

Generic programming

Generic programming is a style of computer programming in which algorithms are written in terms of types *to-be-specified-later* that are then *instantiated* when needed for specific types provided as parameters. This approach, pioneered by ML in 1973,^{[1][2]} permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication. Such software entities are known as *generics* in Ada, C#, Delphi, Eiffel, F#, Java, Rust, Swift, TypeScript and Visual Basic .NET. They are known as *parametric polymorphism* in ML, Scala, Haskell (the Haskell community also uses the term "generic" for a related but somewhat different concept) and Julia; *templates* in C++ and D; and *parameterized types* in the influential 1994 book *Design Patterns*.^[3] The authors of *Design Patterns* note that this technique, especially when combined with delegation, is very powerful, however,

Dynamic, highly parameterized software is harder to understand than more static software.

— Gang of Four, *Design Patterns*^[3] (Chapter 1)

The term **generic programming** was originally coined by David Musser and Alexander Stepanov^[4] in a more specific sense than the above, to describe a programming paradigm whereby fundamental requirements on types are abstracted from across concrete examples of algorithms and data structures and formalized as concepts, with generic functions implemented in terms of these concepts, typically using language genericity mechanisms as described above.

Contents

Stepanov–Musser and other generic programming paradigms

Programming language support for genericity

In object-oriented languages

Generics in Ada

Example

Advantages and limitations

Templates in C++

Technical overview

Template specialization

Advantages and disadvantages

Templates in D

Code generation

Genericity in Eiffel

Basic/Unconstrained genericity

Constrained genericity

Generics in Java

Genericity in .NET [C#, VB.NET]

Genericity in Delphi

Genericity in Free Pascal

Functional languages

Genericity in Haskell

PolyP

Generic Haskell

[Clean](#)[Other languages](#)[See also](#)[References](#)[Citations](#)[Further reading](#)[External links](#)

Stepanov–Musser and other generic programming paradigms

Generic programming is defined in [Musser & Stepanov \(1989\)](#) as follows,

Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software.

— Musser, David R.; Stepanov, Alexander A., [Generic Programming](#)^[5]

Generic programming paradigm is an approach to software decomposition whereby fundamental requirements on types are abstracted from across concrete examples of algorithms and data structures and formalized as [concepts](#), analogously to the abstraction of algebraic theories in [abstract algebra](#).^[6] Early examples of this programming approach were implemented in Scheme and Ada,^[7] although the best known example is the [Standard Template Library \(STL\)](#),^{[8][9]} which developed a theory of [iterators](#) that is used to decouple sequence data structures and the algorithms operating on them.

For example, given N sequence data structures, e.g. singly linked list, vector etc., and M algorithms to operate on them, e.g. `find`, `sort` etc., a direct approach would implement each algorithm specifically for each data structure, giving $N \times M$ combinations to implement. However, in the generic programming approach, each data structure returns a model of an iterator concept (a simple value type that can be dereferenced to retrieve the current value, or changed to point to another value in the sequence) and each algorithm is instead written generically with arguments of such iterators, e.g. a pair of iterators pointing to the beginning and end of the subsequence to process. Thus, only $N + M$ data structure-algorithm combinations need be implemented. Several iterator concepts are specified in the STL, each a refinement of more restrictive concepts e.g. forward iterators only provide movement to the next value in a sequence (e.g. suitable for a singly linked list or a stream of input data), whereas a random-access iterator also provides direct constant-time access to any element of the sequence (e.g. suitable for a vector). An important point is that a data structure will return a model of the most general concept that can be implemented efficiently—[computational complexity](#) requirements are explicitly part of the concept definition. This limits the data structures a given algorithm can be applied to and such complexity requirements are a major determinant of data structure choice. Generic programming similarly has been applied in other domains, e.g. graph algorithms.^[10]

Note that although this approach often utilizes language features of [compile-time](#) genericity/templates, it is in fact independent of particular language-technical details. Generic programming pioneer Alexander Stepanov wrote,

Generic programming is about abstracting and classifying algorithms and data structures. It gets its inspiration from Knuth and not from type theory. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures. Such an undertaking is still a dream.

— Alexander Stepanov, Short History of STL ^{[11][12]}

I believe that iterator theories are as central to Computer Science as theories of rings or Banach spaces are central to Mathematics.

— Alexander Stepanov, An Interview with A. Stepanov^[13]

Bjarne Stroustrup noted,

Following Stepanov, we can define generic programming without mentioning language features: Lift algorithms and data structures from concrete examples to their most general and abstract form.

— Bjarne Stroustrup, Evolving a language in and for the real world: C++ 1991-2006^[12]

Other programming paradigms that have been described as generic programming include *Datatype generic programming* as described in "Generic Programming — an Introduction".^[14] The *Scrap your boilerplate* approach is a lightweight generic programming approach for Haskell.^[15]

In this article we distinguish the high-level programming paradigms of *generic programming*, above, from the lower-level programming language *genericity mechanisms* used to implement them (see Programming language support for genericity). For further discussion and comparison of generic programming paradigms, see.^[16]

Programming language support for genericity

Genericity facilities have existed in high-level languages since at least the 1970s in languages such as ML, CLU and Ada, and were subsequently adopted by many object-based and object-oriented languages, including BETA, C++, D, Eiffel, Java, and DEC's now defunct Trellis-Owl language.

Genericity is implemented and supported differently in various programming languages; the term "generic" has also been used differently in various programming contexts. For example, in Forth the compiler can execute code while compiling and one can create new *compiler keywords* and new implementations for those words on the fly. It has few *words* that expose the compiler behaviour and therefore naturally offers *genericity* capacities that, however, are not referred to as such in most Forth texts. Similarly, dynamically typed languages, especially interpreted ones, usually offer *genericity* by default as both passing values to functions and value assignment are type-indifferent and such behavior is often utilized for abstraction or code terseness, however this is not typically labeled *genericity* as it's a direct consequence of dynamic typing system employed by the language. The term has been used in functional programming, specifically in Haskell-like languages, which use a structural type system where types are always parametric and the actual code on those types is generic. These usages still serve a similar purpose of code-saving and the rendering of an abstraction.

Arrays and structs can be viewed as predefined generic types. Every usage of an array or struct type instantiates a new concrete type, or reuses a previous instantiated type. Array element types and struct element types are parameterized types, which are used to instantiate the corresponding generic type. All this is usually built-in in the compiler and the syntax differs from other generic constructs. Some extensible programming languages try to unify built-in and user defined generic types.

A broad survey of genericity mechanisms in programming languages follows. For a specific survey comparing suitability of mechanisms for generic programming, see.^[17]

In object-oriented languages

When creating container classes in statically typed languages, it is inconvenient to write specific implementations for each datatype contained, especially if the code for each datatype is virtually identical. For example, in C++, this duplication of code can be circumvented by defining a class template:

```
template<typename T>
class List
{
    /* class contents */
};

List<Animal> list_of_animals;
List<Car> list_of_cars;
```

Above, T is a placeholder for whatever type is specified when the list is created. These "containers-of-type-T", commonly called templates, allow a class to be reused with different datatypes as long as certain contracts such as subtypes and signature are kept. This genericity mechanism should not be confused with inclusion polymorphism, which is the algorithmic usage of exchangeable sub-classes: for instance, a list of objects of type `Moving_Object` containing objects of type `Animal` and `Car`. Templates can also be used for type-independent functions as in the Swap example below:

```
template<typename T>
void Swap(T & a, T & b) //"&" passes parameters by reference
{
    T temp = b;
    b = a;
    a = temp;
}

string hello = "World!"; world = "Hello, ";
Swap( world, hello );
cout << hello << world << endl; //Output is "Hello, World!"
```

The C++ template construct used above is widely cited as the genericity construct that popularized the notion among programmers and language designers and supports many generic programming idioms. The D programming language also offers fully generic-capable templates based on the C++ precedent but with a simplified syntax. The Java programming language has provided genericity facilities syntactically based on C++'s since the introduction of J2SE 5.0.

C# 2.0, Oxygene 1.5 (also known as Chrome) and Visual Basic .NET 2005 have constructs that take advantage of the support for generics present in the Microsoft .NET Framework since version 2.0.

Generics in Ada

Ada has had generics since it was first designed in 1977–1980. The standard library uses generics to provide many services. Ada 2005 adds a comprehensive generic container library to the standard library, which was inspired by C++'s standard template library.

A *generic unit* is a package or a subprogram that takes one or more *generic formal parameters*.

A *generic formal parameter* is a value, a variable, a constant, a type, a subprogram, or even an instance of another, designated, generic unit. For generic formal types, the syntax distinguishes between discrete, floating-point, fixed-point, access (pointer) types, etc. Some formal parameters can have default values.

To *instantiate* a generic unit, the programmer passes *actual* parameters for each formal. The generic instance then behaves just like any other unit. It is possible to instantiate generic units at run-time, for example inside a loop.

Example

The specification of a generic package:

```
generic
  Max_Size : Natural; -- a generic formal value
  type Element_Type is private; -- a generic formal type; accepts any nonlimited type
package Stacks is
  type Size_Type is range 0 .. Max_Size;
  type Stack is limited private;
  procedure Create (S : out Stack;
    Initial_Size : in Size_Type := Max_Size);
  procedure Push (Into : in out Stack; Element : in Element_Type);
  procedure Pop (From : in out Stack; Element : out Element_Type);
  Overflow : exception;
  Underflow : exception;
private
  subtype Index_Type is Size_Type range 1 .. Max_Size;
  type Vector is array (Index_Type range <>) of Element_Type;
  type Stack (Allocated_Size : Size_Type := 0) is record
    Top : Index_Type;
    Storage : Vector (1 .. Allocated_Size);
  end record;
end Stacks;
```

Instantiating the generic package:

```
type Bookmark_Type is new Natural;
-- records a location in the text document we are editing

package Bookmark_Stacks is new Stacks (Max_Size => 20,
  Element_Type => Bookmark_Type);
-- Allows the user to jump between recorded locations in a document
```

Using an instance of a generic package:

```
type Document_Type is record
  Contents : Ada.Strings.Unbounded.Unbounded_String;
  Bookmarks : Bookmark_Stacks.Stack;
end record;

procedure Edit (Document_Name : in String) is
  Document : Document_Type;
begin
  -- Initialise the stack of bookmarks:
  Bookmark_Stacks.Create (S => Document.Bookmarks, Initial_Size => 10);
  -- Now, open the file Document_Name and read it in...
end Edit;
```

Advantages and limitations

The language syntax allows precise specification of constraints on generic formal parameters. For example, it is possible to specify that a generic formal type will only accept a modular type as the actual. It is also possible to express constraints *between* generic formal parameters; for example:

```
generic
  type Index_Type is (<>); -- must be a discrete type
  type Element_Type is private; -- can be any nonlimited type
  type Array_Type is array (Index_Type range <>) of Element_Type;
```

In this example, `Array_Type` is constrained by both `Index_Type` and `Element_Type`. When instantiating the unit, the programmer must pass an actual array type that satisfies these constraints.

The disadvantage of this fine-grained control is a complicated syntax, but, because all generic formal parameters are completely defined in the specification, the compiler can instantiate generics without looking at the body of the generic.

Unlike C++, Ada does not allow specialised generic instances, and requires that all generics be instantiated explicitly. These rules have several consequences:

- the compiler can implement *shared generics*: the object code for a generic unit can be shared between all instances (unless the programmer requests inlining of subprograms, of course). As further consequences:
 - there is no possibility of code bloat (code bloat is common in C++ and requires special care, as explained below).
 - it is possible to instantiate generics at run-time, as well as at compile time, since no new object code is required for a new instance.
 - actual objects corresponding to a generic formal object are always considered to be non-static inside the generic; see Generic formal objects in the Wikibook for details and consequences.
- all instances of a generic being exactly the same, it is easier to review and understand programs written by others; there are no "special cases" to take into account.
- all instantiations being explicit, there are no hidden instantiations that might make it difficult to understand the program.
- Ada does not permit "template metaprogramming", because it does not allow specialisations.

Templates in C++

C++ uses templates to enable generic programming techniques. The C++ Standard Library includes the Standard Template Library or STL that provides a framework of templates for common data structures and algorithms. Templates in C++ may also be used for template metaprogramming, which is a way of pre-evaluating some of the code at compile-time rather than run-time. Using template specialization, C++ Templates are considered Turing complete.

Technical overview

There are two kinds of templates: function templates and class templates. A *function template* is a pattern for creating ordinary functions based upon the parameterizing types supplied when instantiated. For example, the C++ Standard Template Library contains the function template `max(x, y)` that creates functions that return either *x* or *y*, whichever is larger. `max()` could be defined like this:

```
template <typename T>
T max(T x, T y)
{
    return x < y ? y : x;
}
```

Specializations of this function template, instantiations with specific types, can be called just like an ordinary function:

```
cout << max(3, 7); // outputs 7
```

The compiler examines the arguments used to call `max` and determines that this is a call to `max(int, int)`. It then instantiates a version of the function where the parameterizing type *T* is `int`, making the equivalent of the following function:

```
int max(int x, int y)
{
    return x < y ? y : x;
}
```

This works whether the arguments x and y are integers, strings, or any other type for which the expression $x < y$ is sensible, or more specifically, for any type for which `operator<` is defined. Common inheritance is not needed for the set of types that can be used, and so it is very similar to duck typing. A program defining a custom data type can use operator overloading to define the meaning of `<` for that type, thus allowing its use with the `max()` function template. While this may seem a minor benefit in this isolated example, in the context of a comprehensive library like the STL it allows the programmer to get extensive functionality for a new data type, just by defining a few operators for it. Merely defining `<` allows a type to be used with the standard `sort()`, `stable_sort()`, and `binary_search()` algorithms or to be put inside data structures such as sets, heaps, and associative arrays.

C++ templates are completely type safe at compile time. As a demonstration, the standard type `complex` does not define the `<` operator, because there is no strict order on complex numbers. Therefore, `max(x, y)` will fail with a compile error, if x and y are complex values. Likewise, other templates that rely on `<` cannot be applied to complex data unless a comparison (in the form of a functor or function) is provided. E.g.: A `complex` cannot be used as key for a map unless a comparison is provided. Unfortunately, compilers historically generate somewhat esoteric, long, and unhelpful error messages for this sort of error. Ensuring that a certain object adheres to a method protocol can alleviate this issue. Languages which use `compare` instead of `<` can also use complex values as keys.

The second kind of template, a *class template*, extends the same concept to classes. A class template specialization is a class. Class templates are often used to make generic containers. For example, the STL has a linked list container. To make a linked list of integers, one writes `list<int>`. A list of strings is denoted `list<string>`. A list has a set of standard functions associated with it, that work for any compatible parameterizing types.

Template specialization

A powerful feature of C++'s templates is *template specialization*. This allows alternative implementations to be provided based on certain characteristics of the parameterized type that is being instantiated. Template specialization has two purposes: to allow certain forms of optimization, and to reduce code bloat.

For example, consider a `sort()` template function. One of the primary activities that such a function does is to swap or exchange the values in two of the container's positions. If the values are large (in terms of the number of bytes it takes to store each of them), then it is often quicker to first build a separate list of pointers to the objects, sort those pointers, and then build the final sorted sequence. If the values are quite small however it is usually fastest to just swap the values in-place as needed. Furthermore, if the parameterized type is already of some pointer-type, then there is no need to build a separate pointer array. Template specialization allows the template creator to write different implementations and to specify the characteristics that the parameterized type(s) must have for each implementation to be used.

Unlike function templates, class templates can be partially specialized. That means that an alternate version of the class template code can be provided when some of the template parameters are known, while leaving other template parameters generic. This can be used, for example, to create a default implementation (the *primary specialization*) that assumes that copying a parameterizing type is expensive and then create partial specializations for types that are cheap to copy, thus increasing overall efficiency. Clients of such a class template just use specializations of it without needing to know whether the compiler used the primary specialization or some partial specialization in each case. Class templates can also be *fully specialized*, which means that an alternate implementation can be provided when all of the parameterizing types are known.

Advantages and disadvantages

Some uses of templates, such as the `max()` function, were previously filled by function-like preprocessor macros (a legacy of the C programming language). For example, here is a possible `max()` macro:

```
#define max(a,b) ((a) < (b) ? (b) : (a))
```

Macros are expanded by preprocessor, before compilation proper; templates are expanded at compile time. Macros are always expanded inline; templates can also be expanded as inline functions when the compiler deems it appropriate. Thus both function-like macros and function templates have no run-time overhead.

However, templates are generally considered an improvement over macros for these purposes. Templates are type-safe. Templates avoid some of the common errors found in code that makes heavy use of function-like macros, such as evaluating parameters with side effects twice. Perhaps most importantly, templates were designed to be applicable to much larger problems than macros.

There are three primary drawbacks to the use of templates: compiler support, poor error messages, and code bloat. Many compilers historically have poor support for templates, thus the use of templates can make code somewhat less portable. Support may also be poor when a C++ compiler is being used with a linker that is not C++-aware, or when attempting to use templates across shared library boundaries. Most modern compilers however now have fairly robust and standard template support, and the new C++ standard, C++11, further addresses these issues.

Almost all compilers produce confusing, long, or sometimes unhelpful error messages when errors are detected in code that uses templates.^[18] This can make templates difficult to develop.

Finally, the use of templates requires the compiler to generate a separate *instance* of the templated class or function for every permutation of type parameters used with it. (This is necessary because types in C++ are not all the same size, and the sizes of data fields are important to how classes work.) So the indiscriminate use of templates can lead to code bloat, resulting in excessively large executables. However, judicious use of template specialization and derivation can dramatically reduce such code bloat in some cases:

So, can derivation be used to reduce the problem of code replicated because templates are used? This would involve deriving a template from an ordinary class. This technique proved successful in curbing code bloat in real use. People who do not use a technique like this have found that replicated code can cost megabytes of code space even in moderate size programs.

— Bjarne Stroustrup, *The Design and Evolution of C++*, 1994^[19]

In simple cases templates can be transformed into generics (not causing code bloat) by creating a class getting a parameter derived from a type in compile time and wrapping a template around this class. It is a nice approach for creating generic heap-based containers.

The extra instantiations generated by templates can also cause debuggers to have difficulty working gracefully with templates. For example, setting a debug breakpoint within a template from a source file may either miss setting the breakpoint in the actual instantiation desired or may set a breakpoint in every place the template is instantiated.

Also, because the compiler needs to perform macro-like expansions of templates and generate different instances of them at compile time, the implementation source code for the templated class or function must be available (e.g. included in a header) to the code using it. Templated classes or functions, including much of the Standard Template Library (STL), if

not included in header files, cannot be compiled. (This is in contrast to non-templated code, which may be compiled to binary, providing only a declarations header file for code using it.) This may be a disadvantage by exposing the implementing code, which removes some abstractions, and could restrict its use in closed-source projects.

Templates in D

The [D programming language](#) supports templates based in design on C++. Most C++ template idioms will carry over to D without alteration, but D adds some additional functionality:

- Template parameters in D are not restricted to just types and primitive values, but also allow arbitrary compile-time values (such as strings and struct literals), and aliases to arbitrary identifiers, including other templates or template instantiations.
- Template constraints and the `static if` statement provide an alternative to C++'s [substitution failure is not an error](#) (SFINAE) mechanism, similar to [C++ concepts](#).
- The `is(...)` expression allows speculative instantiation to verify an object's traits at compile time.
- The `auto` keyword and the `typeof` expression allow [type inference](#) for variable declarations and function return values, which in turn allows "Voldemort types" (types which do not have a global name).^[20]

Templates in D use a different syntax than in C++: whereas in C++ template parameters are wrapped in angular brackets (`Template<param1, param2>`), D uses an exclamation sign and parentheses: `Template!(param1, param2)`. This avoids the C++ [parsing difficulties](#) due to ambiguity with comparison operators. If there is only one parameter, the parentheses can be omitted.

Conventionally, D combines the above features to provide [compile-time polymorphism](#) using trait-based generic programming. For example, an input [range](#) is defined as any type that satisfies the checks performed by `isInputRange`, which is defined as follows:

```
template isInputRange(R)
{
    enum bool isInputRange = is(typeof(
        (inout int = 0)
        {
            R r = R.init;    // can define a range object
            if (r.empty) {}  // can test for empty
            r.popFront();    // can invoke popFront()
            auto h = r.front; // can get the front of the range
        }));
}
```

A function that accepts only input ranges can then use the above template in a template constraint:

```
auto fun(Range)(Range range)
    if (isInputRange!Range)
{
    // ...
}
```

Code generation

In addition to template metaprogramming, D also provides several features to enable compile-time code generation:

- The `import` expression allows reading a file from disk and using its contents as a string expression.
- Compile-time reflection allows enumerating and inspecting declarations and their members during compilation.
- User-defined [attributes](#) allow users to attach arbitrary identifiers to declarations, which can then be enumerated using compile-time reflection.

- Compile-Time Function Execution (CTFE) allows a subset of D (restricted to safe operations) to be interpreted during compilation.
- String mixins allow evaluating and compiling the contents of a string expression as D code that becomes part of the program.

Combining the above allows generating code based on existing declarations. For example, D serialization frameworks can enumerate a type's members and generate specialized functions for each serialized type to perform serialization and deserialization. User-defined attributes could further indicate serialization rules.

The `import` expression and compile-time function execution also allow efficiently implementing domain-specific languages. For example, given a function that takes a string containing an HTML template and returns equivalent D source code, it is possible to use it in the following way:

```
// Import the contents of example.htt as a string manifest constant.
enum htmlTemplate = import("example.htt");

// Transpile the HTML template to D code.
enum htmlDCode = htmlTemplateToD(htmlTemplate);

// Paste the contents of htmlDCode as D code.
mixin(htmlDCode);
```

Genericity in Eiffel

Generic classes have been a part of Eiffel since the original method and language design. The foundation publications of Eiffel,^{[21][22]} use the term *genericity* to describe the creation and use of generic classes.

Basic/Unconstrained genericity

Generic classes are declared with their class name and a list of one or more *formal generic parameters*. In the following code, class `LIST` has one formal generic parameter `G`

```
class
  LIST [G]
  ...
feature -- Access
  item: G
  -- The item currently pointed to by cursor
  ...
feature -- Element change
  put (new_item: G)
  -- Add 'new_item' at the end of the List
  ...
```

The formal generic parameters are placeholders for arbitrary class names that will be supplied when a declaration of the generic class is made, as shown in the two *generic derivations* below, where `ACCOUNT` and `DEPOSIT` are other class names. `ACCOUNT` and `DEPOSIT` are considered *actual generic parameters* as they provide real class names to substitute for `G` in actual use.

```
list_of_accounts: LIST [ACCOUNT]
  -- Account List

list_of_deposits: LIST [DEPOSIT]
  -- Deposit List
```

Within the Eiffel type system, although class `LIST [G]` is considered a class, it is not considered a type. However, a generic derivation of `LIST [G]` such as `LIST [ACCOUNT]` is considered a type.

Constrained genericity

For the list class shown above, an actual generic parameter substituting for *G* can be any other available class. To constrain the set of classes from which valid actual generic parameters can be chosen, a *generic constraint* can be specified. In the declaration of class `SORTED_LIST` below, the generic constraint dictates that any valid actual generic parameter will be a class that inherits from class `COMPARABLE`. The generic constraint ensures that elements of a `SORTED_LIST` can in fact be sorted.

```
class
    SORTED_LIST [G -> COMPARABLE]
```

Generics in Java

Support for the *generics*, or "containers-of-type-T" was added to the [Java programming language](#) in 2004 as part of J2SE 5.0. In Java, generics are only checked at compile time for type correctness. The generic type information is then removed via a process called type erasure, to maintain compatibility with old JVM implementations, making it unavailable at runtime. For example, a `List<String>` is converted to the raw type `List`. The compiler inserts type casts to convert the elements to the `String` type when they are retrieved from the list, reducing performance compared to other implementations such as C++ templates.

Genericity in .NET [C#, VB.NET]

Generics were added as part of [.NET Framework 2.0](#) in November 2005, based on a research prototype from Microsoft Research started in 1999.^[23] Although similar to generics in Java, .NET generics do not apply type erasure, but implement generics as a first class mechanism in the runtime using reification. This design choice provides additional functionality, such as allowing reflection with preservation of generic types, as well as alleviating some of the limitations of erasure (such as being unable to create generic arrays).^{[24][25]} This also means that there is no performance hit from runtime casts and normally expensive boxing conversions. When primitive and value types are used as generic arguments, they get specialized implementations, allowing for efficient generic collections and methods. As in C++ and Java, nested generic types such as `Dictionary<string, List<int>>` are valid types, however are advised against for member signatures in code analysis design rules.^[26]

.NET allows six varieties of generic type constraints using the `where` keyword including restricting generic types to be value types, to be classes, to have constructors, and to implement interfaces.^[27] Below is an example with an interface constraint:

```
1 using System;
2
3 class Sample
4 {
5     static void Main()
6     {
7         int[] array = { 0, 1, 2, 3 };
8         MakeAtLeast<int>(array, 2); // Change array to { 2, 2, 2, 3 }
9         foreach (int i in array)
10             Console.WriteLine(i); // Print results.
11         Console.ReadKey(true);
12     }
13
14     static void MakeAtLeast<T>(T[] list, T lowest) where T : IComparable<T>
15     {
16         for (int i = 0; i < list.Length; i++)
17             if (list[i].CompareTo(lowest) < 0)
18                 list[i] = lowest;
19     }
20 }
```

The `MakeAtLeast()` method allows operation on arrays, with elements of generic type `T`. The method's type constraint indicates that the method is applicable to any type `T` that implements the generic `Comparable<T>` interface. This ensures a compile time error, if the method is called if the type does not support comparison. The interface provides the generic method `CompareTo(T)`.

The above method could also be written without generic types, simply using the non-generic `Array` type. However, since arrays are contravariant, the casting would not be type safe, and compiler may miss errors that would otherwise be caught while making use of the generic types. In addition, the method would need to access the array items as objects instead, and would require casting to compare two elements. (For value types like types such as `int` this requires a boxing conversion, although this can be worked around using the `Comparer<T>` class, as is done in the standard collection classes.)

A notable behavior of static members in a generic .NET class is static member instantiation per run-time type (see example below).

```
//A generic class
public class GenTest<T>
{
    //A static variable - will be created for each type on refraction
    static CountedInstances OnePerType = new CountedInstances();

    //a data member
    private T mT;

    //simple constructor
    public GenTest(T pT)
    {
        mT = pT;
    }
}

//a class
public class CountedInstances
{
    //Static variable - this will be incremented once per instance
    public static int Counter;

    //simple constructor
    public CountedInstances()
    {
        //increase counter by one during object instantiation
        CountedInstances.Counter++;
    }
}

//main code entry point
//at the end of execution, CountedInstances.Counter = 2
GenTest<int> g1 = new GenTest<int>(1);
GenTest<int> g11 = new GenTest<int>(11);
GenTest<int> g111 = new GenTest<int>(111);
GenTest<double> g2 = new GenTest<double>(1.0);
```

Genericity in Delphi

Delphi's Object Pascal dialect acquired generics in the Delphi 2007 release, initially only with the (now discontinued) .NET compiler before being added to the native code in the Delphi 2009 release. The semantics and capabilities of Delphi generics are largely modelled on those had by generics in .NET 2.0, though the implementation is by necessity quite different. Here's a more or less direct translation of the first C# example shown above:

```
program Sample;
```

```

{$APPTYPE CONSOLE}

uses
  Generics.Defaults; //for IComparer<>

type
  TUtils = class
    class procedure MakeAtLeast<T>(Arr: TArray<T>; const Lowest: T;
      Comparer: IComparer<T>); overload;
    class procedure MakeAtLeast<T>(Arr: TArray<T>; const Lowest: T); overload;
  end;

class procedure TUtils.MakeAtLeast<T>(Arr: TArray<T>; const Lowest: T;
  Comparer: IComparer<T>);
var
  I: Integer;
begin
  if Comparer = nil then Comparer := TComparer<T>.Default;
  for I := Low(Arr) to High(Arr) do
    if Comparer.Compare(Arr[I], Lowest) < 0 then
      Arr[I] := Lowest;
  end;

class procedure TUtils.MakeAtLeast<T>(Arr: TArray<T>; const Lowest: T);
begin
  MakeAtLeast<T>(Arr, Lowest, nil);
end;

var
  Ints: TArray<Integer>;
  Value: Integer;
begin
  Ints := TArray<Integer>.Create(0, 1, 2, 3);
  TUtils.MakeAtLeast<Integer>(Ints, 2);
  for Value in Ints do
    WriteLn(Value);
  ReadLn;
end.

```

As with C#, methods as well as whole types can have one or more type parameters. In the example, `TArray` is a generic type (defined by the language) and `MakeAtLeast` a generic method. The available constraints are very similar to the available constraints in C#: any value type, any class, a specific class or interface, and a class with a parameterless constructor. Multiple constraints act as an additive union.

Genericity in Free Pascal

Free Pascal implemented generics before Delphi, and with different syntax and semantics. However, work is now underway to implement Delphi generics alongside native FPC ones (see [Wiki \(http://wiki.lazarus.freepascal.org/User_Changes_2.6.0\)](http://wiki.lazarus.freepascal.org/User_Changes_2.6.0)). This allows Free Pascal programmers to use generics in whatever style they prefer.

Delphi and Free Pascal example:

```

// Delphi style
unit A;

{$ifdef fpc}
  {$mode delphi}
{$endif}

interface

type
  TGenericClass<T> = class
    function Foo(const AValue: T): T;
  end;

implementation

```

```

function TGenericClass<T>.Foo(const AValue: T): T;
begin
    Result := AValue + AValue;
end;

end.

// Free Pascal's ObjFPC style
unit B;

{$ifdef fpc}
    {$mode objfpc}
{$endif}

interface

type
    generic TGenericClass<T> = class
        function Foo(const AValue: T): T;
    end;

implementation

function TGenericClass.Foo(const AValue: T): T;
begin
    Result := AValue + AValue;
end;

end.

// example usage, Delphi style
program TestGenDelphi;

{$ifdef fpc}
    {$mode delphi}
{$endif}

uses
    A, B;

var
    GC1: A.TGenericClass<Integer>;
    GC2: B.TGenericClass<String>;
begin
    GC1 := A.TGenericClass<Integer>.Create;
    GC2 := B.TGenericClass<String>.Create;
    WriteLn(GC1.Foo(100)); // 200
    WriteLn(GC2.Foo('hello')); // hellohello
    GC1.Free;
    GC2.Free;
end.

// example usage, ObjFPC style
program TestGenDelphi;

{$ifdef fpc}
    {$mode objfpc}
{$endif}

uses
    A, B;

// required in ObjFPC
type
    TGenericClassInt = specialize A.TGenericClass<Integer>;
    TGenericClassString = specialize B.TGenericClass<String>;
var
    GC1: TGenericClassInt;
    GC2: TGenericClassString;
begin
    GC1 := TGenericClassInt.Create;
    GC2 := TGenericClassString.Create;
    WriteLn(GC1.Foo(100)); // 200
    WriteLn(GC2.Foo('hello')); // hellohello

```

```
GC1.Free;
GC2.Free;
end.
```

Functional languages

Genericity in Haskell

The type class mechanism of Haskell supports generic programming. Six of the predefined type classes in Haskell (including `Eq`, the types that can be compared for equality, and `Show`, the types whose values can be rendered as strings) have the special property of supporting *derived instances*. This means that a programmer defining a new type can state that this type is to be an instance of one of these special type classes, without providing implementations of the class methods as is usually necessary when declaring class instances. All the necessary methods will be "derived" – that is, constructed automatically – based on the structure of the type. For instance, the following declaration of a type of binary trees states that it is to be an instance of the classes `Eq` and `Show`:

```
data BinTree a = Leaf a | Node (BinTree a) a (BinTree a)
    deriving (Eq, Show)
```

This results in an equality function (`==`) and a string representation function (`show`) being automatically defined for any type of the form `BinTree T` provided that `T` itself supports those operations.

The support for derived instances of `Eq` and `Show` makes their methods `==` and `show` generic in a qualitatively different way from para-metrically polymorphic functions: these "functions" (more accurately, type-indexed families of functions) can be applied to values of various types, and although they behave differently for every argument type, little work is needed to add support for a new type. Ralf Hinze (2004) has shown that a similar effect can be achieved for user-defined type classes by certain programming techniques. Other researchers have proposed approaches to this and other kinds of genericity in the context of Haskell and extensions to Haskell (discussed below).

PolyP

PolyP was the first generic programming language extension to Haskell. In PolyP, generic functions are called *polytypic*. The language introduces a special construct in which such polytypic functions can be defined via structural induction over the structure of the pattern functor of a regular datatype. Regular datatypes in PolyP are a subset of Haskell datatypes. A regular datatype `t` must be of kind `* → *`, and if `a` is the formal type argument in the definition, then all recursive calls to `t` must have the form `t a`. These restrictions rule out higher-kinded datatypes as well as nested datatypes, where the recursive calls are of a different form. The `flatten` function in PolyP is here provided as an example:

```
flatten :: Regular d => d a -> [a]
flatten = cata fl

polytypic fl :: f a [a] -> [a]
case f of
  g+h -> either fl fl
  g*h -> \(x,y) -> fl x ++ fl y
  () -> \x -> []
  Par -> \x -> [x]
  Rec -> \x -> x
  d@g -> concat . flatten . pmap fl
  Con t -> \x -> []

cata :: Regular d => (FunctorOf d a b -> b) -> d a -> b
```

Generic Haskell

Generic Haskell is another extension to [Haskell](#), developed at [Utrecht University](#) in the [Netherlands](#). The extensions it provides are:

- *Type-indexed values* are defined as a value indexed over the various Haskell type constructors (unit, primitive types, sums, products, and user-defined type constructors). In addition, we can also specify the behaviour of a type-indexed values for a specific constructor using *constructor cases*, and reuse one generic definition in another using *default cases*.

The resulting type-indexed value can be specialized to any type.

- *Kind-indexed types* are types indexed over kinds, defined by giving a case for both $*$ and $k \rightarrow k'$. Instances are obtained by applying the kind-indexed type to a kind.
- Generic definitions can be used by applying them to a type or kind. This is called *generic application*. The result is a type or value, depending on which sort of generic definition is applied.
- *Generic abstraction* enables generic definitions be defined by abstracting a type parameter (of a given kind).
- *Type-indexed types* are types that are indexed over the type constructors. These can be used to give types to more involved generic values. The resulting type-indexed types can be specialized to any type.

As an example, the equality function in Generic Haskell:^[28]

```
type Eq {[ * ]} t1 t2 = t1 -> t2 -> Bool
type Eq {[ k -> 1 ]} t1 t2 = forall u1 u2. Eq {[ k ]} u1 u2 -> Eq {[ 1 ]} (t1 u1) (t2 u2)

eq { | t :: k | } :: Eq {[ k ]} t t
eq { | Unit | } _ _ = True
eq { | :+: | } eqA eqB (Inl a1) (Inl a2) = eqA a1 a2
eq { | :+: | } eqA eqB (Inr b1) (Inr b2) = eqB b1 b2
eq { | :+: | } eqA eqB _ _ = False
eq { | :* | } eqA eqB (a1 :* b1) (a2 :* b2) = eqA a1 a2 && eqB b1 b2
eq { | Int | } = (==)
eq { | Char | } = (==)
eq { | Bool | } = (==)
```

Clean

[Clean](#) offers generic programming based PolyP and the generic Haskell as supported by the GHC>=6.0. It parametrizes by kind as those but offers overloading.

Other languages

The [ML](#) family of programming languages support generic programming through [parametric polymorphism](#) and generic modules called *functors*. Both [Standard ML](#) and [OCaml](#) provide functors, which are similar to class templates and to Ada's generic packages. [Scheme](#) syntactic abstractions also have a connection to genericity – these are in fact a superset of templating à la C++.

A [Verilog](#) module may take one or more parameters, to which their actual values are assigned upon the instantiation of the module. One example is a generic [register](#) array where the array width is given via a parameter. Such the array, combined with a generic wire vector, can make a generic buffer or memory module with an arbitrary bit width out of a single module implementation.^[29]

[VHDL](#), being derived from Ada, also have generic ability.

See also

- [Concept \(generic programming\)](#)

- [Partial evaluation](#)
- [Template metaprogramming](#)
- [Type polymorphism](#)

References


1. Lee, Kent D. (15 December 2008). *Programming Languages: An Active Learning Approach* (<https://books.google.com/books?id=OuW5dC2O99AC&pg=PA9>). Springer Science & Business Media. pp. 9–10. ISBN 978-0-387-79422-8.
2. Milner, R.; Morris, L.; Newey, M. (1975). "A Logic for Computable Functions with Reflexive and Polymorphic Types". *Proceedings of the Conference on Proving and Improving Programs*.
3. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns*. Addison-Wesley. ISBN 0-201-63361-2.
4. Musser & Stepanov 1989.
5. Musser, David R.; Stepanov, Alexander A. *Generic Programming* (<http://stepanovpapers.com/genprog.pdf>) (PDF).
6. Alexander Stepanov; Paul McJones (June 19, 2009). *Elements of Programming*. Addison-Wesley Professional. ISBN 978-0-321-63537-2.
7. Musser, David R.; Stepanov, Alexander A. "A library of generic algorithms in Ada" (<http://dl.acm.org/citation.cfm?doid=317500.317529>). *Proceedings of the 1987 annual ACM SIGAda international conference on Ada*: 216–225.
8. Alexander Stepanov and Meng Lee: The Standard Template Library. HP Laboratories Technical Report 95-11(R.1), 14 November 1995
9. Matthew H. Austern: Generic programming and the STL: using and extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA 1998
10. Jeremy G. Siek, Lie-Quan Lee, Andrew Lumsdaine: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley 2001
11. Stepanov, Alexander. *Short History of STL* (<http://stepanovpapers.com/history%20of%20STL.pdf>) (PDF).
12. Stroustrup, Bjarne. *Evolving a language in and for the real world: C++ 1991-2006* (<http://www.stroustrup.com/hopl-alm-ost-final.pdf>) (PDF).
13. Lo Russo, Graziano. "An Interview with A. Stepanov" (<http://www.stlport.org/resources/StepanovUSA.html>).
14. Roland Backhouse; Patrik Jansson; Johan Jeuring; Lambert Meertens (1999). "*Generic Programming — an Introduction*" (<http://www.cse.chalmers.se/~patrik/poly/afp98/genprogintro.pdf>) (PDF).
15. Lämmel, Ralf; Peyton Jones, Simon. "Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming" (<https://www.microsoft.com/en-us/research/wp-content/uploads/2003/01/hmap.pdf>) (PDF). Microsoft. Retrieved 16 October 2016.
16. Gabriel Dos Reis; Jaakko Järvi (2005). "What is Generic Programming? (preprint LCSD'05)" (https://web.archive.org/web/20051225114849/http://lcsd05.cs.tamu.edu/papers/dos_reis_et_al.pdf) (PDF). Archived from the original (http://lcsd05.cs.tamu.edu/papers/dos_reis_et_al.pdf) (PDF) on 25 December 2005.
17. R. Garcia; J. Järvi; A. Lumsdaine; J. Siek; J. Willcock (2005). "An extended comparative study of language support for generic programming (preprint)" (http://www.osl.iu.edu/publications/prints/2005/garcia05:_extended_comparing05.pdf) (PDF).
18. Stroustrup, Dos Reis (2003): Concepts - Design choices for template argument checking (<http://www.stroustrup.com/N1522-concept-criteria.pdf>)
19. Stroustrup, Bjarne (1994). "15.5 Avoiding Code Replication". *The Design and Evolution of C++*. Reading, Massachusetts: Addison-Wesley. pp. 346–348. ISBN 978-81-317-1608-3.
20. Bright, Walter. "Voldemort Types in D" (<http://www.drdoobbs.com/cpp/voldemort-types-in-d/232901591>). *Dr. Dobbs*. Retrieved 3 June 2015.
21. *Object-Oriented Software Construction*, Prentice Hall, 1988, and *Object-Oriented Software Construction, second edition*, Prentice Hall, 1997.
22. *Eiffel: The Language*, Prentice Hall, 1991.

23. .NET/C# Generics History: Some Photos From Feb 1999 (<http://blogs.msdn.com/b/dsyme/archive/2011/03/15/net-c-generics-history-some-photos-from-feb-1999.aspx>)
24. C#: Yesterday, Today, and Tomorrow: An Interview with Anders Hejlsberg (<http://www.ondotnet.com/pub/a/dotnet/2005/10/17/interview-with-anders-hejlsberg.html>)
25. Generics in C#, Java, and C++ (<http://www.artima.com/intv/generics2.html>)
26. Code Analysis CA1006: Do not nest generic types in member signatures (<http://msdn.microsoft.com/en-us/library/ms182144.aspx>)
27. Constraints on Type Parameters (C# Programming Guide) (<http://msdn2.microsoft.com/en-us/library/d5x73970.aspx>)
28. The Generic Haskell User's Guide (<http://www.cs.uu.nl/research/projects/generic-haskell/compiler/diamond/GHUsersGuide.pdf>)
29. Verilog by Example, Section *The Rest for Reference*. Blaine C. Readler, Full Arc Press, 2011. ISBN 978-0-9834973-0-1

Citations

- Musser, D. R.; Stepanov, A. A. (1989). "Generic programming". In P. Gianni. *Symbolic and Algebraic Computation: International symposium ISSAC 1988*. Lecture Notes in Computer Science. **358**. pp. 13–25. doi:10.1007/3-540-51084-2_2 (https://doi.org/10.1007%2F3-540-51084-2_2). ISBN 978-3-540-51084-0.
- Stroustrup, Bjarne (2007). *Evolving a language in and for the real world: C++ 1991-2006* (<http://www.research.att.com/~bs/hopl-almost-final.pdf>) (PDF). ACM HOPL 2007.
- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.

Further reading

- Gabriel Dos Reis and Jaakko Järvi, *What is Generic Programming?* (<http://www.elegantcoding.com/2012/04/what-is-generic-programming.html>), LCSD 2005 (<http://lcsd05.cs.tamu.edu>).
- Gibbons, Jeremy (2007). Backhouse, R.; Gibbons, J.; Hinze, R.; Jeuring, J., eds. *Datatype-generic programming*. Spring School on Datatype-Generic Programming 2006. Lecture Notes in Computer Science. **4719**. Heidelberg: Springer. pp. 1–71. CiteSeerX 10.1.1.159.1228 (<https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.159.1228>) .
- Bertrand Meyer. "Genericity vs Inheritance (<http://se.ethz.ch/~meyer/publications/acm/geninh.pdf>).". In *OOPSLA (First ACM Conference on Object-Oriented Programming Systems, Languages and Applications)*, Portland (Oregon), 29 September–2 October 1986, pages 391–405.

External links

- [generic-programming.org](http://www.generic-programming.org) (<http://www.generic-programming.org>)
- Alexander A. Stepanov, *Collected Papers of Alexander A. Stepanov* (<http://www.stepanovpapers.com/>) (creator of the STL)

C++/D

- Walter Bright, *Templates Revisited* (<http://www.digitalmars.com/d/templates-revisited.html>).
- David Vandevoorde, Nicolai M Josuttis, *C++ Templates: The Complete Guide*, 2003 Addison-Wesley. ISBN 0-201-73484-2

C#/.NET

- Jason Clark, "Introducing Generics in the Microsoft CLR (<http://msdn.microsoft.com/msdnmag/issues/03/09/NET/>)", September 2003, *MSDN Magazine*, Microsoft.

- Jason Clark, "More on Generics in the Microsoft CLR (<http://msdn.microsoft.com/msdnmag/issues/03/10/NET/>)," October 2003, *MSDN Magazine*, Microsoft.
- M. Aamir Maniar, *Generics.Net* (<http://codeplex.com/Wiki/View.aspx?ProjectName=genericsnet>). An open source generics library for C#.

Delphi/Object Pascal

- Nick Hodges, "Delphi 2009 Reviewers Guide (<http://edn.embarcadero.com/article/38757>)," October 2008, *Embarcadero Developer Network*, Embarcadero.
- Craig Stuntz, "Delphi 2009 Generics and Type Constraints (<http://blogs.teamb.com/craigstuntz/2008/08/29/37832/>)," October 2008
- Dr. Bob, "Delphi 2009 Generics (<http://www.drbob42.com/examines/examinA4.htm>)"
- Free Pascal: Free Pascal Reference guide Chapter 8: Generics (<http://www.freepascal.org/docs-html/ref/refch8.html>), Michaël Van Canneyt, 2007
- Delphi for Win32: Generics with Delphi 2009 Win32 (<http://sjrd.developpez.com/delphi/tutoriel/generics/>), Sébastien DOERAENE, 2008
- Delphi for .NET: Delphi Generics (http://www.felix-colibri.com/papers/ooop_components/delphi_generics_tutorial/delphi_generics_tutorial.html), Felix COLIBRI, 2008

Eiffel

- Eiffel ISO/ECMA specification document (<http://www.ecma-international.org/publications/standards/Ecma-367.htm>)

Haskell

- Johan Jeuring, Sean Leather, José Pedro Magalhães, and Alexey Rodriguez Yakushev. *Libraries for Generic Programming in Haskell* (<http://www.cs.uu.nl/wiki/pub/GP/CourseLiterature/afp08.pdf>). Utrecht University.
- Dæv Clarke, Johan Jeuring and Andres Löb, The Generic Haskell user's guide (<http://www.cs.uu.nl/research/projects/generic-haskell/compiler/diamond/GHUsersGuide.pdf>)
- Ralf Hinze, "Generics for the Masses (<http://www.cs.ox.ac.uk/ralf.hinze/publications/Masses.pdf>)," In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2004.
- Simon Peyton Jones, editor, *The Haskell 98 Language Report* (<http://haskell.org/onlinereport/index.html>), Revised 2002.
- Ralf Lämmel and Simon Peyton Jones, "Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming," In *Proceedings of the ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, 2003. (Also see the website devoted to this research (<https://web.archive.org/web/20041207211740/http://www.cs.vu.nl/boilerplate/>))
- Andres Löb, *Exploring Generic Haskell* (<http://www.cs.uu.nl/~andres/ExploringGH.pdf>), Ph.D. thesis, 2004 Utrecht University. ISBN 90-393-3765-9
- Generic Haskell: a language for generic programming (<http://www.generic-haskell.org/>)

Java

- Gilad Bracha, *Generics in the Java Programming Language* (<http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>), 2004.
- Maurice Naftalin and Philip Wadler, *Java Generics and Collections*, 2006, O'Reilly Media, Inc. ISBN 0-596-52775-6
- Peter Sestoft, *Java Precisely, Second Edition*, 2005 MIT Press. ISBN 0-262-69325-9
- Java SE 9 (<https://docs.oracle.com/javase/9/docs/>), 2004 Sun Microsystems, Inc.
- Angelika Langer, *Java Generics FAQs* (<http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Generic_programming&oldid=831635746"

This page was last edited on 21 March 2018, at 14:52.

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia](#)

Foundation, Inc., a non-profit organization.