Generic programming

Define software components with type parameters

A sorting algorithm has the same structure, regardless of the types being sorted.
Stack primitives have the same semantics, regardless of the objects stored on the stack.

Most common use: algorithms on containers: updating, iteration, search

- C model: macros (textual substitution)
- Ada model: generic units and instantiations
- C++, Java C# models: templates
## Parametrizing software components

<table>
<thead>
<tr>
<th>Construct</th>
<th>Parameter</th>
<th>Supplying Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>values (bounds)</td>
<td>declaring object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>declaring subtype</td>
</tr>
<tr>
<td>record</td>
<td>discriminant</td>
<td>object/subtype</td>
</tr>
<tr>
<td>subprogram</td>
<td>values (actuals)</td>
<td>calling subprogram</td>
</tr>
<tr>
<td>generic unit</td>
<td>types, values</td>
<td>instantiating</td>
</tr>
<tr>
<td>template</td>
<td>classes, values</td>
<td>instantiating, specializing</td>
</tr>
</tbody>
</table>
Generics in Ada95

I/O for integer types. Identical implementation, but need separate procedures for strong-typing reasons.

generic

  type Elem is range <> ; -- any integer type

package Integer_IO is

  procedure Put (Item : Elem);

  ...
A generic Package

generic
    type Elem is private;             -- parameter
package Stacks is
    type Stack is private;
    procedure Push (X : Elem; On : in out Stack);
...
private
    type Cell;
    type Stack is access Cell;       -- linked list representation
    type Cell is record
        Val : Elem;
        Next : Ptr;
    end record;
end Stacks;
Instantiations

with Stacks;

procedure Test_Stacks is
  package Int_Stack is new Stacks (Integer);  -- list of integers
  package Float_Stack is new Stacks (Float);  -- list of floats
  S1 : Int_Stack.Stack;                        -- a stack object
  S2 : Float_Stack.Stack;
  use Int_Stack, Float_Stack;                  -- ok, regular packages
begin
  Push (15, S1);                               -- Int_Stack.Push
  Push (3.5 * Pi, S2);
  ...
end Test_Stacks;
Type parameters

- `type T is private;`  --- any type with assignment (Non-limited)
- `type T is limited private;`  -- any type (no required operations)
- `type T is range <>;`  --- any integer type (arithmetic operations)
- `type T is (<>);`  --- any discrete type (enumeration or integer)
- `type T is digits <>;`  --- any floating-point type
- `type T is delta <>;`  --- any fixed-point type

- Within the generic, the operations that apply to any type of the class can be used.
- The instantiation must use a specific type of the class
A generic function

generic
    type T is range <>; -- parameter of some integer type
    type Arr is array (Integer range <>) of T; -- parameter is array of those

function Sum_Array (A : Arr) return T;

-- Body identical to non-generic version
function Sum_array (A : Arr) return T is
    Result : T := 0; -- some integer type, so literal 0 is legal
begin
    for J in A’range loop -- some array type, so attribute is available
        Result := Result + A (J); -- some integer type, so “+” available.
    end loop;
    return Result;
end;
Instantiating a generic function

```plaintext
type Apple is range 1 .. 2 **15 - 1;
type Production is array (1..12) of Apple;

type Sick_Days is range 1..5;
type Absences is array (1 .. 52) of Sick_Days;

function Get_Crop is new Sum_array (Apple, Production);
function Lost_Work is new
    Sum_array (Sick_Days, Absences);
```
generic private types

• Only available operations are assignment and equality.

generic
type T is private;
procedure Swap (X, Y : in out T);

procedure Swap (X, Y : in out T) is
    Temp : constant T := X;  -- no other properties of T
    -- are needed
begin
    X := Y;
    Y := Temp;
end Swap;
A generic sorting routine should apply to any array whose components are comparable, i.e. for which an ordering predicate exists. This class includes more than the numeric types:

```
generic
  type T is private; -- parameter
  with function "<" (X, Y : T) return Boolean; -- parameter
  type Arr is array (Integer range <>) of T; -- parameter
procedure Sort (A : in out Arr);
```
Supplying subprogram parameters

- The actual must have a matching signature, not necessarily the same name:

  ```
  procedure Sort_Up is new Sort (Integer, "<", ...);
  procedure Sort_Down is new Sort (Integer, ">", ...);
  
  type Employee is record .. end record;
  function Senior (E1, E2 : Employee) return Boolean;
  function Rank is new Sort (Employee, Senior, ...);
  ```
Value parameters

Useful to parametrize containers by size:

```plaintext
generic
  type Elem is private; -- type parameter
  Size : Positive; -- value parameter
package Queues is
  type Queue is private;
  procedure Enqueue (X : Elem; On : in out Queue);
  procedure Dequeue (X : out Elem; From : in out Queue);
  function Full (Q : Queue) return Boolean;
  function Empty (Q : Queue) return Boolean;
private
  type Contents is array (Natural range <>) of Elem;
  type Queue is record
    Front, Back: Natural;
    C : Contents (0 .. Size);
  end record;
end Queues;
```
Packages as parameters

```
generic
type Real is digits <>; -- any floating type
package generic_complex_types is
   -- complex is a record with two real components
   -- package declares all complex operations: +, -, Re, Im...
end generic_complex_types;
```

We also want to define a package for elementary functions (sin, cos, etc.) on complex numbers. We need the complex operations, which are parametrized by the corresponding real value.
The instantiation requires an instance of the package parameter

```pascal
with generic_complex_types;

generic
    with package compl is new generic_complex_types (<>);

package generic_complex_elementary_functions is
    -- trigonometric, exponential, hyperbolic functions.
end generic_complex_elementary_functions;
```

- Instantiate complex types with long_float components:

```pascal
package long_complex is new generic_complex_type (long_float);
```

- Instantiate complex functions for long_complex types:

```pascal
package long_complex_functions is
    new generic_complex_elementary_functions (long_complex);
```
Templates in C++

template <class T> class Vector {
public:
    Vector (size_t n); // constructor
    T& operator [ ] (size_t); // subscript (hopefully with checks!)

private:
    ... // a struct with a size and a pointer to an array
};

Vector<int> V1 (100); // instantiation
Vector <int> Vdef; // use default constructor
typedef Vector<employee> Dept; // named instance
Class and value parameters

template <class T, int i> class Buffer {
    T v[i];  // storage for buffer
    int sz;  // total capacity
    int count;  // current contents

public:
    Buffer() : sz(i), count(0) { };
    T read();
    void write(T elem);
};

Buffer <Shape, 100> picture;  // instance for a graphics app.
Template hopes for the best

template <class T> class List {
    struct Link {  // for a list node
        Link* pre;  // doubly linked
        Link* succ;
        T val;
        Link (Link* p, s, const T& v): pre (p), succ(s), val (v) { };
    }
    Link * head;
    public:
    void print_all () { for Link*p = head; p; p=p->succ)
        cout << p -> val << \n        // operator<< better exist for T!
}
Function templates

```cpp
template <class T> void sort (vector <T>&) { .. };

void testit (<vector<int>&vi) {
    sort (vi); // implicit instantiation
    // can also write sort<int> (vi);
}
```
Functions and function templates

- Templates and regular functions overload each other:

  ```cpp
  template <class T> class complex {...};
  
  template <class T> T sqrt (T);                -- template
  template <class T> complex <T> sqrt (complex <T>);    -- different algorithm
  
  double sqrt (double);                        -- regular function
  void testit (complex<double> cd) {
    sqrt (2);       // sqrt<int>: specialization of template
    sqrt (2.0);     // sqrt (double): regular function
    sqrt (cd);      // sqrt(<double>(complex<double>)): specialization
  }
  ```
Iterators and containers

Collections

- insert, delete, search, count

Standard algorithms over collections use iterators:

```cpp
#include <collections/iterators.hpp>

// Example of a template iterator for a collection.
// This iterator allows you to perform operations on elements.

template <class Coll, class Elem> class Iterator {
private:
    Coll* coll;
    Elem* elem;
public:
    Iterator(Coll* c) : coll(c), elem(c->begin()) {} // Constructor
    Iterator& operator++() { // Increment iterator
        ++elem;
        return *this;
    }
    bool done() const { // Check if iterator is done
        return elem == coll->end();
    }
    // Other iterator methods...
};
```
The Standard Template Library

A set of useful data structures and algorithms, mostly to handle collections.

**Sequential containers:** List, vector, deque

**Associative containers:** set, map

**Iterators provided:**
- `vector<T>::iterator`
- `vector<T>::const_iterator`
- `vector<T>::reverse_iterator`
- `vector<T>::const_reverse_iterator`
Generics in Java

Only class parameters
Implementation by type erasure: all instances share code

```java
interface Collection <E> {
    public void add (E x);
    public Iterator<E> iterator ();
}
```

Collection <Thing> is a parametrized type
Collection (by itself) is a raw type
Generic methods in Java

class Collection <A> {
    public static <A extends Comparable <A>>
        max (Collection <A> xs) {
            Iterator <A> xi = xs.iterator();
            A biggest = xi.next();
            while (xi.hasNext()) {
                A x = x.next();
                if (w.compareTo(x) < 0) biggest = x;
            }
            return biggest;
        }