r
```

is easy to solve, and then find a way to decrease r while maintaining the equation (2).

Now there are two initial values for r for which (2) is easy to solve, namely x and y, for which we have the solutions

```
u = 1, v = 0
```

and

```
u = 0, v = 1
```

respectivley.

So let's introduce variables r0, u0, v0, r1, u1, v1, and initialize them as follows:

```
r0 = x; u0 = 1; v0 = 0;
r1 = y; u1 = 0; v1 = 1;
```

so that we have the equations

```
(3) u0 * x + v0 * y == r0
```

and

```
(3) u1 * x + v1 * y == r1
```

Now we can get a simliar equation with a smaller r by setting

```
q = r0/r1;

r = r0 - q * r1;
u = u0 = q * u1;
v = v0 - q * v1;
```

and we have

```
ux + vy = r, where r < r1.
```

as we see simply by multiplying equation (4) by q and subtracting from equation (3).

This leads to the following code:

```
template <class EuclideanRingElement>
triple <EucldeanRingElement, EuclideanRingElement, EuclideanRingElement>
extended_gcd(const EuclideanRingElement & x,
             const EuclideanRingElement & y) {
  if (x == 0) return make_triple(0, 1, y);
  if (y == 0) return make_triple(1, 0, x);

  EuclideanRingElement r0 = x, u0 = 1, r1 = y, u1 = 0, v1 = 1;

  while (r1 != 0) {
    EuclideanRingElement q = r0 / r1;
    EuclideanRingElement r = r0 - q * r1;
    EuclideanRingElement u = u0 - q * u1;
    EuclideanRingElement v = v0 - q * v1;
    r0 = r1; u0 = u1; v0 = v1;
    r1 = r;  u1 = u;  v0 = v;
  }

  return make_triple(u0, v0, r0);
}
```

Now let's see if we can optimize this a little. We notice that the variables v0, v1, v are not used in the

computation of the other variables. They are just going along for the ride, so to speak, except that v0 is part of the final result. Their usefulness is that they let us see we are maintaining the invariants throughout the computation.

But we can simply compute v0 after we exit the loop, since it is just the solution of

```
u0 * x + v0 * y == r0
```

Eliminating the v's from the code gives us

```
template <class EuclideanRingElement>
triple <EucldeanRingElement, EuclideanRingElement, EuclideanRingElement>
extended_gcd(const EuclideanRingElement & x,
             const EuclideanRingElement & y) {
  if (x == 0) return make_triple(0, 1, y);
  if (y == 0) return make_triple(1, 0, x);

  EuclideanRingElement r0 = x, r1 = y, u1 = 0;

  while (r1 != 0) {
    EuclideanRingElement q = r0 / r1;
    EuclideanRingElement r = r0 - q * r1;
    EuclideanRingElement u = u0 - q * u1;
    r0 = r1; u0 = u1;
    r1 = r;  u1 = u;
  }

  return make_triple(u0, (r0 - y0 * x) / y, r0);
}
```

Generic Programming

Part II: Algorithms on Algebraic Structures

Chapter 5: Algebra and Number Theory

Last updated 29 December 1997

Send comments about this page to Jim Dehnert or Alex Stepanov.

Generic Programming

Part II: Algorithms on Algebraic Structures

Chapter 6: Encryption

Last updated 29 December 1997

Send comments about this page to Jim Dehnert or Alex Stepanov.

Generic Programming

Part III: Algorithms on Sequential Structures

Chapter 7: Iterators

Last updated 06 May 1998

---

§ 7.1 -- Introduction

It is a rare program which does not involve iterating through some data structure. Tradeoffs between the ease of iteration, the data structure memory required to support iteration, and similar issues, can be critical to the performance of a program. Yet in spite of the fact that such iteration is conceptually trivial, its precise implementation can make a significant difference in the form of the code.

As a result, the abstraction of methods for iterating through data is critical to programming generically. In this chapter we will develop a family of such abstractions, identifying requirements, classifying the useful cases, and formally axiomatizing them.

---

§ 7.2 -- Positions in a Sequence

Developing abstractions does not proceed axiomatically from first principles, but must begin with concrete examples. The abstractions with which we deal in generic programming emerged after years of studying such examples. Here we will attempt to telescope this process. Necessarily this will be somewhat artificial, but we hope to capture some of the essence of the process by looking at a few key problems.

One of the most important problems both in computer science literature and in programming practice is *linear search*: given a sequence of values and a test value, we want to find a member of the sequence with value equal to the test value, if such a member exists. Let us look first at a typical solution of this problem. Here the sequence is an array of integers; the test value, naturally, is an integer also. The size of the array is passed as a parameter.

```
int find(int a[], int size, int value) {
  int i;
  for (i = 0; i < size; i++)
    if (a[i] == value) return i;
  return -1;
}
```

This algorithm returns the first value of i for which a[i] is equal to the test value if there is such value, and returns -1 otherwise. This is not too bad; we can distinguish between found and not found, for example, and we return a useful value rather than a bool, so we can tell where we found the value in case we want to do something with the location: we didn't just return "true" or "false."

On the other hand, this function is very specifically adapted to arrays. In order to see if we can get at the essential features of the problem, let's try to make a version that works on singly linked lists and see whether a study of the differences will give as a clue to how to write a generic algorithm.

Suppose then that we have a simple "list-of-integer" type declared as

```
struct list {
  int value;
  list * next;
};
```

(This is far from being a satisfactory definition, but it suffices for our immediate purpose.)

Here we have no indices to return. What should the return type be? We want something that will get us to a particular position in the list, and the obvious choice is "list *." Returning a pointer to an element of the list allows us to get to it, and we can return a null pointer if we don't find an element with value equal to the test value. So we might try something like

```
struct list {
  int value;
  list * next;
};

list * find(list * l, int value) {
  list * p;
  for (p = l; p != NULL; p = p->next)
     if (p->value == value) return p;
  return NULL;
}
```

We notice that in some ways this is more attractive than the array version. For example, the test value for termination is the same as the return value for failure, which thus loses some of its arbitrary character.

We also do not have an extra variable of different type. (It is of course entirely coincidental that the type of i is the same as the type of the test value in the array version.)

In fact, the trouble with the array version is that arrays are too rich a type to make it easy to see the essentials of the problem. They have a subscript operator, an ordering on indices, and so on.

Suppose we try to make the array example more like the list example. Since our goal is a *generic* algorithm, eventually we want to have the *same* code.) The first thing we're going to want to do is to change the return type from int to int *; that is we're going to return a pointer to the array element that's equal to the test value instead of an index into the array.

But if we don't use an index, how are we going to terminate? (The i < size condition). Fortunately, the rules of C and of C++ allow us to refer to the address one past the end of the array, so we can rewrite the function as follows (replacing the complicated "for" construction by the simpler while loop at the same time):

```
int * find(int *a, int size, int value) {
   int * p = a;
   while (p != a + size) {
     if (*p == value) return p;
     ++p;
   }

   return ???
}
```

What should we return? a - 1 is NOT guaranteed to be a valid pointer value. But p is! If it's a + size, then the search failed; otherwise it succeeded.

The size parameter is still awkward. How is it actually used? It's used only in the expression p != a + size, and of course a + size has the same type as p or as a, which suggests the modification

```
int * find(int * a, int * b, int value) {
  int * p = a;
  while (p != b) {
    if (*p == value) return p;
    ++p;
  }
  return p;
}
```

Or, just using a instead of p and renaming the parameters for mnemonic value:

```
int * find(int * first, int * last, int value) {
  while (first != last) {
    if (*first == value) return first;
    ++first;
  }
  return first;
}
```

And now, we're going to return first in any case, so we may simplify further to

```
int * find(int * first, int * last, int value) {
  while (first != last && *first != value) ++first;
  return first;
}
```

Note that in passing to this improved version, besides simplifying the code, we have made it more general, since it can be used to search in any part of an array: it doesn't have to be the whole array.

Meanwhile, we've moved way ahead of the list version. Let's go back and look at it now.

Suppose we use the exact same code except with "int *" replace by "list". This gives us

```
list * find(list * first, list * last, int value) {
  while (first != last && *first != value) ++first;
  return first;
}
```

But this won't do at all, because *first is not an int but a struct, and ++first does not get us a pointer to the next element in the list, but an address sizeof(list) beyond first!

What we need is not a pointer to list, but something that behaves with respect to a list the way a pointer behaves with respect to an array.

Let's call it a list_iterator. What does it need? We know it needs somehow to encapsulate a pointer to list, and it needs to have an operator* returning an int (we're still just dealing with a list of ints), and an operator++ that makes the encapsulated pointer point to the next item on the list. We might try

```
struct list_iterator {
  list * current;
  int operator*() {return current->value;}
  void operator++() {current = current->next;}}
};
```

This is on the right track, but it is not quite adequate. One problem is that our algorthm uses not just '*' and '++' but also '!=', and C and C++ do not define equality and inequality for structs. Therefore we are going to need to supply these operations. (For *find* we need only inequality, but we certainly expect in general to have to use both, and moreover to have them related in the obvious way: more on this later.) A second problem is that our definition provides no way to attach a list_iterator to a list. For this purpose we need a constructor for list_iterator taking a list* parameter. So let's supply the deficiencies:

```
struct list_iterator {
  list * current;
  list_iterator(list * l): current(l) {}
  int operator*() {return current->value;}
  void operator++() {current->next;}}
};

inline bool operator==(const list_iterator& i1, const list_iterator& i2) {
  return i1.current == i2.current;
}

inline bool operator != const list_iterator& i1, const list_iterator& i2) {
  return !(i1 == i2);
}
```

Finally, we rewrite the algorithm as

```
list_iterator find(list_iterator first, list_iterator list, int value) {
  while (first != last && *first != value) ++first;
  return first;
}
```

And this now actually works, as you can easily verify.

Furthermore, we are now ready to write this algorithm in generic form and to isolate the assumptions we need to make about iterators in order for it to work.

```
template <class Iterator, class T>
Iterator find(Iterator first, Iterator list, T value) {
  while (first != last && *first != value) ++first;
  return first;
}
```

This looks plausible, but is it quite right? The code looks good, but what about the interface? One decision that we need to make in writing generic algorithms in C++ is how to declare the parameters? Should we pass by value? by reference? by const reference? In this algorithm all the parameters are being passed by value. Is this right? How do we decide?

Passing a parameter by value means that a copy is generated to do the call. For an arbitrary type T, a copy may be quite expensive. In our algorithm, *value* is not modified in the code, so we are probably better off passing it by const reference rather than by value.

The parameter *first* is modified in the code, so we cannot very well pass it by reference; presumably the user of *find* wants to retain the value passed in and not have it modified. So *first* should definitely be passed by value.

The case of *last* is less clear-cut. It is not modified by the code, so it *could* be passed by const reference. This would, however, make the interface awkwardly asymmetrical. Also, there is a cost associated with passing by reference, since it means passing a pointer instead of a value, and necessitates an extra indirection. So we need to ask ourselves whether the copy of an iterator is likely to be expensive. In our two examples, the iterator copy is cheap: in the array case, the iterator is just a pointer; in the list case, the iterator is an encapsulated pointer: its only data member is just a pointer. And in general we expect an iterator *not* to have a lot of parts. So in almost all the generic algorithms we will study, and in the STL in general, iterator parameters are generally passed by value, not by reference.

The above discussion suggests that we should make just one change to our *find* algorithm: changing the declaration of the *value* parameter so that it is passed by const reference rather than by value. This change gives

us

```
template<class Iterator, class T>
Iterator find(Iterator first, Iterator list, const T& value) {
  while (first != last && *first != value) ++first;
  return first;
}
```

This *function template* has two template parameters, *Iterator* and *T*. We have a pretty good idea by now what requirements we expect of the template arguments that will get substituted for these parameters. We give a hint of what we are expecting of the first parameter by giving it the name "Iterator," but of course the compiler does not understand anything by this name. We are trying to compensate for a deficiency in the language by the use of a mnemonic name, that's all.

The type that we have called "Iterator" must satisfy certain requirements in order for our "find" algorithm to work. C++ does not provide any mechanism within the language for specifying such requirements on the parameters to a template, so such specifications will belong to documentation, which is a very important part of generic programming in C++.

We're not going to get all the requirements right immediately. We'll make a start and refine our definitions as we progress. We're going to have to look at a few algorithms, not just *find* before we really see what is going on.

We can see immediately from the code for the *find* algorithm that *Iterator* must support three operations, '*', '++' and '!='. What are the requirements on these operations?

First of all, when we say the iterator type supports '*' for example, we do not mean that '*' can be applied to everything of that type. If we think of iterators as a generalization of pointers, we see this immediately: we cannot dereference a null pointer, for example. Similarly, we cannot always legitimately increment a pointer. Indeed, if we look at *find*, we are careful to check that first != last *before* we try to dereference first. But if we find that first != last, then we can dereference first and also increment it.

Let us see what we can make of these rambling remarks. First of all, it seems useful to distinguish between *valid* and *invalid* values of an iterator. A valid value of an iterator can always be dereferenced, but a value may be valid even thought it cannot be dereferenced. Such a value we call *past-the-end*. We need to be able to compare to a past-the-end value to make *find* work. Furthermore, if i is dereferenceable, then ++i is valid, and vice versa. Valid values of an iterator can always be compared for equality (or inequality). Furthermore if i and j are iterators with i == j, and i is dereferenceable, the j is dereferenceable, and *i and * j are not just equal, they are the *same object*; that is &*i == &*j. (This holds even if no equality is defined on objects of the type of *i!).

Note that the systematizing impulse has taken on a life of its own here. We get some hints from our algorithm (quit a lot of hints for such a simple algorithm); some from the given properties of C and C++ pointers. But we also need to appeal to common sense, mathematical general principles, and a little imagination. We hope we're

getting things more or less right: if our axioms are too restrictive we'll miss some important applications; if they're too general we won't be able to do anything with them. Getting things right may require a few rounds of refinement as we look at more algorithms.

---

§ 7.3 -- Varieties of Iterators

It's time to try another algorithm. Another thing we do all the time is *copy*: reading a list of numbers into an array, copying one array to another, and printing a list of numbers from an array are all examples of a problem that seems to cry out for a generic algorithm.

By now we should have a good idea of how to start:

```
template<class Iterator>
Iterator copy(Iterator first, Iterator last, Iterator out) {
  while (first != last)
    *out++ = *first++;
  return out;
}
```

using the kind of cryptic and powerful single-instruction loop so beloved of C programmers.

Hey, that was easy! Unfortunately, it's not quite right. Looking at the examples in the opening paragraph, it's clear we can use this to copy one array to another, but not to read numbers into an array or print numbers from an array, because we only have one iterator type here. What we need is one type for input (what we copy from) and one type for output (what we copy to). With this correction, we get

```
template <class Iterator1, class Iterator2>
Iterator2 copy(Iterator1 first, Iterator1 last, Iterator2 out) {
  while (first != last)
    *out++ = *first++;
  return out;
}
```

Note that the *code* was OK; it was the *interface* that was wrong.

Now this is really quite nice. Note the convenience of the return value. It points just beyond the last item copied. So if we want to copy something else at the end, we're all set to do so.

Another nice thing is that our algorithms work well together. Here, for example, is a function to copy the values between the first and second zero in a sequence:

```
template <class Iterator1, class Iterator2>
Iterator2 copy_between_zeros(Iterator1 first,
                             Iterator1 last,
                             Iterator2 out) {
  first = find(first, last, 0);
  if (first == last) return out;
  return copy(++first, find(first, last, 0), out);
}
```

Section: Iterator Axioms and Models

Present Peano axioms and modify.

Section: Iterator Categories

Section: Iterator Traits and Compile-time Dispatch

Section: Iterator Adapters

Reverse iterators and stride iterators are the key examples here.

Send comments about this page to John Wilkinson, Jim Dehnert or Alex Stepanov.

Generic Programming

Part III: Algorithms on Sequential Structures

Chapter 8: Permutations

Last updated 06 May 1998

Send comments about this page to John Wilkinson, Jim Dehnert or Alex Stepanov.

Generic Programming

Part III: Algorithms on Sequential Structures

Chapter 9: Index- and Value-Based Permutation Algorithms

Last updated 15 December 1998

---

§ 9.1 -- Introduction

---

§ 9.2 -- Swapping Ranges

---

§ 9.3 -- Reverse Algorithms

---

§ 9.4 -- Memory-Adaptive Algorithms

---

§ 9.5 -- In-Place Rotation Algorithms

Gries-Mills algorithm, 3-reverse algorithm, and GCD-cycles algorithm

---

§ 9.6 -- Stability of Permutations and Pseudo-Permutations

---

§ 9.7 -- Remove Algorithms

Remove and stable remove

---

§ 9.8 -- Partition Algorithms

Hoare partition, Lomuto partition, stable partition

---

§ 9.9 -- Random Shuffle and Random Selection Algorithms

Lo-Ho algorithm

---

§ 9.10 -- Reduction Algorithms

Reduction, parallel reduction, and binary-counter reduction.

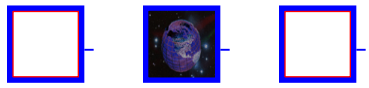
---

§ 9.11 -- NUMA-iterators

Implementation Queue

The following are points to be integrated somewhere:

Send comments about this page to Jim Dehnert Alex Stepanov, or John Wilkinson.



Generic Programming

Part III: Algorithms on Sequential Structures

Chapter 10: Sorting And Related Algorithms

Last updated 29 December 1997

Send comments about this page to Jim Dehnert or Alex Stepanov.

Generic Programming

Part IV: Data Structures and Containers

Chapter: Data Structure Principles

Last updated 29 December 1997

Send comments about this page to Jim Dehnert or Alex Stepanov.

Generic Programming

Part IV: Data Structures and Containers

Chapter: Sequential Data Structures

Last updated 29 December 1997

Send comments about this page to Jim Dehnert or Alex Stepanov.

Generic Programming

Part IV: Data Structures and Containers

Chapter: Associative Data Structures

Last updated 29 December 1997

Send comments about this page to Jim Dehnert or Alex Stepanov.

Generic Programming

Part V: Programming Languages and Software Engineering

Chapter: Language Design for Generic Programming

Last updated 26 March 1998

---

§ x.1 -- Introduction

Implementation Queue

The following are points to be integrated somewhere:

- *Specialization: there are two levels of meaningful function specialization. The first, from functions on concepts (i.e. template functions) to functions on types, is supported by C++, and used extensively in STL. The second, from functions on types to functions on literal values, is not supported by C++.*

  (Specializing classes is not supported in C++.)

- *Function objects: Lexical closures in functional languages implement function objects. C++ definition syntax is not very elegant. For every function (generic or specific) there is a correponding function objec type. It is, however, impossible to obtain this type in C++ without typing in a corresponding definition.*

Send comments about this page to Jim Dehnert or Alex Stepanov.

Generic Programming
Appendices

Appendix A: Glossary

Last updated 23 March 1998

Algorithm
: A precise procedure for performing a calculation -- computing a function or performing an action -- relative to a set of *concepts*. It is written in terms of some fundamental algorithmic language, together with the set of operations provided by the *concepts* that are part of its interface, which terminates for all valid inputs.

Allocator
: A concept which describes memory, via affiliated pointer and reference types, their affiliated size and difference types, and operations for allocating and freeing memory.

Concept
: An interface between algorithms and types, describing a base type, affiliated types, operations on the types, and properties of the types and operations.

Container
: ...

Data Structure
: ...

Function
: The programming language interface to an algorithm, specifying parameter and result concepts (generic) or types (concrete).

Function Object
: A component object (or its type) which combines the code of a function with state used by the function and persisting between calls.

Iterator

...

Object

...

Ordering

...

Part

...

Predicate

...

Regular Type

...

Relation

...

Tuple

...

Type

An interpretation of a set of values, comprising their representation in memory and the set of operations defined on them.

Type Association

A function mapping types to related types, e.g. from a pointer type to the type to which it points.

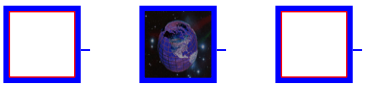
Send comments about this page to Jim Dehnert or Alex Stepanov.

Generic Programming
Appendices

Appendix B: Axioms and Key Principles

Last updated 20 March 1998

Send comments about this page to Jim Dehnert or Alex Stepanov.

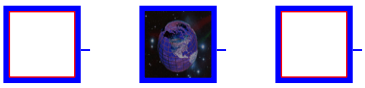


Generic Programming
Appendices

Appendix C: Code Examples

Last updated 20 March 1998

Send comments about this page to Jim Dehnert or Alex Stepanov.



Generic Programming
Appendices

Bibliography

Last updated 23 March 1998

[Backus78]
John Backus, *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, Communications of the ACM 21(8), August 1978.

[Euclid00]
Euclid, **The Elements**, ca. 300 B.C. (English edition translated with introduction and commentary by Sir Thomas L. Heath, vols. 1-3, Dover Publications, 1956.)

[Gauss01]
Carl Friedrich Gauss, **Disquisitiones Arithmeticae**, 1801. (English edition translated by Arthur A. Clarke, Springer-Verlag, 1986.)

[Heath21]
Sir Thomas L. Heath, **A History of Greek Mathematics**, Volumes 1 and 2, 1921. Dover Publications edition, 1981.

[KaMS81a]
Deepak Kapur, David R. Musser, and Alexander A. Stepanov, *Operators and Algebraic Structures*, Proc. of the Conf. on Functional Programming Languages and Computer Architecture, New Hampshire, Oct. 1981. Also Tech. Report 81CRD114, GE Corporate Research and Development, August 1981.

[KaMS81b]
Deepak Kapur, David R. Musser, and Alexander A. Stepanov, *Tecton: A Language for Manipulating Generic Objects*, Tech. Report 9681, GE Corporate Research and Development, 1981.

[Knuth97]

Donald E. Knuth, **The Art of Computer Programming, Vol. 1, Fundamental Algorithms**, Third Ed., Addison Wesley, 1997.

[Knuth98a]
Donald E. Knuth, **The Art of Computer Programming, Vol. 2, Seminumerical Algorithms**, Third Ed., Addison Wesley, 1998.

[Knuth73]
Donald E. Knuth, **The Art of Computer Programming, Vol. 3, Sorting and Searching**, Addison Wesley, 1973.

[Shen95]
Alexander Shen, **Algorithms and Programming: Problems and Solutions**, 1995. (English edition, Birkhäuser, Boston, 1997.)

Send comments about this page to Jim Dehnert or Alex Stepanov.