Implicitly-Callable Functions in C++0x

Document number: WG21/N1611 = J16/04-0051
Date: February 17, 2004
Revises: None
Project: Programming Language C++
Reference: ISO/IEC IS 14882:2003(E)
Reply to: Walter E. Brown <wb@fnal.gov>
Mail Station 234
CEPA Dept., Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500
USA

Contents

1 What is an *Implicitly-Callable Function*?

In C++, invoking (calling) a function requires an explicit use of parentheses[1]. These parentheses are variously known as the *function-call operator*, as `operator()`, as the *apply operator*, or, somewhat loosely, as *function-call syntax*.

In this paper, we explore the notion of a function that is called without the use of any such parentheses. We term such a function an *Implicitly-Callable Function* (ICF, for short). In particular, in the context of an expression, any use of a function name that denotes an *Implicitly-Callable Function* results in a call to the function. We will refer to such a call as an *implicit call*.

Two observations come immediately to mind.

1. The absence of parentheses in an implicit call leaves us no syntax to provide function arguments at the call site.
2. It is not possible to distinguish an implicit call from an implicit *function-to-pointer conversion*[2], often loosely termed a *decay*.

To address these two issues, we promulgate the following two rules for *Implicitly-Callable Functions*:

1. An ICF must be *niladic*: its parameter list must be empty in order to correspond to the (implicitly) empty argument list at each implicit call.
2. An ICF never decays into a pointer-to-function; no *function-to-pointer conversion* may be applied to an ICF[3]. This leaves no ambiguity in interpreting the syntax of an implicit call: if it looks like an implicit call, it is an implicit call.

In brief, an *Implicitly-Callable Function* is a niladic function that is implicitly called whenever it is mentioned in an expression context. The remainder of this paper will explore application of this basic notion to C++ programming.

2 ICFs as interfaces to constants

In mathematics, the notion of *constant* is associated with a function that has no independent variables, and so always yields the same result whenever it is evaluated. Such a function, when graphed, thus produces a flat line (slope zero). Using the nomenclature of computing, an analogous C++ function would have no parameters and no observable side effects.

The "no observable side effects" property ensures that the function, when called, leaves behind no internal or external spoor to affect the behavior of any future call. The "no parameters" requirement ensures that no externally-supplied value, no matter how it is made available, can influence the function's behavior. This latter property rules out any reliance on an explicit argument list, of course, but also disallows use of implicit arguments (obtained, say, from input or from non-local storage à la a Fortran `COMMON` block).

A function generating pseudo-random numbers would certainly fail these requirements and so (as expected!) not qualify as yielding a constant. In contrast, consider the following trivial functions:

---

[1] "A function call is a postfix expression followed by parentheses. . . " [ISO:14882, §5.2.2, ¶1].

[2] "An lvalue of function type ... can be converted to.... a pointer to the function" [ISO:14882, §4.3, ¶1]. Informally, the name of a non-member function is treated as such an lvalue, as is the qualified name of a static member function.

[3] It is therefore not possible to obtain the address of an *Implicitly-Callable Function*. There is precedent for such a rule: C++ 98 has no pointer-to-reference types; it is thus not possible to obtain the address of a reference. Any attempt to take the address of a bound reference produces the address of the referent instead. In our context, any attempt to take the address of an ICF will be interpreted as an attempt to produce a pointer to the implicit call's result.

```
double  pi()      { return 3.1415926;  }
double  two_pi()  { return 2.0 * pi(); }
```

Wrapping constants in this way has a major payoff: we are free of the *order-of-initialization problem* that arises in C++ whenever initialization of one object depends on prior initialization of another object from a different compilation unit. It no longer matters, for example, whether the functions in the above example happen to reside in the same compilation unit: the `two_pi` function can safely rely on the value returned from `pi()`, no matter where `pi` is defined. The advantage holds:

- No matter how many constants may be involved in the definition of another constant, and
- No matter how their definitions may be distributed across compilation units.

As shown above, a client of such a wrapped constant today pays a small syntactic price, however: C++ requires function call syntax to obtain the constant's value. While most of us would likely agree that this is a minor inconvenience, our repeated (informal) user surveys of a representative programmer community clearly demonstrate that, in our context for our intended use, this need is at best deemed "unnatural" and is at worst considered to be "odious." Even though a constant can certainly be mathematically modeled via a niladic function, programmers' mindsets evidently do not permit easy application of such a model to their coding practices.

However, an *Implicitly-Callable Function* by definition needs no function-call syntax. It would therefore be the perfect solution: although it syntactically mimics an ordinary object, its use nonetheless provokes a function call. We thus obtain the best of both worlds: the syntax heartily desired by a very large community of programmers as well as the benefits of a well-defined order of initialization that requires one or more function calls to achieve:

```
double  pi()     implicit { return 3.1415926; }  // ICF
double  two_pi() implicit { return 2.0 * pi;  }  // ICF; note also the implicit call
```

For purposes of exposition, we have invented a new keyword, `implicit`. This is intended as declarative syntax; if declaration and definition are separated, the keyword is optional in the definition[4]:

```
double  pi()  implicit;
// ...
double  pi()  { return 3.1415926;  }  // ICF per its earlier declaration
```

Other implementations are certainly possible, such as this value-caching approach:

```
double  two_pi()  implicit  {
  static double const  two_pi  =  2.0 * pi;
  return two_pi;  // local variable, not implicit call
}
```

However, as demonstrated by the following simple function, choice of implementation does not affect ICF usage:

[4] Suggestions for a better ICF declarative syntax are invited.

```
double  circumference( double radius )  {
  return two_pi * radius;  // don't care about two_pi's implementation
}
```

Thus, our first application of *Implicitly-Callable Functions* is to present interfaces to constants, preserving the appearance of a simple object access, yet still obtaining the semantics of a function call in order to solve the order-of-initialization problem among dependent objects. In subsequent sections, we will extend these ICF basics to additional contexts, in each case solving additional problems.

3 ICFs as interfaces to arbitrary objects

The previous section presented a case for ICFs as interfaces to (wrappers for) constants, conveniently solving the order-of-initialization problem. In this section, we point out a straightforward extension: an ICF can wrap any object, `const`-qualified or not. The only difference is whether the implicit call produces an lvalue or an rvalue result. Here is an lvalue-returning example:

```
double &  pi()  implicit  {
  static double  pi  =  3.1;  // poor approximation
  return pi;  // reference to the local variable
}
```

This interface to the wrapped variable will permit a client to update the variable's value, *e.g.*, by assignment or by input. For our example, a client may wish to experiment with an algorithm's sensitivity based on the precision used to denote π, and so may, from time to time, need to update the variable's value to reflect various precisions of interest.

The major benefit of wrapping an arbitrary object with an ICF is the same as that of similarly wrapping a constant: the order-of-initialization problem goes away. That is such a significant improvement, that this technique of wrapping an object has almost certainly been independently rediscovered many times.

Our contribution is to make access to the wrapped object syntactically equivalent to the present access to the (unwrapped) object. Additionally, we believe it is possible to see no difference in performance: contemporary inlining technology is sufficiently powerful to eliminate even the implicit call.

Indeed, some circumstances may produce a performance improvement. Contemporary implementations of objects declared at namespace scope typically allocate and initialize all of them without considering whether they will be subsequently used. Indeed, it is not always possible to determine at compile- or at link-time whether a given such object will be used. However, instantiation of objects at block scope does not occur before the block is entered. Thus, wrapping an object with an ICF interface adheres to the zero-overhead principle in that users pay the cost of the object's instantiation (including initialization) if and only if the wrapped object is actually accessed.

4 Function templates of ICFs

We next address function templates that, when instantiated, produce *Implicitly-Callable Functions*. We believe this to be a very useful extension of the underlying concept, as the following example will show.

To motivate the utility of such extension, let us first consider a family of area-computing functions overloaded on the types of their respective `radius` parameters:

```
float        area( float       radius )  { return pi * radius * radius; }
double       area( double      radius )  { return pi * radius * radius; }
long double  area( long double radius )  { return pi * radius * radius; }
```

If, as shown, all overloads share a common `pi` entity, then two of the three overloads may well incur the cost of one or two widening or narrowing conversions, no matter which technique was used to declare and define `pi`. Further, depending on the type of that single instance of `pi`, one or two of the overloads may yield a result with less precision than otherwise possible.

If each overload were instead provided a distinct `pi` entity whose type matched the type of the function's parameter, then no conversions would be needed. This approach represents one possible trade-off between performance and computational accuracy. However, we now require additional names in order to refer to the `pi`s of the various desired types. One possible approach to selecting such names follows the naming convention of many of the functions in the C portion of the C++ standard library: use a canonical name (here, `pi`) for the `double` version, and attach distinct suffixes to denote the `float` and `long double` versions:

```
float       pif()  implicit  { return 3.14159F;           }
double      pi ()  implicit  { return 3.1415926;          }
long double pil()  implicit  { return 3.1415926535897 93L; }
```

We could then take advantage of these ICFs by rephrasing our `area` overloads to make use of the `pi` matching each `area` overload's parameter type:

```
float        area( float       radius )  { return pif * radius * radius; }
double       area( double      radius )  { return pi  * radius * radius; }
long double  area( long double radius )  { return pil * radius * radius; }
```

But suppose we prefer to provide a single generic computation, rather than a family of overloaded functions. While it seems straightforward to express most of this in the form of a function template, the desire to employ a `pi` whose result type matches the deduced function template parameter suggests we write:

```
template< class T >
T  area( T radius )  {
  return static_cast<T>(pi) * radius * radius;
}
```

Because this approach uses a single value (produced by the ICF `pi`) in all instantiations of the `area` template, it encounters the performance and precision issues described above. If, however, we could provide specializations of `pi` (*e.g.*, `pi<float>`, `pi<double>`, *etc.*) to accommodate each intended template parameter `T`, we could write our generic algorithm as:

```
template< class T >
T  area( T radius )  {
  return pi<T> * radius * radius;
}
```

which selects, at compile time, that version of `pi` that corresponds to the type of the algorithm's instantiating template parameter.

The template code to provide such a `pi` would resemble the following, expressed in a very natural syntax that combines the ICF declaration syntax presented earlier with standard syntax for declaring and specializing function templates:

```
// primary definition of ICF template function:
template< class T = double >
T  pi()  implicit  { return 3.141592653589793; }

// specializations, each an ICF based on primary declaration:
template<>  float       pi<float      >()  { return 3.14159F;           }
template<>  long double pi<long double>()  { return 3.141592653589793L; }
```

In such a context, the *Implicitly-Callable Function* `pi<T>()` very acceptably mimics a feature, historically lacking from C++, that might be termed an *Object Template* and that might plausibly have been expressed via the notation `pi<T>` — which happens to be exactly our implicit call syntax! It is in a sense analogous to a static data member of a class template, except that (yet again) we are untroubled by the order-of-initialization problem.

Finally, while all our examples in this section employ template type parameters and corresponding arguments, we do not mean to restrict ICF function templates from non-type or even template template parameters:

```
template< int I = 1 >
double  pi()  implicit  { return I * 3.1415926; }
```

We have not explored in any detail the utility of such variations but are certain our colleagues will find novel and compelling use cases to take advantage of these features.

5 *Implicitly-Callable Member Functions*

We next consider the utility of member functions that have implicit-call semantics, as this direction seems a natural extension of the underlying concept of ICFs. We will refer to such *Implicitly-Callable Member Functions*, member functions that are ICFs, as ICMFs for short.

Our first application for such ICMFs is as an implementation vehicle for the oft-requested *properties* feature. As defined in [N1384, pp. 1, 4], "A property is a conceptual attribute of an object that can be queried and modified at runtime. It differs from a regular data member in that the underlying value of the property might be computed rather than stored.... A property allows one to interact with an object in a natural, simple fashion, hiding the underlying complexity of implementation from the user." ICMFs seem to have sufficient power to address such a need and, in particular, to meet all the design goals for properties set forth in [N1384, p. 4]:

- Only those who use properties should pay for them.
- [Properties] should not interfere with current syntax and semantics.
- Declaration and usage should be intuitive to C++ users.

In the following subsections, we will present representative implementations for properties of varying degrees of complexity. As a common starting point, we present the following simple class:

```
class  Square  {
public:
  explicit Square( double s = 0.0 ) : side_(s)  { }
  // ...
private:
  double side_;  // length in cm
};
```

A ICMFs as property getters

Let us add two *Implicitly-Callable Member Functions* to our `Square` example:

```
class  Square  {
public:
  explicit Square( double s = 0.0 ) : side_(s)  { }
  double  side() const  implicit  { return side_; }
  double  area() const  implicit  { return side_ * side_; }
  // ...
private:
  double side_;  // length in cm
};
```

Such straightforward coding conveniently provides us the effect of properties with "read" attributes (*i.e.*, "getters"), for we can now write such code as:

```
Square my_square( 4.0 );  // a 4x4 square
//...
cout << my_square.side << '\t' << my_square.area;
```

Indeed, this capability exceeds that proposed in [N1384]. There, only properties are only suggested in one-to-one correspondence with data members. In contrast, as shown above, ICMFs allow not only getters (*e.g.*, `side`) for true data members, but also getters (*e.g.*, `area`) for synthesized or pseudo-data.

B ICMFs as property setters

In discussing the application of *Implicitly-Callable Member Functions* to properties with "write" attributes (*i.e.*, "setters"), we distinguish two cases. Given a data value, a setter may:

- Directly update a data member with that value.
- Filter that value before updating a data member (or several data members).

"Filtering" a value before use may involve, for example, range-checking the value, or otherwise deriving a new value from the supplied value.

The first of these cases is near-trivial to implement via ICMFs:

```
class  Square  {
public:
  explicit Square( double s = 0.0 ) : side_(s)  { }
  double &  side()  implicit  { return side_; }
  // ...
private:
  double side_;  // length in cm
};
```

The entirety of the technique is (a) to make the ICMF non-const, and (b) to have it return an lvalue. Subsequent usage would have the following very natural form:

```
Square  s;  // default-constructed
// ...
s.side = 5.0;
// ...
cin >> s.side;
```

We note in passing that this implementation technique does not produce a pure setter. Rather, it results in a combined getter/setter for non-`const` objects: the reference that it returns can be used either to read or to write a value, depending on the context in which it is used. To produce a pure setter requires a restricted form of our second case.

The second case, filtering a supplied value before use, requires a bit more infrastructure, as illustrated below. We present a combined getter/setter to mirror the previous case, but note that removing a single line of code results in a pure setter:

```
class AreaFilter  {
public:
  AreaFilter( Square s )  :  sq(s)  { }
  double  operator double() const  { return sq_.side_; }  // read
  double  operator=( double area ) const  {  // write
    sq_.side_ = std::sqrt(area):
    return area;
  }
private:
  Square &  sq_;
  AreaFilter( AreaFilter const & );
  void operator=( AreaFilter const & );
};
```

```
class  Square  {
  friend class AreaFilter;
public:
  explicit Square( double s = 0.0 ) : side_(s)  { }
  AreaProperty  area()  implicit  { return AreaProperty(*this); }
  // ...
private:
  double side_;  // length in cm
};
```

This idiomatic technique, delegating detailed work to an intermediate proxy object, has been long known in the C++ community (see, for example, [Coplien, pp. 50-52]), and so represents no new technology.

Other implementation techniques are possible, of course. For example, the filtering class (here, `AreaFilter`) could be nested within the class (here, `Square`) for which it serves as a proxy. However, the basic notion remains unaffected by such decisions.

With such an ICMF in place, usage of the `area` property mirrors that of the `side` property shown in the previous subsection:

```
Square  s;  // default-constructed
// ...
s.area = 25.0;
// ...
cin >> s.area;
```

C *Implicity-called functors*

In this subsection, we explore the extension of *Implicitly-Callable Function* concepts to function objects ("functors"). A class qualifies as an *Implicitly-Callable Functor* if and only if it has an `operator()` that is an ICMF. By analogy with an ICF, each object of such a class would have the following main property of interest: its use in the context of an expression results in an implicit call to its `operator()`.

```
class  SerialNumberGenerator  {
public:
  SerialNumberGenerator( int start_from ) : next_(start_from)  { }
  int  operator()()  implicit  { return next_++; }
private:
  int  next_;
};
```

```
SerialNumberGenerator  unique( 1 );  // instantiation
// ...
cout << unique;  // use
```

6 A few musings

A Overloading

Because ICFs take no arguments, there's no way to write multiple *Implicitly-Callable Functions* by the same name, yet with distinct signatures. This leaves `const` and non-`const` versions of an *Implicitly-Callable Member Function* as the sole source for overloading.

For the record, there is also a second reason to forbid overloads involving the name of an ICF. Consider the call `f(a,b,c)`, where:

- `f` is an ICF,
- `f` returns an object of class type,
- that class includes an `operator()` member function, and
- that member function has parameters compatible with the `(a,b,c)` argument list shown above.

Given such a scenario, the above call `f(a,b,c)` would be ambiguous if `f` were also permitted to participate in overload resolution:

- The expression could be legitimately interpreted as a call to an appropriate (3-parameter) overload of `f`, or,
- The expression could equally legitimately be treated as a call to the ICF `f`, immediately followed by a call to the result's `operator()`.

To prevent such ambiguity, we disallow *Implicitly-Callable Functions* from having overloads, thus favoring the latter interpretation.

B Address-of an ICF

Since we prohibited obtaining a pointer-to-function from an ICF `f`, how is `&f` interpreted?

As we pointed out near the beginning of this paper, this expression would have been ambiguous in the absence of a rule (such as we proposed) to forbid obtaining the address of an ICF. Therefore, `&f` yields the address of the result produced by the implicit call to `f`.

The same analysis seems to hold for the case of an ICMF. Without a special rule, an expression such as `&x.f` would have also been ambiguous. It remains to be seen whether this rule poses a serious restriction in practice.

If so, an alternate formulation would remove the ambiguity, yet produce a different irregularity. We could promulgate a rule that inhibits any implicit call following an address-of operator. Following such a rule, `&f` would indeed yield a pointer-to-function. However this would introduce a different irregularity: we could no longer form a pointer to the result of an implicit call. This is probably not a serious limitation if the ICF returns an rvalue, but could be problematic in the case of lvalue-returning ICFs.

A third possibility would introduce another new concept into the language: the notion of a *property pointer*, a form of generalized member pointer. Such a property pointer could refer either to a data member or to an ICMF of a class, and so from a user's perspective would blur the distinction between the two. In the following brief illustration, we declare such a hypothetical property pointer object using a fictional declarator `@`, obtaining such a pointer from a property via a corresponding `operator@`:

```
class Square;
double Square::@  mp  // property pointer yielding a double when dereferenced
  = @Square::side;    // no distinction between data member and ICMF;
                      // if an ICMF, it must not here be implicitly called
//...
Square s;
cout << s.*mp;
```

To be clear, we are not proposing such a property pointer construct. We simply point out its possibility for the sake of completeness.

C Virtual ICMFs

A final variation on *Implicitly-Callable Member Functions* is the option to declare such functions in the context of an inheritance hierarchy, and to take advantage of polymorphic behavior. We regard this capability as very desirable. Note that, just as the keyword `virtual` is optional in declaring overriding functions in derived class, so is the keyword `implicit` optional under the same circumstances:

```
class  Shape  {  // base class
public:
  virtual double  area() const  implicit  = 0;
  // ...
};
```

```
class  Square  :  public Shape  {  // derived class #1
public:
  virtual double  area() const  implicit  { return side_ * side_; }
  // ...
private:
  double  side_;
};
```

```
class  Circle  :  public Shape  {  // derived class #2
public:
  // virtual and implicit, based on declaration in Shape class:
  double  area() const  { return pi * radius_ * radius_; }
  // ...
private:
  double  radius_;
};
```

```
// representative use case:
double  f( Shape const & s )  { return ... s.area ...; }
```

D Forwarding

ICFs appear also to be useful in addressing certain kinds of forwarding problems. While we have not investigated such application in any detail, the following code sketch comes quickly to mind as a starting prototype for further study and analysis:

```
double  really_do_it( char const *, bool, float);
/...
typedef  double (*Doer)( char const *, bool, float);
Doer  do_it()  implicit  { return  &really_do_it; }

// user code:
cout << do_it( "Hi", true, 3.14F );
```

E About nomenclature

It has been pointed out to us that the term *Implicitly-Callable Function* and its related terms *Implicitly-Callable Member Function* and *Implicitly-Callable Functor* are all strongly indicative of the behavior of these proposed features. Alternative nomenclature that focuses on their application has been suggested as desirable. The terms *attribute* and/or *attribute functions* have been proposed as replacements, and so are herewith submitted for consideration.

7 Conclusion

This paper has presented a new concept and language feature, *Implicitly-Callable Functions*. We have described the intended semantics of this feature, and have set forth a notional syntax for its use in C++ programs. Further, we have exhibited a number of case studies in which ICFs have successfully applied to programming scenarios of significant interest and utility.

These case studies have demonstrated rather broad applicability for ICFs. We believe the concept shows considerable potential to address even more situations of common utility. We seek and will welcome additional feedback regarding this new facility, and respectfully urge the C++ programming community to give strong consideration to this feature for inclusion in C++ 0x.

8 Acknowledgments

We are grateful to the many individuals who (a) have helped us, over many years, to refine the ideas underlying this proposal, and (b) have reviewed this document's expression of those ideas.

While there are too many to name individually, we do wish to acknowledge, with special thanks, the contributions of (in alphabetical order) Jim Kowalkowsi, Marc Paterno, Bjarne Stroustrup, and David Vandevoorde.

We also appreciate the support of the Fermi National Accelerator Laboratory's Computing Division, sponsors of our participation in the C++ standards effort.

Bibliography

[N1384] Borland Software Corp.: *PME: Properties, Methods and Events*. WG21/N1384 (same as J16/02-0042): 2002.

[Coplien] Coplien, James O.: *Advanced C++ Styles and Idioms*. Addison-Wesley, 1992. Reprinted with corrections, 1994. ISBN 0-201-54855-0.

[ISO:14882] International Standards Organization: *Programming Languages — C++* . International Standard ISO/IEC 14882:1998.