Lexical Closures for C++

Thomas M. Breuel *

Abstract

We describe an extension of the C++ programming language that allows the nesting of function definitions and provides lexical closures with dynamic lifetime.

Our primary motivation for this extension is that it allows the programmer to define iterators for collection classes simply as member functions. Such iterators take function pointers or closures as arguments; providing lexical closures lets one express state (e.g. accumulators) naturally and easily. This technique is commonplace in programming languages like Scheme, T, or Smalltalk-80, and is probably the most concise and natural way to provide generic iteration constructs in object oriented programming languages. The ability to nest function definitions also encourages a modular programming style.

We would like to extend the C++ language in this way without introducing new data types for closures and without affecting the efficiency of programs that do not use the feature. In order to achieve this, we propose that when a closure is created, a short segment of code is generated that loads the static chain pointer and jumps to the body of the function. A closure is a pointer to this short segment of code. This trick allows us to treat a closure the same way as a pointer to an ordinary C++ function that does not reference any non-local, non-global variables.

We discuss issues of consistency with existing scoping rules, syntax, allocation strategies, portability, and efficiency.

1 What we would like to add... and why

We would like to be able to nest function definitions in C++ programs. In this section, we will discuss a number of reasons why the ability to nest function definitions is desirable. Almost all other modern programming languages such as Scheme, T, Smalltalk-80, Common Lisp, Pascal, Modula-2, and Ada offer this feature.

To illustrate the utility of this language feature, we have to agree on syntax. To indicate that a function is defined inside another function, we will simply move its definition inside that function but otherwise write it the same way we would at global level. For example, the following fragment of code defines `function2(int)` inside `function1(int)`:

```
function1(int x) {
    // ...
    function2(int y) {
        // ...
    }
    // ...
}
```

*Author's address: MIT Artificial Intelligence Laboratory, Room 711, 545 Technology Square, Cambridge, MA 02139, USA. The author was supported by a fellowship from the Fairchild foundation.

Unless `function2` declares the identifier `x` itself, any mention of `x` inside `function2` will refer to the argument of `function1`.

To allow the definition of mutually recursive functions at inner lexical levels, it is necessary to provide some way of declaring functions that are not defined at global level. We suggest that this is done by prefixing the function declaration by the keyword `auto`. Currently, it is illegal to use the keyword `auto` before a function declaration. This extension is therefore compatible.

1.1 Nesting and Modularity

The ability to nest function definitions encourages modular design of programs. It allows the programmer to keep functions and variables that are only used by one function local to that function. In C, such modularity is only possible at the level of files: the scope of the identifier of a function can be limited to a file by declaring it `static`. The scope of a variable's identifier that is to be shared among functions must encompass at least a compilation unit since it must be declared at global level. C++ supports limiting the scope of certain kinds of identifiers for functions and variables to member functions by making those functions and variables members of a class.

However, it is often not natural to introduce a new class simply to limit the scope of an identifier. Consider, for example, the heapsort algorithm[Wir79]. It consists of a `sort` function that repeatedly calls a `sift` function to insert elements into a heap. In C or C++, we would express this as follows:

```
sift(int* v,int x,int l,int r) {
   // insert x into the heap
   // formed by elements l...r of v
}

// sort an array of integers v
// v is n elements long

heapsort(int* v,int n) {
   // code that calls sift
}
```

This is unsatisfactory, however, because the function `sift` is unlikely to be of any use anywhere else in the program. The function `sift` ought to be visible only inside the function `heapsort`. Furthermore, we would like to refer to the variable `v` inside `sift` without having to pass it as a parameter. Nesting allows us to rewrite this as:

```
// sort an array of integers v
// v is n elements long

heapsort(int* v,int n) {
   sift(int x,int l,int r) {
       // insert x into the heap
       // formed by elements l...r of v
   }
   // code that calls sift
}
```

Notice in particular that any use of the identifier `v` inside the function `sift` refers to the argument `v` of the lexically enclosing function `heapsort`[1].

As another example, assume that we have a function `integrate` that integrates a given function between two bounds and we would like to integrate a parameterized family of functions. Again, the most natural way of expressing this is as follows:

```
integrate_all(int n,double* as,double *bs,double *cs,double eps) {
    double integrate(double low,double high,
                     double epsilon,double (*f)(double));
    double a,b,c;
    double f(double x) {
        return a*x*x+b*x+c;
    }
    for(int i=0;i<n;i++) {
        a=as[i]; b=bs[i]; c=cs[i];
        printf("a: %g, b: %g, c: %g, i: %g\n",
               a,b,c,integrate(0.0,1.0,eps,f));
    }
}
```

1.2 Iterators

The ability to nest function definitions and to reference variables declared by enclosing blocks is particularly useful together with iterators over collection classes. Consider the following simple collection class:

```
class BagOfInts {
public:
   // add an int to the collection
   void add(int);

   // test whether an int
   // is in the collection
   int member(int);

   // apply a function to every
   // element in the collection
   void walk(void (*)(int));
};
```

For example, to print all the elements in a bag, we would write in standard C++:

```
printit(int x) {
   printf("integer in bag: %d\n",x);
}

main() {
   BagOfInts aBag;
```

[1] unless `sift` declares or defines another identifier `v`

```
    ...
    aBag.walk(printit);
}
```

Note that we are forced to define the function `printit` far away from the place where it is actually used. The reason why we made `printit` a function in the first place is not that it is a useful abstraction of some process or that we are going to use it in several places, but simply because the member function `walk` demands a function pointer as its argument.

Even more disturbing is that in standard C++ the only side effects a function that is passed as an argument to the iterator can have are to static variables or global variables. If we would like to use the iterator `walk` to sum all the elements in a bag, we would have to use a global variable:

```
int xxx_counter;

int xxx_count(int x) {
    xxx_counter+=x;
}

fizzle() {
    BagOfInts aBag;
    ...
    xxx_counter=0;
    aBag.walk(xxx_count);
    int sum=xxx_counter;
    ...
}
```

If we were allowed to nest function definitions, we could express this simply as:

```
fizzle() {
    BagOfInts aBag;
    ...
    int sum=0;
    count(int x) {
        sum+=x;
    }
    aBag.walk(count);
    ...
}
```

Now, all the identifiers are declared and defined exactly where they are used. No identifiers appear at global scope that should not be visible at global scope[2]. And, no global data space is wasted for the counter; the space for the counter exists only while function `fizzle` is active.

This approach to writing iterators for user-defined classes is actually very commonly used in Smalltalk and Lisp-like languages. For example, it Smalltalk-80[GR83], we would express the function `fizzle` as follows:

[2]It is still disturbing that we had to invent a name for the function `count`. We will suggest syntax to define unnamed functions in later sections. We used the named version here in order to avoid getting into questions of syntax at this point.

```
fizzle
   | sum |
   ...
   sum <- 0.
   aBag do: [x | sum <- sum + x].
   ...
```

And in CommonLisp[Ste84] we might write:

```
(define fizzle ()
   (let ((aBag ...)
         (sum 0))
      ...
      (map nil #'(lambda (x) (incf sum x)))
      ...))
```

The reader might ask whether there are alternative approaches to iteration just using standard C++ constructs. We will describe one of them shortly. Let us first remind ourselves, though, what we ask of an iteration construct in any language.

An iterator is basically a construct that takes a piece of code and invokes it repeatedly and changes the values of some bindings in the environment of that piece of code. From the built-in iterators of C and C++ we are used to being able to do the following.

1. We can write down the piece of code that is to be executed repeatedly *at the point* in the source code where the iteration is performed.

2. The code that is executed repeatedly can reference identifiers whose scope is limited to the enclosing function.

3. The names of variables modified by the iteration construct can be chosen freely.

These properties are very useful features of built-in iterators, and it would be most unfortunate to have to give up any one of them for user-defined operators.

One way to achieve these goals it to introduce a new class together with a macro, as follows:

```
class BagOfIntsStepper {
public:
   int more();
   int next();
};

#define iterateBagOfInts(bag,var) \
   for(BagOfIntsStepper stepper=bag.makeStepper(); \
       stepper.more(); \
       var=stepper.next())
```

This construct does not violate any of the above constraints. However, it has several disadvantages. It requires us to introduce a new class for the purpose of expressing iteration, it requires two calls to member functions per iteration step, and it requires us to define a macro. Most of all, however, we dislike about it that it is difficult to extend to the case when

we want to iterate over different collection classes with the same base type[3]. For example, we might also have a class `SetOfInts`. If we used the `walk` style iteration construct, we could simply make `BagOfInts` and `SetOfInts` subclasses of a class `CollectionOfInts` and declare `walk` to be virtual. To achieve the same effect with a stepper class is much less straightforward and might also require the extra overhead of two virtual function call for each iteration step.

2 How to Implement It

In order to implement nesting and lexical closures in C++, we have to introduce a static link chain that links each activation record to the correct activation record for the lexically enclosing function (see [AU79] and [Wir77] for terminology). When we invoke a function, we not only have to know its address, but we also have to pass along a pointer to the correct activation record for the lexically enclosing function. There are two exceptions to this rule, however.

No space for a static link pointer needs to be reserved in the activation record of a function defined at global level because its lexical environment is known to be the global environment and because it is clear at compile time whether an identifier refers to a global variable.

Furthermore, a function defined at global level does not need to be passed a pointer to the lexically enclosing environment, because the lexically enclosing environment is the global environment which is unique and has a known address.

These two exceptions together with the fact that C++ allows functions to be defined only at global level make it possible to implement lexical scoping in C++ without using a display, a static link chain, or passing around pointers to environments.

Adding space for the static link chain to the activation record of functions that are not defined at the global level is trivial (they can be thought of simply as additional automatic variables). Since the compiler always knows which absolute lexical level an activation record corresponds to, it is not a problem that the activation record for functions at the outermost lexical level differs from that of activation records of functions at other lexical levels.

What does present a significant problem is the fact that instead of just a pointer to code, the invocation of a function at an inner lexical level in addition requires a pointer to the proper activation record of the lexically enclosing function.

If we want to be able to pass around pointers to functions at inner lexical levels freely, this information must be passed around together with the pointer to the code of the function, since this information cannot be derived in any other way.

As long as we are not using functions as arguments to other functions or assign functions to variables, the compiler can silently take care of making sure that the information about the proper lexical environment is passed along to the callee. However, as soon as we try to pass around functions as data, we encounter a problem. In C++, only enough space for one pointer needs to be reserved and passed around when defining and using a function pointer. However, a closure, i.e. a function with an environment requires two pointers in general, as we have just seen.

Changing the representation of a C++ function pointer to be two pointers large rather than one pointer is unacceptable. This would mean, for example, that a "function pointer" could not be assigned to a variable of type `void*` without losing information.

[3]The base type of a collection is the type of the elements of the collection.

An alternative might be to introduce a new data type, "closure", that must be used to express references to functions and their environment if those functions are not defined at global level. Using implicit type conversions from function pointers to closures would allow us to mix function pointers could be used in place of closures. However, closures could not be passed to existing functions that expect function pointers. Furthermore, the language becomes unnecessarily cluttered by two data types for essentially the same concept.

Clearly, neither of these alternatives is acceptable. Fortunately, there is a simple and efficient solution. In fact, our solution is completely compatible with C++ and C. In particular:

1. our closures can be used anywhere a C or C++ function pointer can be used, even if the code using the closure was compiled with a compiler that does not know about closures

2. the code generated for functions that do not contain nested functions does not change when closures are added to the compiler

The code segment that generates a closure does so by generating a short segment of code that loads the static link pointer into some known register and jumps to the function (lines 0043-0046 in the listing below). At the beginning of a function, the contents of the register is moved into the static link field of the newly created activation record (line 0076). In an assembly language similar to 68000 assembly language[4], this would look something like:

```
0000  ;;; source code:
0001  ;;;
0002  ;;; function1() {
0003  ;;;     function2() {
0004  ;;;             ...
0005  ;;;     }
0006  ;;;     void (*x)();
0007  ;;;     x=function2;
0008  ;;;     ...
0009  ;;; }
0010
0011  ;;; machine registers:
0012  ;;; FP: current frame
0013  ;;; SL: a register that is used to hold the static link pointer
0014  ;;;     temporarily
0015
0016  ;;; instruction to load the SL register with constant
0017  INST_LOADSL     equ 0x77777777
0018
0019  ;;; instruction for direct jump
0020  INST_JUMP       equ 0x88888888
0021
```

[4] Assume that all data and instructions are 32bits wide. Temporary labels begin with a "`$`". The instruction `moveea` moves the effective address of its first operand into a location. The `link` instruction fills in the "`$dl`" field of the activation record.

```
0022  function1:
0023          ;;; allocate space for the following variables:
0024          ;;; void* $dl -- dynamic link chain
0025          ;;; void* $sl -- static link chain
0026          ;;; int $stub[4] -- space for machine code for closure
0027          ;;; int (*x)() -- a function pointer
0028          ;;; also sets up the dynamic link chain
0029
0030  $dl       equ -4
0031  $sl       equ -8
0032  $stub     equ -24
0033  function1.x       equ -28
0034
0035          link FP,28
0036
0037          ;;; set up the static link chain
0038
0039          move SL,$sl(FP)
0040
0041          ;;; create closure for function2
0042
0043          move #INST_LOADSL,$stub(FP)
0044          move FP,$stub+4(FP)
0045          move #INST_JUMP,$stub+8(FP)
0046          moveea function2,$stub+12(FP)
0047
0048          ;;; x=function2
0049
0050          moveea stub(FP),function1.x(FP)
0051          ...
0052
0053          ;;; code using x
0054
0055          move function1.x(FP),R1
0056          call (R1)
0057          ...
0058
0059          ;;; finish up
0060
0061          unlink FP
0062          return
0063
0064  function2:
0065          ;;; function entry, as above
0066
0067          link 4
0068
0069  $dl     equ -4
```

```
0070  $sl     equ -8
0071
0072          ;;; set up the static link chain
0073          ;;; register SL was set up by the stub
0074          ;;; code in function1
0075
0076          move SL,$sl
0077          ...
0078
0079          unlink FP
0080          return
```

The lifetime of the code array `$stub` is identical to the lifetime of the activation record of `function1`. This is reasonable, since the static link chain implicitely defined by `$stub` becomes invalid as soon as `function1` exits[5].

This is the basic idea. There are several compile-time optimizations possible, some of which we have already mentioned. If `function1` is defined at global level, for example, it does not need to reserve space for a static link chain in its activation record. If `function2` does not reference any variables in the activation record of `function1`, the compiler can leave the activation record of `function1` out of the static link chain handed to `function2`. If `function2` does not reference any non-local, non-global variables, its definition can effectively be moved to the global level (except for the scope of its name), and neither a static link field nor a code stub needs to be generated inside `function1`. If `function2` is only used within `function1` and no closure is passed around, the code generating the stub code can be eliminated since the compiler can generate code to load register `SL` inline just before calls to `function2`[6].

3 How Efficient is It?

To see how efficient this scheme is, we have to compare it with alternative implementations. The two most straightforward implementations are to represent a closure as either a structure with two elements, a pointer to code and a pointer to the environment, or as a pointer to such a structure.

The additional space required by our scheme consists of only the two machine instructions contained in the stub code, and the two instructions inside `function1` used to generate the two machine instructions in the stub code. Assigning and passing closures is as efficient as in the case where we use a pointer to a structure.

However, the overhead of closure creation is comparatively unimportant. As we have seen above, the compiler has to create a closure only if a function pointer is to be passed around. In our experience, a function pointer that is passed around is usually used repeatedly, so the overhead of invoking a closure is much more important than the overhead of creating one.

Let us look at the instruction sequences that are executed in each of the three different implementations of closures. First, here is the instruction sequence for closures that are

[5] We will later discuss possible extensions to extend the lifetime of a particular static link chain beyond the dynamic lifetime of the component activation records.

[6] In fact, a restricted form of nesting where we disallow taking the address of functions defined at an inner lexical levels can be implemented without extending the compiler to generate code stubs.

represented directly as structures[7]. The closure consists of two machine words (pointers) at offset `x` in the current activation record [8]:

```
move x(FR),SL
move x+4(FR),R1
call (R1)
```

In the case of a pointer to a closure, the calling sequence becomes a bit more complex. Assume that `px` is the offset of the pointer to the closure in the current activation record[9]:

```
move px(FR),R1
move (R1),SL
move 4(R1),R1
call (R1)
```

For our proposal, the sequence of instructions encountered is:

```
move px(FR),R1
call (R1)
move #staticLink,SL
jmp function
```

We see that the instruction sequences executed when a function pointer to a function requiring a static chain pointer is called are not too different. The major differences in efficiency will probably come from cache and instruction prefetch effects. Some simple experimentation indicates that on the 80386 a call/return pair that passes a static link pointer along takes about 1.5 times as long as a simple indirect or direct function call. If we use our proposed instruction sequence, this figure is increased to about 1.8 times.

4 Issues of Portability

To implement our proposal, it is necessary for a piece of code to be able to generate short pieces of code at runtime and execute them; the generated code does not necessarily have to be placed on the stack, however.

This is possible and straightforward on most modern computer architectures like the 68000 series of microprocessors[Mot80], the VAX[Dig81b], and the 80386[Int87]. Even on the PDP-11 series processors with separate instruction and data space ("I and D spaces"), the stack is ordinarily mapped into both I and D space to permit execution of instructions on the stack (in the standard PDP-11 instruction calling sequence, the `MARK` instruction is executed off the stack, see the PDP-11 processor handbook[Dig81a]).

There are, however, some architectures and/or operating systems that forbid a program to generate and execute code at runtime. We consider this restriction arbitrary[10] and consider it poor hardware or software design. Implementations of programming languages

[7] These are not assembly language program fragments but traces of the assembly language instructions executed during invocation of a closure.

[8] `R1` is some general purpose register.

[9] The contents of locations `(R1)` and `4(R1)` in the following example could conceivably be cached.

[10] Such systems usually provide operating system calls to move data into the instruction space, for example for the benefit of a loader; however, the overhead of an operating system call is too high for the creation of a closure.

such as FORTH, Lisp, or Smalltalk can benefit significantly from the ability to generate or modify code quickly at runtime.

We can use another trick to implement lexical closures even on these architectures. We pre-allocate in instruction space an array of instruction sequence of the form:

```
stub_n    move location_n,R1
          move (R1),SL
          move location_n+4,R1
          jmp (R1)
```

We use this array as a stack to allocate and deallocate closure stubs. A corresponding array of locations in data space holds the actual pointers to the code and the static link chain of the closures. These two new stacks behave essentially like the runtime stack. In particular, `longjmp` must be modified to restore the two stack pointers for the stub stack and the location stack appropriately.

5 Further Extensions

We observed in some of the above examples that often there is no need to name a function explicitly. This is particularly so when we use iterators that take function pointers as arguments. The examples of Lisp and Smalltalk code given above involve unnamed functions. We suggest to express an unnamed function as a cast of a *compound-statement* to a function pointer. For example, the value of the following *expression* is a function pointer or a closure (depending on the context):

```
(int (*)(int x)){ return x+1; }
```

Alternatively, we could introduce a new keyword, `unnamed`, and write the same construct as

```
(int unnamed(int x){ return x+1; })
```

We prefer the first form slightly, but the second form may be easier to parse and allow better syntax error detection and recovery.

The way we have proposed to implement closures limits their lifetime to that of the activation record in which they were created. In order to make closures a data type of the same standing as any other data type in C++, it should be possible to allocate and deallocate closures. There are good reasons to provide closures that can be treated as data structures. Abelson and Sussman[AS85] argue strongly for this feature, and in the Scheme programming language[RC86] they are often used to build complex data types. However, classes and structures provide much of the functionality of heap-allocated closures.

6 Conclusions

The ability to nest function definitions and to create lexical closures with at least dynamic lifetime is an important part of many modern programming styles. Most modern programming languages provide it, and it can be incorporated into the C++ (and C) programming language without affecting the efficiency of execution or meaning of programs that do not take advantage of the feature. We would therefore like to see nested function definitions

and lexical closures to be incorporated into the C++ language definition. We are currently working on extending the GNU C[Sta88a][Sta88b] and C++[Tie88] compilers to provide nested function definitions and closures. The availability of a good, free compiler with source level debugger should encourage more people to use this feature.

Acknowledgements

I would like to thank Richard M. Stallman, Robert S. Thau, and many others who have made useful comments on the proposal and the paper.

References

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1979.

[Dig81a] Digital Equipment Corporation. *PDP-11 Processor Handbook*, 1981.

[Dig81b] Digital Equipment Corporation. *VAX Architecture Handbook*, 1981.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Int87] Intel Corporation. *80386 Programmer's Reference Manual*, 1987.

[Mot80] Motorola Semiconductors Products, Inc. *MC68000 16-bit Microprocessor User's Manual*, 1980.

[RC86] Jonathan Rees and William Clinger. The Revised^3 Report on the Algorithmic Language Scheme. Technical Report 848a, MIT Artificial Intelligence Laboratory, September 1986.

[Sta88a] Richard M. Stallman. *Internals of GNU CC*, April 1988.

[Sta88b] Richard M. Stallman. *The GNU Debugger for GNU C++ Free Software*, April 1988.

[Ste84] Guy L. Steele Jr. *Common LISP: The Language*. Digital Press, 1984.

[Tie88] Michael D. Tiemann. *User's Guide to GNU C++*, May 1988.

[Wir77] Niklaus Wirth. *Compilerbau*. B. G. Teubner, Stuttgart, 1977.

[Wir79] Niklaus Wirth. *Algorithmen und Datenstrukturen*. B. G. Teubner, Stuttgart, 1979.