# Implicit and Dynamic Parameters in C++

Christian Heinlein

Dept. of Computer Science, University of Ulm, Germany
christian.heinlein@uni-ulm.de

**Abstract.** Implicit and dynamic parameters are proposed as a general means to reduce the length of argument lists of function calls without resorting to dangerous global variables. In C++, these new kinds of parameters constitute a generalization of parameters with default arguments, whose values can be omitted in function calls. In contrast to the latter, however, the values of implicit and dynamic parameters are not obtained from a function's definition context, but rather from its different calling contexts. This is in turn similar to so-called dependent names in function templates, but offers a higher degree of flexibility and comprehensibility.

## 1 Introduction

There are basically two ways to pass information from one procedure or function of a program to another: parameters and global variables. Even though the former are usually preferred for good reasons and use of the latter for this purpose is generally discouraged, there are circumstances where parameters turn out to be inconvenient and cumbersome and therefore the use of global variables becomes tempting. In particular, if functions require large numbers of parameters, most of which are simply passed down to other functions, providing this information via global variables could significantly reduce the size of many parameter and argument lists. Furthermore, if major parts of this information usually remain unchanged during a program execution, the use of global variables is even more appealing. Finally, if it becomes necessary to retroactively extend the parameter list of some deeply nested function, each call of this function must be augmented with additional arguments, which usually requires the parameter lists of all functions containing these calls to get extended, too, etc.

On the other hand, using global variables to pass information between functions is dangerous, especially in multi-threaded programs, where one thread might inadvertently change the value of variables needed by other threads. But even in single-threaded applications, it might happen that global variables required by a particular function are modified by a subordinate function called from it. Finally, if exceptions cause unexpected and premature terminations of functions, temporary modifications to global variables performed by these functions might not be undone as expected.

To dissolve this longstanding tension between using parameters and global variables, *implicit parameters* [7] and *dynamic variables* [3] have been proposed

D. Lightfoot and C. Szyperski (Eds.): JMLC 2006, LNCS 4228, pp. 37–56, 2006.

earlier as different means to provide the benefits of global variables, i. e., short and comprehensible parameter and/or argument lists, without suffering from their drawbacks. While the former are specifically tailored to functional programming languages and provide static type checking, the latter also address imperative languages, but lack static type safety. Building on these approaches, the main contribution of this paper is their combination into a single coherent framework for imperative languages, i. e., *implicit and dynamic parameters*, that provides a large degree of flexibility combined with static type safety. More specifically, language extensions for C++ are proposed which generalize its notion of parameters with *default arguments* and also provide a superior replacement for *dependent names* in templates. Nevertheless, the basic concept of implicit and dynamic parameters is actually language-independent and might be incorporated into many other languages, too.

After reviewing in Sec. 2 the basics of functions, overloading, and default arguments in C++, the concept of implicit and dynamic parameters is introduced and developed in Sec. 3. Its basic implementation ideas as a precompiler-based language extension for C++ are described in Sec. 4, before concluding the paper with a discussion of related work in Sec. 5.

## 2 Functions, Overloading, and Default Arguments in C++

Even though C++ provides a large number of different function kinds, including global functions, virtual, non-virtual, and static member functions, constructors, and function call operators [10], their basic principle is always the same: a function consists of a name, a parameter list, a (possibly `void`) result type, and a body. Therefore, examples will be limited to global functions and constructors in the sequel.

All kinds of functions can be statically *overloaded* by defining multiple functions of the same name (in the same scope) with different parameter lists, e. g.:

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

When resolving a call to such a function, the *static types* of all arguments are used to determine the *best viable function* at compile time. If no viable function is found at all or several viable functions remain where none is better than the others, the call is ill-formed. For example, `max(1, 2)` and `max(1.0, 2.0)` will call the first resp. second version of `max` defined above, while `max("a", "b")` and `max(1, 2.0)` cannot be resolved due to missing resp. ambiguous definitions.[1]

The trailing parameters in a function definition might have *default arguments*, i. e., associated expressions whose values will be used to initialize these parameters if corresponding arguments are missing in a call, e. g.:

[1] Note that in C++ an `int` value is implicitly convertible to `double` *and vice versa*, and both conversions are *equally ranked*.

```
// Print floating point number d on standard output stream
// with minimum field width w and precision p.
void print (double d, int w = 10, int p = 5) { ..... }
```

This function can be called with one, two, or three arguments, where `print(d)` and `print(d, w)` are equivalent to `print(d, 10, 5)` and `print(d, w, 5)`, respectively. Therefore, the single definition of `print` above is similar in effect to the following three definitions:

```
// Print d with field width w and precision p.
void print (double d, int w, int p) { ..... }

// Print d with field width w and precision 5.
void print (double d, int w) { print(d, w, 5); }

// Print d with field width 10 and precision 5.
void print (double d) { print(d, 10, 5); }
```

In the first case, however, a *single* function is defined which might be *called* with different numbers of arguments, while in the second case, there are actually three different overloaded functions.

A default argument is not restricted to a simple value such as `10`, but might be any expression, which is evaluated *each time* it is used in a call. Names appearing in such an expression are interpreted and bound in the context of its *definition*, not in the context of a call, e. g.:

```
int width = 10;
void print (double d, int w = max(width, 20), int p = 5);
```

If `print` is called with a single argument `d`, its second argument `w` is initialized with the value of the expression `max(width, 20)`, where `width` corresponds to the global variable defined before, even if the calling function contains a local variable of the same name that basically hides the global one. Therefore, *literally* adding a default argument expression to a function call might lead to a quite different result than omitting the argument.

Default arguments might be specified later on, after a function has been declared or defined for the first time, and it is even possible to specify different default arguments for the same function in different local scopes, e. g.:

```
// Initial definition without default arguments.
void print (double d, int w, int p) { ..... }

// Later declaration with one default argument.
void print (double d, int w, int p = 5);

// Client function.
void client (double d) {
  // Local declaration with two default arguments.
```

```
  void print (double d, int w = 10, int p = 4);

  // Call equivalent to: print(d, 10, 4).
  print(d);
}
```

When an overloaded function is called, only the *explicitly specified arguments* are considered to determine the best viable function. However, functions possessing more parameters than arguments given will be included in the set of *candidate functions* if the missing arguments could be provided by default arguments. If the selected function actually has more parameters than arguments given, the corresponding default arguments will be supplied afterwards, e. g.:

```
void f (int x, int y);
void f (double x, int y = 0);
```

Here, a call such as `f(0)` would select the second function since the explicitly specified argument `0` of type `int` is compatible with its first parameter of type `double` and the second parameter can be satisfied from its default argument, while the first function cannot be called with only one argument. On the other hand, `f(0, 0)` would select the first function since the arguments `(0, 0)` exactly match its parameters, while the second function would require a conversion of the first argument from `int` to `double`. Again, literally adding a default argument expression to a function call might lead to a different result than omitting the argument. Furthermore, if a *member function* has default arguments, these expressions might refer to private or protected members of the class, which are inaccessible to clients calling the function; in that case, literally adding a default argument expression could even lead to a compile time error.

## 3 Implicit and Dynamic Parameters

This section introduces *implicit and dynamic parameters* as language extensions to C++. Their basic idea is similar to parameters with default arguments, as corresponding arguments can be omitted in function calls, too. However, the way to obtain values for missing arguments is quite different.

Implicit and dynamic parameters as well as parameters with default arguments will be collectively called *optional parameters* in the sequel, and the original C++ rules for parameters with default arguments are generalized to all kinds of optional parameters, i. e.:

- If a particular parameter of a function is declared optional, all subsequent parameters must be optional, too.
- Parameters might be declared optional later on, and the same function might have different optional parameters in different scopes.
- During overload resolution, only the arguments which are explicitly specified in a call are used to select the best viable function. Afterwards, if the selected function has more parameters than arguments given, values for optional parameters will be added as required.

### 3.1 Implicit Parameters

Function parameters are declared *implicit* by prefixing their declaration with the C++ keyword `using`[2], e. g.:

```
void print (double d, using int width, using int prec);
```

If an argument corresponding to an implicit parameter is missing in a call, an entity with the *same name* as the parameter from the *calling context* is substituted. If no such entity is found there, or if its type is incompatible with the parameter's type, the call is rejected by the compiler. For instance, the call `print(d)` is equivalent to `print(d, width, prec)` where the names `width` and `prec` are looked up and bound in the calling context. This implies that the names of implicit parameters are mandatory in declarations and significant for callers, in contrast to ordinary parameters whose names are irrelevant for clients and might even be omitted in function declarations.

According to the general rules about optional parameters stated above, it is possible to declare a parameter implicit later on, possibly in a different scope, and it is also possible to change its name on that occasion, including the possibility to introduce a name for a formerly anonymous parameter, e. g.:

```
void print (double d, int w, int);
......
void print (double d, using int width, using int prec);
```

Usually, the "entity" that is used to satisfy an implicit parameter of a called function is some kind of variable, including local variables and parameters of the calling function, member variables of an enclosing class, and global variables. In particular, the calling function might itself possess an implicit parameter of the same name (and a compatible type). If the implicit parameter has a function pointer or reference type, however, the entity might also be a function of an appropriate type, e. g.:

```
// Sort vector v using function less to compare its elements.
void sort (vector<string>& v,
  using bool less (const string&, const string&)) { ..... }

// Namespace N1 containing a definition of less
// and a client function calling sort with the latter.
namespace N1 {
  bool less (const string& s1, const string& s2) { ..... }
  void client (vector<string>& v) {
    sort(v); // calls: sort(v, N1::less)
  }
```

[2] At first sight, a different keyword such as `implicit` might be more appropriate. However, to avoid incompatibilities with C++ code using this as an identifier, the existing keyword `using` has been "re-used."

```
}

// Namespace N2 containing a different definition of less
// and a client function calling sort with the latter.
namespace N2 {
  bool less (const string& s1, const string& s2) { ..... }
  void client (vector<string>& v) {
    sort(v); // calls: sort(v, N2::less)
  }
}
```

### 3.2 Constructors with Implicit Parameters

If a constructor has implicit parameters, their values are also supplied from the calling context if necessary, no matter whether the constructor is called directly in so-called *functional notation* or indirectly in variable initializations (or *member initializers* of other constructors), e. g.:

```
// Hash table for strings.
class HashTable {
  ......
public:
  // Create hash table with given size and hash function.
  HashTable (using int size, using int hash (const string&));
};

// Preferred hash table size and hash function.
int size = 193;
int hash (const string& s) { ..... }

// Direct constructor call.
HashTable t1 = HashTable(101); // calls: HashTable(101, hash)

// Indirect constructor call.
HashTable t2;                  // calls: HashTable(size, hash)
```

In this example, the direct constructor call `HashTable(101)` used in the declaration of `t1` is equivalent to `HashTable(101, hash)`, using the definition of `hash` given before, while the declaration of `t2` contains an indirect call to the *default constructor* `HashTable()` (because the variable is not explicitly initialized), which is equivalent to `HashTable(size, hash)`.

In other contexts, however, where constructor calls are completely invisible, these calls must not depend on implicit parameters in order to avoid too much implicitness and consequent incomprehensibility. This includes implicit *copy* and *conversion constructor* calls as well as implicit default constructor calls in *ctor-initializers* [10]. For example, even though the `HashTable` constructor defined above may be called with a single argument of type `int` if the second parameter

`hash` can be satisfied from the calling context (as shown in the declaration of `t1`), this constructor cannot be used as a conversion constructor to implicitly convert an `int` value to a `HashTable` object.

### 3.3 Dynamic Parameters and Environment Variables

EEven though implicit parameters need not be specified explicitly in function calls, the compiler checks that corresponding entities are found in the calling context and rejects calls otherwise. While this avoids run time errors or undefined behaviour due to missing parameter values, it leads to a rather tight coupling between callers and callees. In particular, it is not generally possible to transparently add another parameter to an existing function, even if it is declared implicit, because an entity of the same name must be present in every calling context.

To relax this strict rule and to support more loose couplings between callers and callees, *dynamic parameters* are introduced. Syntactically, dynamic parameters look like implicit parameters with default arguments, i. e., their declaration is preceded by the keyword `using` and followed by an equals sign and an accompanying expression, e. g.:

```
void print (double d,
  using int width = 10, using int prec = 5);
```

As with implicit parameters, the value for a dynamic parameter is retrieved from the calling context if the corresponding argument is missing in a call. In contrast to implicit parameters, however, the value need not be provided by the *direct* caller, but might also come from an *indirect* caller, i. e., from the complete *dynamic scope* of the call.

However, since the compiler normally does not know the set of all callers of a function (and recursively their callers etc.), it can no longer check statically whether a value required for a dynamic parameter is actually provided. Therefore, dynamic parameters always possess a default argument which will be used in cases where no value can be found in the dynamic scope of the call. As with normal default arguments, the corresponding expression is evaluated each time it is used, i. e., whenever a call is made that neither provides an explicit argument nor an implicit value for the parameter, and names appearing in the expression are interpreted and bound in the context of its definition.

As another significant difference to implicit parameters, entities intended to provide values for dynamic parameters must be explicitly marked as such to avoid accidental matches with local variables defined in indirect callers, which might not even know about a particular dynamic parameter. The underlying model employed for that purpose is quite similar to the concept of *environment variables* found in operating systems: At any point in time during the execution of a program (or a single thread within a multi-threaded program) there is a *dynamic scope* or *environment* containing variables which have been declared by so-called *export declarations* described below.

To give an example, if `print(d)` is called (with the declaration of `print` given above), this call is always accepted by the compiler and will actually call `print(d, width, prec)` if variables `width` and `prec` are found in the current environment, or `print(d, 10, 5)` if no such variables are found (or a combination of these if only one of the variables is found).

Because the compiler cannot and does not check whether the value for a dynamic parameter will be actually present at run time, it is indeed possible to transparently add additional parameters to existing functions without needing to check or change their direct callers.[3] Of course, to be actually useful, values for such parameters should be exported to the environment by some indirect caller.

### 3.4 Export Declarations

An *export declaration* is a definition of a global or local variable prefixed by the keyword `export`[4], e. g., `export int width = 10;`
The *environment variable* declared that way is initialized just like a regular variable by evaluating the optional initializer expression and/or executing an appropriate constructor. Furthermore, the variable is destroyed in the same way as a regular variable by executing its destructor when it gets out of scope, i. e., when the statement block containing the export declaration terminates (either normally or abruptly by executing a jump statement or throwing an exception) or (for a globally declared environment variable) when the entire program terminates.

In contrast to a regular variable, however, the variable is not inserted into any *static* scope at *compile time* (i. e., it will not be found by normal static name lookup), but rather added to the current *dynamic* scope (i. e., the environment) when the declaration is executed at *run time*. It is automatically removed from there immediately before it is destroyed, i. e., at the end of the enclosing statement block (if it is declared locally) or at the end of the program (if it is declared globally). If a variable of the same name and type as a newly exported variable is already present in the environment, the former is hidden by the latter until the latter is removed again, i. e., the environment is organized in a stack-like manner, and a dynamic parameter always receives the value of the top-most matching variable, if any. For example:

```
void print (double d,
  using int width = 10, using int prec = 5);

void client1 (double d) { print(d); }

void client2 (double d) {
```

[3] It will depend on the implementation strategy whether or not it is necessary to recompile the callers.

[4] Again, an existing C++ keyword is re-used, which is used for a similar purpose in Unix shells.

```
  client1(d);     // client1 calls print(d, 10, 5)
  { export int width = 20;
    client1(d);   // client1 calls print(d, 20, 5)
    { export int width = 30, prec = 10;
      client1(d); // client1 calls print(d, 30, 10)
    }
    client1(d);   // client1 calls print(d, 20, 5)
  }
  client1(d);     // client1 calls print(d, 10, 5)
}
```

### 3.5 Environment Variables with Constant and Reference Types

The type of an environment variable might be any suitable C++ type including `const`-qualified and/or reference types. Accordingly, dynamic parameters might possess such types, and an environment variable is said to *match* a dynamic parameter if the following conditions hold:

- the names of the variable and the parameter are equal;
- the *core types*, i. e., the types without any top-level `const` or `&` qualifier, of the variable and the parameter are identical;
- if the common core type is `T` and the parameter's type is `T&`, the variable's type is `T` or `T&`, but not `const T` or `const T&`.

The last rule is in accordance with normal C++ rules, which do not allow to bind a constant object to a non-constant reference via which it could be modified inadmissibly. On the other hand, the rule about identical core types is much stricter than normal C++ type compatibility rules, as it completely excludes any implicit type conversions such as standard conversions between numeric types or conversions from derived classes to base classes. The main reason for not allowing such conversions is to avoid confusion and unpleasant surprises due to unexpected or unintended conversions, which already happen occasionally with normal parameters. Combined with the loose coupling between dynamic parameters and environment variables, the danger of an accidental match would become even greater. Furthermore, when exporting a variable to the environment, one should have a clear conception about the dynamic parameters this variable is intended to match, and thus it should be easily possible to choose the appropriate type exactly, without relying on any implicit conversions. (And if really necessary, one might export several variables of the same name with different types.) Finally, an efficient implementation of more flexible matching rules would be extremely difficult, since it would actually require to perform extensive analyses at run time which are normally carried out at compile time. (For the same reason, the C++ rules for finding a matching handler for a thrown exception do not allow the full range of implicit type conversions either. However, to support

the typical idiom of catching exceptions of multiple classes with a single handler for a common base class, conversions from derived to base classes are considered in this context.)

Declaring an environment variable of some core type `T` with or without `const` or `&` qualifiers has important consequences for its usage and actually sets up different "access rights" for it:[5]

- If its type is simply `T`, it will match dynamic parameters with all kinds of qualification. In particular, it will be possible to change the variable's value indirectly via dynamic reference parameters of type `T&`, even though direct manipulations of the variable are impossible since it is not part of any static scope!
- To forbid such indirect modifications of an environment variable, a `const`-qualified type, i. e., `const T` or `const T&`, can be chosen, because a variable of such a type does not match a parameter of type `T&`. In the former case (`const T`), the variable becomes completely immutable.
- When using a reference type, i. e., `T&` or `const T&`, the environment variable can be initialized with a regular variable of type `T` in both cases and then actually constitutes an *alias* for the latter. Therefore, modifications to the regular variable are immediately reflected in the environment variable, i. e., the latter can be directly manipulated through the former, no matter whether its type is `T&` or `const T&`. However, the distinction between these types decides whether indirect manipulations via dynamic parameters are possible or not: A variable of type `T&` matches a parameter of the same type which allows such manipulations, while a variable of type `const T&` does not match such a parameter and therefore cannot be modified indirectly.

### 3.6 Correspondence to Checked and Unchecked Exceptions

The conceptual distinction between implicit and dynamic parameters exhibits interesting parallels with *checked* and *unchecked exceptions* in Java [2].

If the signature of a Java method declares a checked exception, the compiler checks that each caller of this method either catches the exception or declares it in its own signature. Likewise, if a function declares an implicit parameter, the compiler checks that each caller of this function either passes an explicit argument for it or provides an entity to satisfy it, including an implicit parameter of its own. Therefore, both scenarios are statically safe: a checked exception is guaranteed to be caught somewhere, while an implicit parameter is guaranteed to receive a value.

On the other hand, since unchecked exceptions need not be declared, the compiler cannot and does not check that they will be caught somewhere. Likewise, the compiler cannot and does not check that a dynamic parameter always receives a value from the current environment. However, the default value provided

[5] For experienced C++ programmers, the following considerations are quite obvious, since they are completely in line with standard C++ rules about constant and reference types. For less experienced or novice C++ programmers, however, they might require some time of accommodation.

for such cases guarantees well-defined behaviour anyway, while an uncaught exception leads to program termination.

In the same way as both checked and unchecked exceptions have useful applications in practice (even though some programmers tend to avoid the former to circumvent the strict checks performed by the compiler), both implicit and dynamic parameters turn out to be useful for different kinds of applications: If it is essential that a value for a particular parameter is provided (e. g., the comparison function for a sort routine) – and there is no reasonable general default value –, an implicit parameter should be used. If, on the other hand, a value is dispensable and/or a reasonable default value can be specified (e. g., the field width for a print function), a dynamic parameter is usually more appropriate as it allows greater flexibility.

### 3.7 Replacing Dependent Names in C++ Templates

If a normal C++ function such as `max` calls another function such as `less`, the latter must have been declared earlier, e. g.:

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

For functions called from function *templates*, however, this simple declare-before-use rule is replaced with rather complicated rules about *dependent names*, the *point of instantiation* of a template, etc. [10]. For example, the following generic definition of `max` is accepted by the compiler without any preceding definition of `less`, since the latter is a *dependent name* because its arguments `x` and `y` depend on the template parameter `T`:

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

If, however, `max` is actually called with arguments `x` and `y` of a particular type `T0`, a definition of `less` accepting these arguments must be found, either in the *definition context* of `max` or in the current *instantiation context*, i. e., in the calling context of the function. In this regard, dependent names are similar to implicit parameters, which are also interpreted in the calling context of a function. Therefore, the fact that `max` requires a matching definition of `less`, which is *hidden* in its *body* above, can be specified *explicitly* in its *signature* by means of an implicit parameter:

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

Similar to the original definition of `max` shown before, this function is also accepted by the compiler without any preceding definition of `less`, and a matching definition is required only when `max` is called. This time, however, no special

rules about dependent names are required, since the usage of `less` in the function's body obviously refers to the parameter declared in its signature, whose context-dependent binding is achieved by the much simpler rules for implicit parameters. Furthermore, using an implicit parameter is actually more flexible than a dependent name, since it might naturally be bound to different functions `less` in different calling contexts (cf. Sec. 3.1), whereas a dependent name is required to refer to the *same* function in *all* instantiation contexts, even across multiple translation units.[6] Finally, it is even possible to pass a function with a different name as an explicit argument.

In summary, implicit parameters constitute a superior replacement for dependent names in function templates, since they reveal hidden dependencies by moving their names from a function's body to its signature and provide more flexible means for their context-dependent binding. On the other hand, calling a function such as `less` that is passed as an argument to another function such as `max` might be less efficient than a direct, inlined call to `less` in `max`. However, if the code of both `max` and `less` is visible, a call such as `max(x, y, less)` might be completely inlined by an optimizing compiler, too, yielding, e. g., `x < y ? y : x` if `less` is defined as shown above.

## 4 Prototypical Implementation

Implicit and dynamic parameters have been implemented prototypically in a precompiler for C++ that is based on the EDG C++ Front End (cf. `www.edg.com`). In contrast to a real implementation in a compiler, a precompiler-based approach has the advantage that it is independent of any particular compiler and requires much less implementation time and effort. Both of these aspects improve the possibility to experiment with the new language constructs early and quickly and thus gain important practical experience, which might help to improve the concepts before hard-wiring them in real implementations. Of course, a precompiler-based approach does usually not achieve the same performance as a direct implementation in a compiler, but for typical experimental applications this does not really constitute a problem.

### 4.1 Dynamic Parameters and Environment Variables

Conceptually, the environment or dynamic scope of a program (or a single thread within a multi-threaded program) can be represented by a *stack* whose entries each contain a pointer or reference to a variable plus information about the variable's name and type, where the name could be stored as a string of characters, while the type could be represented by its `typeid` [10]. To find the topmost variable that matches a given dynamic parameter, the stack must be searched in top-down order for the first entry whose name and type are equal to the parameter's name and type.

[6] And to make things worse, a compiler is not forced to diagnose violations of this rule!

Although possible in principle, this representation of the environment suffers from both unnecessary storage consumption for the name and type of each variable and unnecessary run time overhead to find the topmost matching variable. To reduce these costs, the environment can be represented by a *set* of stacks, where each stack contains only references to variables of the *same* name and type. In this case, the topmost variable matching a given dynamic parameter is directly referenced by the topmost entry of the appropriate stack, which can be found rather quickly, e. g., with binary search or hashing, if names and types are still represented as strings and `typeid`s, respectively.

Even though much better than the initial solution, this approach still suffers from avoidable run time overhead to find the appropriate stack matching a dynamic parameter. To completely eliminate these costs, the name of a variable can be encoded *statically* as a (dummy) type, which can be used as one of two *template arguments* for a template class `Dyn`, where the actual type of the variable is used as the second argument:

```
// Entry of stack identified by Name and Type.
template <typename Name, typename Type>
struct Dyn {
  Type& var; // Reference to variable.
  Dyn* prev; // Pointer to previous stack entry.

  // Pointer to topmost stack entry (initially null).
  static Dyn* top;

  // Constructor pushing variable v on the stack.
  Dyn (Type& v) : var(v), prev(top) { top = this; }

  // Destructor popping topmost stack entry.
  ~Dyn () { top = prev; }
};

// Dummy template class to encode variable names as types.
template <char head, typename Tail = void>
struct Name {};
```

Using the template class `Name`, a variable name such as `x` or `xyz` is uniquely identified (even across different namespaces and translation units) by the dummy type `Name<'x'>` and `Name<'x', Name<'y', Name<'z'> > >`, respectively. Therefore, the stack containing all references to variables of type `int` and name `x` (more precisely, the pointer to its topmost element) is *statically* identified by `Dyn<Name<'x'>, int>::top`. Consequently, the topmost variable matching a dynamic parameter of type `int` and name `x` can be immediately accessed as `Dyn<Name<'x'>, int>::top ->var`, without any overhead for comparing strings or `typeid`s at run time.

Since variable export operations are performed explicitly by means of export declarations, while the corresponding remove operations shall happen

automatically when the enclosing statement block (or the entire program) terminates (cf. Sec. 3.4), it is advantageous to embed these operations in the constructor resp. destructor of class `Dyn`, as already shown above. Then, an export declaration such as

```
export int x = expr;
```

can be transformed to a corresponding declaration of a regular variable with some unique internal name such as `x__1234` (internal, since the original name `x` shall not appear in any static scope, and unique, since multiple environment variables of the same name might be defined), followed by a declaration and initialization of an additional dummy variable `_x__1234` of the corresponding `Dyn` class:

```
int x__1234 = expr;
Dyn<Name<'x'>, int> _x__1234(x__1234);
```

When the control flow of the program reaches these declarations, `x__1234` is initialized with `expr` and afterwards the constructor of `_x__1234` is called, receiving a reference to `x__1234` as an argument and pushing it onto its stack. The corresponding destructor performing the matching pop operation is automatically executed when the enclosing statement block (or the entire program) is terminated.

By implementing the stacks as linked lists whose elements are global or local variables of type `Dyn<...>` declared at the corresponding export points, no dynamic storage management is necessary for maintaining the stacks. Instead, they are "threaded" through the normal run time stack and possibly the global data segment of the program.

Based on this representation of the environment, a dynamic parameter declaration as in the following example:

```
void print (double d, int w, using int prec = 5);
```

is transformed to an ordinary parameter declaration with a default argument:

```
void print (double d, int w, int prec =
 Dyn<Name<'p', Name<'r', ...> >, int>::top ?
 Dyn<Name<'p', Name<'r', ...> >, int>::top->var : 5);
```

If the function `print` is called with three arguments, the default argument is simply ignored. If it is called with only two arguments, the default argument is evaluated as follows: If the pointer `top` of the stack identified by the name `prec` and the type `int` is not null, i. e., if this stack is not empty, the variable referenced by its topmost entry is used; otherwise, the original default argument of the dynamic parameter (which is simply `5` in the example) is evaluated. In particular, the original default argument is evaluated only if necessary, i. e., if no suitable value is found in the environment.

To simplify the presentation, the above description has ignored two details: First, if the type of an environment variable is `const T` or `const T&`, it must not match a dynamic parameter with a non-`const` reference type `T&`. To accomplish this, all `Dyn` classes actually have a second static data member `nctop` as well as a second link field `ncprev` that points to the topmost resp. previous stack entry referencing a non-`const` variable. Furthermore, two different constructors for pushing a `const` resp. non-`const` variable are provided which do not resp. do modify the `nctop` pointer. To find the topmost variable matching a non-`const` reference type `T&`, the `nctop` pointer is used instead of `top`, while for all other kinds of types (i. e., `T`, `const T`, and `const T&`) `top` is used as described above.

Second, if multi-threaded programs shall be supported, the static data members `top` and `nctop` must not directly point to stack entries, but rather refer to some kind of thread-local objects which contain such pointers, in order to maintain a separate environment for each thread.

### 4.2 Implicit Parameters

While the implementation of dynamic parameters is rather simple and straightforward – and the conceptual decoupling between export declarations and function calls using exported entities leads to an analogous decoupling in the implementation –, the precompiler-based implementation of implicit parameters turns out to be more difficult.

A rather obvious idea is to simply add missing arguments to function calls, e. g., to transform a call such as `print(d, w)` to `print(d, w, prec)` if `print`'s third parameter `prec` is implicit. If the identifier `prec` is not known in the calling context or has an incompatible type, this approach would naturally lead to a corresponding compiler error message in that case.

However, there are two important problems with this approach, which have already been pointed out in Sec. 2: First, the process of overload resolution might lead to different results if an argument for any kind of optional parameter is either explicitly specified or omitted. Second, if an implicit parameter is preceded by a parameter with a default argument (including a dynamic parameter), and the corresponding argument is also omitted in a call, it would have to be explicitly added, too. However, this is generally difficult for a precompiler operating on source code for two reasons: First, the meaning of names occurring in the default argument expression might be different when it is evaluated in the calling context instead of its definition context; second, some of these names might be inaccessible in the calling context if they refer to private or protected data members of a class (if the function to be called is a member function of this class).

The problem regarding overload resolution can be solved as follows: Similar to a dynamic parameter, an implicit parameter is also transformed to a parameter with a (dummy) default argument. This allows overload resolution to be performed before adding any missing arguments, i. e., by considering only the explicitly specified arguments. Afterwards, any implicit type conversions which are necessary to convert the arguments to the exact parameter types of the selected function are made explicit. (In the example given at the end of Sec. 2, the

call `f(0)`, which will be resolved to the second definition of `f` and thus requires an implicit conversion of its argument `0` from `int` to `double`, would be transformed to `f((double)0)`.) Finally, missing arguments corresponding to implicit parameters are added to the call, again using explicit conversions to the corresponding parameter types. (If the second parameter `y` of `f` would be implicit, the resulting call would be `f((double)0, (int)y)`.) By inserting explicit type conversions for all arguments of the augmented call, the process of overload resolution will in fact resolve it to the same function as the original call without any additional arguments.[7]

The problem regarding preceding parameters with default arguments can be solved in principle by encapsulating default argument expressions into parameterless auxiliary functions with unique compiler-generated internal names. Calling such a function will always execute the encapsulated expression in the context of its definition, without any interference of the calling context. Furthermore, if a default argument belongs to a member function of a class, the corresponding auxiliary function can be defined as a member function of the same class in order to have access to private and protected members of the class. By using these auxiliary functions, it is in fact possible to explicitly add all missing arguments to a function call, no matter whether they belong to parameters with default arguments, to dynamic parameters, or to implicit parameters.

However, since the actual generation of these auxiliary functions is amazingly complicated in practice (in particular for function templates, where the auxiliary functions must be templates, too, and for member functions defined outside their class, where the auxiliary functions must be predeclared in the class), it has not actually been implemented yet. As a consequence, the current prototypical, precompiler-based implementation does not allow parameters with default arguments (including dynamic parameters) to appear in a parameter list *before* a dynamic parameter. For practical applications, this does not constitute a severe restriction, since it is always possible to place implicit parameters before any other kind of optional parameter. Of course, in a real compiler, the problems discussed above do not exist at all, since it is always possible to appropriately add missing arguments to a function call in assembly or machine code.

Based on the preceding considerations, a function declaration such as

```
void print (double d, using int width, int p = 5);
```

will be transformed to

```
void print (double d, int width = *(int*)0, int p = 5);
```

where `*(int*)0` is a dummy expression of type `int`, actually dereferencing a null pointer of type `int*`. A call to `print` such as `print(d)` will be transformed to `print((double)d, (int)width)`, i. e., by explicitly adding an argument for the second parameter `width`, its default argument expression will never get executed

[7] Except in very strange situations where multiple functions with the same parameter types defined in different namespaces are visible simultaneously.

at run time. Furthermore, by not adding an explicit argument for the third parameter `p`, its default argument will be used correctly as expected.[8]

To summarize, the transformation of a function call generally proceeds as follows: First, the normal process of overload resolution is performed to select the best viable function according to the explicitly specified arguments. (For that purpose, a complete semantic analysis of the source program is necessary, which is indeed performed by the EDG C++ Front End.) Then, it is checked whether the selected function has implicit parameters whose values are not provided by the explicit arguments. If this is the case, the names and types of these parameters are used to add corresponding arguments to the call. If one of these names is not known in the calling context, or its type is incompatible with the type of the parameter, this automatically causes the Front End to issue a corresponding error message.[9] Furthermore, implicit type conversions of the explicitly specified arguments are made explicit to guarantee (in most circumstances) that the process of overload resolution will select the same function as for the original call.

In contrast to dynamic parameters, where an exact match of core types is required (cf. Sec. 3.5), implicit parameters naturally allow implicit type conversions: If the type of the entity denoted by the respective name `x` in the calling context is different from the type `T` of the implicit parameter, a corresponding conversion is performed automatically (if possible) when the augmented argument expression `(T)x` is evaluated.

### 4.3 Constructor Calls with Implicit Parameters

Constructor calls depending on implicit parameters can be transformed in exactly the same manner as calls to ordinary functions, i. e., by adding corresponding arguments, no matter whether they appear directly in so-called functional notation [10] or indirectly in variable and member initializers. For example, the declarations of `t1` and `t2` shown in Sec. 3.2 will be transformed as follows:

```
HashTable t1
  = HashTable((int)101, (int (*) (const string&))hash);
HashTable t2((int)size, (int (*) (const string&))hash);
```

In the same way, so-called mem-initializers of constructors can be transformed.

### 4.4 Invisible Constructor Calls

If a constructor call is completely invisible, it must not rely on implicit parameters in order to avoid too much implicitness and consequent incomprehensibility

[8] Basically, this could cause overload resolution to fail for the transformed call, if another definition `print (double, int)` accepting the same arguments is visible. Since such a function is not directly callable due to ambiguity, such cases are not expected to be practically relevant. In the worst case, the programmer must specify all arguments explicitly in order to select the desired function.

[9] In particular, no attempt is made in such a case to find a worse matching function that does not require these implicit parameters.

(cf. Sec. 3.2). Therefore, an appropriate error message is produced by the precompiler if such a constructor call is encountered.

## 5 Related Work and Discussion

The most obvious related work to implicit parameters as proposed in this paper are implicit parameters as proposed by Lewis et al. [7]. Even though the motivation for introducing such a concept as well as the basic idea is very similar in both cases, there are several differences in detail, however: Most obviously, Lewis et al. present their work in the realm of functional programming languages, while this paper specifically addresses imperative languages. Apart from that, Lewis et al. draw a clear syntactic distinction between implicit and regular parameters of a function: The former are not specified in the function's parameter list, but simply used in its body, where they are distinguished from other identifiers by prefixing their name with a question mark. Nevertheless, the *type* of a function, which is usually inferred from its body by the interpreter or compiler, but might also be specified explicitly in an additional *signature*, contains the information about implicit parameters. By that means, a function calling another function with implicit parameters implicitly inherits the latter's implicit parameters in its own type. Furthermore, since implicit parameters do not belong to the regular parameter list of a function, special syntax is required to explicitly pass their values in a call.

In contrast, implicit and regular parameters are treated uniformly in our approach, i. e., both are explicitly declared in a function's parameter list and both are used homogeneously in a function's body. In fact, since the implicitness of a parameter might be declared later on, there is no distinction whatsoever between implicit and regular parameters in a function's body. As a consequence of this uniformity, explicit values for implicit parameters are passed in the same way as values for regular parameters, i. e., via the normal argument list of a call.

The fact that implicit parameters are part of a function's type in both approaches enables static type checking and guarantees that functions cannot be called without directly or indirectly supplying values for all implicit parameters. The other side of the coin, i. e., the drawbacks of this tight coupling between calling and called functions, is also pointed out by Lewis et al.: If another implicit parameter is added to a function later on, its own signature as well as the signatures of all direct and indirect callers have to be modified if they have been specified explicitly. To avoid this bother, they suggest as a compromise to use ellipses to obtain signatures with only partially specified *context information*. However, since the type of a function that is inferred by the compiler still contains complete information about all implicit parameters, this approach does not really relax the tight coupling mentioned above.

For exactly this reason, *dynamic parameters* and *environment variables* are proposed in this paper as a dual concept to implicit parameters, that allows a more loose coupling between callers and callees. This part of our proposal is similar to *dynamic variables* as proposed by Hanson and Proebsting [3], again

with some important differences, however: First of all, dynamic variables have no relationship with function parameters; they are created and "exported" to the environment with a `set` statement corresponding to our export declarations, and accessed anywhere in a program with a matching `use` statement. Therefore, similar to the implicit parameters of Lewis et al., the uses of dynamic variables are "hidden" in function bodies, i. e., a function's dependency on the value of a dynamic variable is not documented in its signature. In contrast, dynamic parameters in our approach integrate the effect of a `use` statement with a parameter declaration and therefore explicitly reveal the uses of environment variables in a function.

Furthermore, in the C++ implementation of dynamic variables, their types are restricted to pointers to "polymorphic" classes [10], i. e., pointers which might be used in `dynamic_cast` operations, while dynamic parameters and environment variables might possess any C++ type. In particular, the different combinations of `const` and reference types described in Sec. 3.5 allow a very fine-grained control of "access rights" to an environment variable, ranging from completely immutable variables to those that can be modified (directly or indirectly) both in their export context and in any using context. On the other hand, dynamic variables support a more flexible matching between `set` and `use` statements by allowing a pointer to a derived class object to match a pointer to a base class, while the types of dynamic parameters and environment variables are required to match exactly except for differing `const` and reference qualifiers. In addition to the conceptual reasons for this restriction outlined in Sec. 3.5, this enables a maximally efficient implementation that does not require any kind of searching for matching variables at run time. In contrast, any implementation of dynamic variables, whether straightforward or more sophisticated, requires a linear search through the environment stack (which might be threaded through the normal run time stack) to find the first variable with a matching name and type. Even if variable names would be encoded as dummy types and used as template arguments as described in Sec. 4.1 (which is even better than any kind of hashing proposed in [3]), a search for a matching type cannot be avoided if a pointer to a derived class object shall match a pointer to a base class.

Similar to Sec. 3.6, Hanson and Proebsting also point out that dynamic variables are a data construct based on dynamic scoping, while exception handling is actually a dynamically scoped control construct. We add the observation, that the distinction between checked and unchecked exceptions has conceptual parallels to our distinction between implicit and dynamic parameters.

Other control constructs based on dynamic scoping include *control flow join points* in aspect-oriented languages [6, 8], Costanza's *dynamically scoped functions* [1], and the author's *local virtual functions* [4, 5]. As shown in [4], the latter can actually be used to simulate both exception handling and dynamically scoped variables (called semi-global variables there), even though the latter is somewhat cumbersome in practice.

The general concept of *dynamically scoped variables* can be traced back to early implementations of Lisp, where it was actually a bug instead of an intended

feature. Nevertheless, since it is still considered a useful concept in addition to the usual static scoping, the basic idea has survived in Common Lisp's *special variables* [9]. Similarly, scripting languages such as PostScript, Tcl, Perl, etc. also provide similar concepts. Finally, as already mentioned in Sec. 3.3, environment variables found in operating systems are another embodiment of basically the same idea.

Of course, any kind of implicitness in a program bears the danger of obscuring important details and thus making programs harder to understand and debug. On the other hand, however, explicitly passing around large numbers of parameters also bears the danger of obscuring a few important ones with many unimportant ones. Therefore, just like any other language construct, implicit and dynamic parameters should be used with care and perceptiveness to make programs easier to read and understand in the end.

## References

1. P. Costanza: "Dynamically Scoped Functions as the Essence of AOP." *ACM SIGPLAN Notices* 38 (8) August 2003, 29–36.
2. J. Gosling, B. Joy, G. Steele, G. Bracha: *The Java Language Specification* (Third Edition). Addison-Wesley, Reading, MA, 2005.
3. D. R. Hanson, T. A. Proebsting: "Dynamic Variables." In: *Proc. 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Snowbird, UT, June 2001). ACM, 2001, 264–273.
4. C. Heinlein: "Local Virtual Functions." In: R. Hirschfeld, R. Kowalczyk, A. Polze, M. Weske (eds.): *NODe 2005, GSEM 2005* (Erfurt, Germany, September 2005). Lecture Notes in Informatics P-69, Gesellschaft für Informatik e. V., Bonn, 2005, 129–144.
5. C. Heinlein: "Global and Local Virtual Functions in C++." *Journal of Object Technology* 4 (10) December 2005, 71–93, http://www.jot.fm/issues/issue_2005_12/article4.
6. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: "An Overview of AspectJ." In: J. Lindskov Knudsen (ed.): *ECOOP 2001 – Object-Oriented Programming* (15th European Conference; Budapest, Hungary, June 2001; Proceedings). Lecture Notes in Computer Science 2072, Springer-Verlag, Berlin, 2001, 327–353.
7. J. R. Lewis, M. B. Shields, E. Meijer, J. Launchbury: "Implicit Parameters: Dynamic Scoping with Static Types." In: *Proc. 27th ACM Symp. on Principles of Programming Languages* (Boston, MA, January 2000). ACM, 2000, 108–118.
8. O. Spinczyk, A. Gal, W. Schröder-Preikschat: "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language." In: J. Noble, J. Potter (eds.): *Proc. 40th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)* (Sydney, Australia, February 2002), 53–60.
9. G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.
10. B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.