ISO Document WG21/N0295
ANSI Document X3J16/93-0088
Author: John Max Skaller
Reply to: maxtal@suphys.physics.su.oz.au
Coauthor: Fergus Henderson

` Date 6/4/93

A proposal for NESTED FUNCTIONS

Abstract

We discuss the introduction of nested functions into C++. Nested functions are well understood and their introduction requires little effort from either compiler vendors, programmers, or the committee. Nested functions offer significant advantages, including use for rapid prototyping and functional decomposition, as well as a bridge from a procedural to object orieneted style, and offer significant gains in both processor and programmer performance. We believe the overall cost/benefit ratio warrants their introduction into the Working Paper at this time. These advantages include:

a) Natural completion
b) Familiarity
c) Rapid prototyping
d) Helper functions
e) Functional decomposition
f) Bridge to object oriented programming
g) Efficiency
h) Localised cost

We discuss closures of nested functions and bound members of objects. While both these facilities offer significant advantages, urgency of other work suggests that the relevant syntax be specified as implementation defined, particularly as these features would be greatly enhanced by garbage collection.

1. Syntax and Semantics

In short, this paper suggests it be allowed to define functions in the scope of other functions, including member functions. This elevates C++ to the status of a true static block structured language, and provides it the same scoping mechanism as is available in Pascal.

The rule for nested functions are surprisingly simple, and do not appear to raise any particularly difficult issues. Basically, nested functions must be either auto or static, with the usual restrictions to access this implies.

```
int f(int a) {
        static int count=0;
        static int inc() { return count++; }

        int b=inc();
        int g(int c) { return a+b+c; }

        return inc()+g();
}
```

Functions nested directly or indirectly in member functions are also member functions, the major difficulty here being terminology: there arises the possibility of a function which is not static, yet is a static member function: such a function might be termed non-non-static.

Declarations of nested functions are allowed, but they must be defined in the same region as their declaration. It is not allowed to initialise an accessible variable between a nested function declaration and its definition, as might be equivalent to jumping past an initialisation.

Nested functions may be considered as members of the object which is the stack frame of the enclosing function. As such there is a natural syntax for obtaining their address, which is an abbreviated form of the pointer to member syntax. Such an address binds to the current context on use, and the syntax for it such that it is not possible to pass such a pointer to nested function into any scope in which a call would be unsafe.

Functions can be considered classes or namespaces in the sense their stack frames are objects, and the class qualifier syntax can be used to access hidden identifiers.

1.1. Storage Class and linkage specifier.

1.1.1 One of the following combinations must be given for a declaration:

Specifier	*Type*	*meaning*
(none)	extern	external function declaration
`extern`	extern	external function declaration
`auto`	inline auto	ordinary nested function
`static`	inline static	static nested function

The default of extern is required for compatibility, functions declared `extern` are not nested functions, and no nested definition may be given.

1.1.2 For a function defintion, the default is `auto`.

1.1.3 For a function definition, `inline` may also be specified, but acts only as a hint to the compiler that particularly aggressive optimisation may be desirable: all nested functions may be inlined once the body of the function is seen by the compiler.

1.1.4 A declaration may not conflict with a definition, or another declaration in determining whether the function is extern, auto, or static.

1.1.5 Nested functions may not be declared or defined without specifying a return type, that is, the default int return type is specifically disallowed.

1.1.6 No accessible variables may be initialised between a function declaration and a subsequent definition. This would be equivalent to jumping over an initalisation, and is disallowed for the same reason. The meaning of 'accessible' is described below.

```
void f() {
        auto void after();
        after();
        int x=1;                        // illegal initialisation
        void after() {x; }
}

void f() {
        static void after();
        after();
        int x=1;                        // OK
        static void after() {x; }       // error: x is not accessible
}
```

1.1.7 Nested functions have internal linkage

1.2. Scope of names and overloading

1.2.1 If a nested function is declared in a region (block, scope) , there must be a subsequent definition in the same region.

1.2.2 Nested functions with the same name declared in the same region overload each other.

1.2.3 Names of nested functions in an inner region hide the name of any identifier in an enclosing region, as usual.

1.2.4 A function is a namespace when view from inside the function. In particular, a deeply nested function may refer to a hidden outer name by using class name qualifier syntax. A nested function name followed by `::` is used as a class or namespace name for lookup purposes if it is a function, whereas without the class qualifier syntax it is used as a function name.

```
// file scope
static f() {
  static g() {
    static h() {
      static g() {
        f::g();
        ::f::g();
}}}}
```

1.2.5 It is not permitted to hide the name of a function directly in the body of the function: this is the namespace name injection rule. A function name is implicitly injected into its body, however that injected name is not accessed when the class qualifier syntax is used. (This rule should be accepted if and only if the equivalent rule for class name injection is accepted).

1.2.6 A name from an enlosing function may be injected into a local scope with a using definition. As for classes, using directives may not be given.

```
f(int) { ... }
void g(){
  int i;
  void f() {
    int i;
    void g() {
      int i;
      void h() {
        using f::i;
        using ::f;
}}}}
```

1.3. Access

1.3.1 An auto nested function gains access to all visible and accessible objects and names of the block it is nested in.

1.3.2 A static nested function gains access to all the accessible static objects and names of the block it is nested in.

These two rules are complete, but we will give some special cases as examples.

A function nested directly or indirectly in a member function is a member function. By considering the chain of functions from the member function down to the nested function in question, one can determine that the function is a non-static member function only if all the enclosing functions are also non-static member functions, that is, all are specified auto (except the original member function, which may be considered to be auto by default).

```
class X {
  void f();
  int i;
```

```
  static int j;
};

void X::f() {
  void g() { ++i; }
  static void h() {
    int q=1;
    int k() { ++q; ++j; }
    ++j;
  }
}
```

If even one of these functions is static, the function is a static member function, however it need not itself be a static function: it may be an auto function enclosed in a static member. This is terminologically confusing: the function is not static, it is a member, yet it is a static member: one could say it is a non-non-static member.

A function nested in a friend function is also a friend function: in this case friendship is inherited.

The rule about initialising accessible variables between a declaration and a defintion can now be appreciated: there is no restriction on initialising an auto variable between the declaration and definition of a static nested function.

1.4. Semantics

1.4.1 A goto in a nested function may not jump out of the function. This restriction against non-local gotos is for simplicity. Exceptions can be used instead.

1.4.2 An auto nested function always refers to the most recent activation of the surrounding scope.

1.4.4 The address of a static nested function may be taken and used without restriction, it is an ordinary function pointer.

1.4.3 The address of a nested function may be taken, the use of the unary & operator is optional. The resulting object is of a type similar to a 'pointer to member', and is represented simply as the physical address of the function.

1.4.4 A variable of type 'pointer to function nested in a function' may be declared and may be initialised with a pointer to nested function, or have such a pointer assigned to it. The syntax is illustrated by

```
int func()
{
       auto int nest(long) { .. }
       auto int nest(int) { .. }
       int (func::*var)(long) = nest;
       int result = (*var)(3L); //1
       int result = var(3L);  //2
}
```

The function name specified where the class name is normally used in the pointer to member declarator, in this case 'func', must refer to an enclosing function, class name lookup is used so that such a reference can never be ambiguous. Qualification may be used if necessary to access a hidden function. The usual context sensitive mechanism of overload resolution is applied to determining which of a set of overloaded function is refered to. The nested function use to initialise the variable must be directly nested in the named function (and of course must be visible and accessible).

The invocation of a nested function does not require the enclosing function name to be repeated, since the name is implicit in the name of the pointer: the most recent activation record of the function is bound to at the time and point of the call.

The pointer to nested function may be represented by an ordinary function pointer on many machines, since the binding to the current context is usually done in the prolog of the function itself.

Because the qualifying part of the pointer to nested function syntax must refer to an enclosing function, it is syntactically impossible to pass such a pointer out of the scope in which there exists a current activation record, so that no possibility of refering to a non-existent stack frame exists: the mechanism is completely secure with the usual caveat that invoking an uninitialised pointer has unpredictable results.

Because it is not legal to declare a function and a variable with the same name in the same region, the syntax allows eliding the usual derefencing as shown above at (2).

(Comment: We have tested the lookup mechanism with a computer program, and the mechanism is complete and consistent, although my description of it may need language lawyering. I used 'class name injection' in the test, I have not tested it without class name injection.)

1.5. Nested Function Closures

It is necessary to specify what happens if a pointer to nested function is copied to a pointer to an ordinary function (with the same signature), or, equivalently, if a reference to such is formed.

This is done implicitly when invoking a nested function via a pointer to nested function.

The explicit case is demonstrated by

```
int func()
{
        auto int nest(long) { .. }
        int (func::*var)(long) = nest;
        int (*pclos)(long) = nest;
}
```

in which case pclos is an ordinary function pointer that points to a closure, that is, an ordinary function in which the context is bound in at the time and point of the conversion. Such a function may be called a bound nested function or a closure.

It is possible to implement closures using either dynamic function generation (known as trampolines or thunks) or by using two addresses in all function pointers. Such a pointer to a closure may be passed out of the function, and may be used for callback into the context that was active at the time of the binding. As usual, if the context is no longer active, a call on the pointer has undefined results.

While closures are extremely useful, and can actually be used, in conjunction with garbage collection, to implement object oriented programming (as has been done in CLOS), there may be objections that this extension is beyond the scope of the current Standardisation effort, and perhaps beyond the scope of C++. There is existing practice: GNU implements closures in C, for example. I do not believe that closures of nested function create any major problems on any architectures: the trampoline technique can be used even on machines with separate instruction and data banks (by accepting some maximum number of active closures).

There are four options

1) Specify that syntax is conforming and represents formation of closures
2) Specify that the syntax is non-conforming
3) Specify that the syntax must be implementation defined
4) Specify that the syntax is not defined

Option (3) requires a conforming processor to compile code which uses the syntax, however the results of executing this code are implementation defined. Certain restrictions could be placed on this, for example, the implementors choices may be restricted to:

a) form a closure
b) throw an exception
c) return a 0

Although it may appear unwise to allow an implementation to always return a 0, for example, in fact even when closures are formed using the trampoline method, it is possible that memory will be exhausted, and so a conforming program should handle the error conditions anyhow.

Case (4) allows that any program with this syntax may cause a diagnostic to be issued by the processor (usually at compile time), such a program cannot be strictly conforming, but it may be conforming.

We believe that if the committee rejects mandating closures, that the ability of vendors to provide this facility as an extension should not be precluded by the Standard.

2. Advantages and Uses

Nested function have several important uses. While it cannot be argued that they introduce a major new programming paradigm, at least without garbage collected closures, they offer significant advantages and I believe the advantages far outweigh the costs.

2.1 Nested functions are Natural

C already supports block structure for unnamed blocks. C++, particularly with the advent of namespaces, already contains the machinery for compilers to implement nested name lookup, for programmers to understand its use in a way consistent with the rest of the language, and for the committee to write the appropriate words into the Working Paper.

Many languages, modern and not so modern, allow lexical nesting of functions, in particular, the semantics are well understood by computer scientists, most programmers, and even many students. In particular, Algol, PL/1, Pascal, and Modula support nested functions.

Nested functions are so well understood many CISC microprocessors have dedicated instructions, registers, and mechnisms to support their implementation.

One can almost view the proposal to introduce nested functions as the removal of an unnecessary restriction, rather than a complete language extension.

2.2 Rapid acceptance by programmers

Many programmers are used to using nested functions and functional decomposition. Providing this facility will enhance their ability to become productive in C++ quickly. I am especially concerned that programmers familiar with nested function technique will use static variables to provide context if nested functions are not available, destroying re-entrancy.

2.3 Rapid Prototyping

C++ is often criticised for being hard to prototype rapidly in. Nested function will aid this by allowing functional decomposition without the programmer overheads of declaring and passing long lists of parameters.

2.4 Helper functions

When an ordinary external function is being coded, it is sometimes useful to declare a helper function at file scope (with internal linkage), such a function is called a helper function, and is not accessible outside the scope of the file in which the implementation of the main function is being coded.

Unfortunately, the equivalent helper functions for members cannot always be made file local, since they will not have access to the private details of the objects implemetation, yet their principal purpose is to aid in decomposition of operations on implementation details.

However, declaring such helpers in the class interface is contrary to the principle of information hiding and is not not required for technical reasons either. It has the effect not only of cluttering the clas interface with irrelevant implementation details, but, as a side effect, causing unnecessary recompilations, sometimes of millions of lines of code.

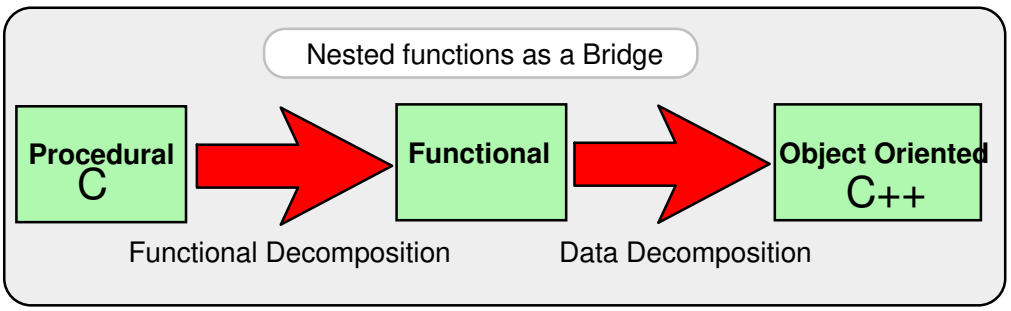
Nested function do not provide a complete solution to this problem, since they can not be shared by several member functions, however they do help decompose a single member function.

2.5 Functional Decomposition: a bridge to Object Oriented programming

Nested functions facilitate functional decomposition without the need for simultaneous data decomposition. They provide a natural bride to go from a large procedure first by functional decomposition into nested functions, and then by creation of a class.

We dont believe I can overemphasise the importance of this bridge as a program development techique, since it allows binding of functions to objects to be defered until part of the interface of the class to be is already existant.

Conversion of a large function directly into classes is very difficult and time consuming: I recently wrote a parser function in several days that became too big to manage: it took over a week to rewrite the function in terms of classes and get it running again.



2.6 Efficiency

We have left the best to last. Nested functions allow highly efficient code to be generated. There are several reasons for this.

2.6.1 Machine Architecture

Many machine architectures were specifically design to include facilities to manage nested functions. These include the Intel 80x68 (above 186) family and the 68000 family, which have ENTER and LEAVE instructions specifically designed for this purpose, and registers dedicated to stack frame and display maintenance and access.

Even on RISC machines, the overheads required to support this functionality are minimal.

2.6.2 Savings on parameter passing

Access to data via the display of local scope is much more efficient when code is generated by the compiler to provide that access than when the user passes one or more arguments. While the user can reduce the number of arguments to a function by packaging several arguments in an object, such packaging is time consuming, error prone, and hard to undo: it often represents premature binding.

At best, emulating nested function by passing only a pointer to a struct cannot be as efficient as compiler generated code to do the same thing automatically.

2.6.3 Global optimisation

Nested functions can usually be inlined: they can certainly be inlined after the body of the function has been seen, even on a one pass compiler. However nested functions also allow global optimisation is a way that separately compiled and separately accessible functions cannot, since every use of such functions is visible.

2.7 Localised Cost

Usually, it is necessary to argue that an extension or change to a language meets the criteria of localised cost: the penalty associated with the use of a feature only affects users of the feature, and is prefered to be localised to the specific times and places that the feature is used.

We are happy to argue that the cost of nested function use will generally be negative, that is, it offers possibilities of performance gains rather than any costs.

A completely dumb implementation for the 80486 processor, for example, would incur a cost of one word on the stack for each function, plus the extra time to setup this word. This is because the ENTER instruction with an argument of 1 is required if a function at the outer most nesting level contains a nested function, whereas an ENTER with an argument of 0 can be used if it is known that the function contains no nested functions. A slightly smarter compiler could assume 0 until a nested function is encountered and backpatch offsets to the stack frame if this had to be adjusted to 1.

Thus, it is not hard to ensure that the generated code for global functions not containing nested functions is identical to that which would have been used if nested functions were not added to the language. On the other hand, when nested functions are used, an actual gain in efficiency will almost invariably result, even for the dumbest of compilers.

A smart compiler may be able to realise quite significant savings, using inlining or global optimisation techniques.

Appendix 1: A GNU C routine with nested functions

```
/* example of nested function use*/
/**/
/* infix calculator*/

/* class SyntaxError{};*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

volatile void syntax_error(int i) { printf("Syntax error %d\n", i); exit(1); }

float calculate(const char *s) /* throw(SyntaxError)*/
{

  enum token_t {lit, plus, minus, mul, div, left, right, eol} token;
  float value;


  /* tokenisation ------------------------------*/

  void get_token()
  {
    void skip_spaces() {while(*s==' ')++s;}

    char lexeme[20];

    void get_lexeme()
    {
      char *t=lexeme;
      skip_spaces();
      while(*s && *s !=' ')*t++=*s++;
      *t=0;
    }


    void tokenise()
    {
      float eval_lit()
      {
        float x;
        sscanf(lexeme,"%f",&x);
        return x;
      }

      switch(lexeme[0])
      {
        case '+'  : token = plus;   break;
        case '-'  : token = minus;  break;
        case '*'  : token = mul;    break;
        case '/'  : token = div;    break;
        case '('  : token = left;   break;
        case ')'  : token = right;  break;
        case '\x0': token = eol;    break;
        case '0'  :
        case '1'  :
        case '2'  :
        case '3'  :
        case '4'  :
        case '5'  :
        case '6'  :
        case '7'  :
        case '8'  :
        case '9'  : token = lit; value=eval_lit(); break;
        default: /* throw(SyntaxError);*/
       syntax_error(1);
      }
    }

    get_lexeme();
    tokenise();
```

```
}

/* parsing ------------------------------*/

auto float eval_expr(); /* forward reference*/

float eval_nestexpr()
{

  float val;
  switch(token)
  {
    case left:
    {
      get_token();
      val=eval_expr();
      if(token != right) /* throw(SyntaxError);*/
         syntax_error(2);
      get_token();
    }
    break;

    case lit:
    {
      val=value;
      get_token();
    }
    break;

    default: /* throw(SyntaxError);*/
    syntax_error(3);
  }
  return val;
}

float eval_preop()
{
  switch(token)
  {
    case plus:
    {
      get_token();
      return eval_nestexpr();
    }

    case minus:
    {
      get_token();
      return - eval_nestexpr();
    }
    default:
     return eval_nestexpr();
  }
}

float eval_mulop()
{
  float arg1=eval_preop();
  switch(token)
  {
    case mul: get_token(); return arg1 * eval_mulop();
    case div: get_token(); return arg1 / eval_mulop();
    default: return arg1;
  }
}

float eval_addop()
{
  float arg1=eval_mulop();
  switch(token)
  {
    case plus:  get_token(); return arg1 + eval_addop();
    case minus: get_token(); return arg1 - eval_addop();
    default: return arg1;
  }
```

```
  }

  float eval_expr()
  {
    get_token();
    return eval_addop();
  }

  return eval_expr();
}

int main(int argc, char **argv)
{
  if (argc > 1)
    printf("%f\n", calculate(argv[1]));
  else
    printf("usage: %s expression\n", argv[0]);
  return 0;
}
```

Appendix 2: A sample of generated code for nested functions

```
; demonstration of assembler code for nested functions 80286 processor
;--------------------------------------------------------------------
;
; the assembler equivalent to the following C routine
; is given below
;
; this routine is callable by Borland C++ 3.1 passing
; parameters on the stack using the standard stack frame
; and returning int results in the ax register
;
; The assembler was tested by John Skaller
; The C was tested by Fergus Henderson
;

        .386p
;
; int func0(int a)
; {
;   int b=1;
;
;   int func1(int c)
;   {
;     ++b;
;     int d=a+b+c;
;
;     int func2(int e);
;     {
;       int f=a+b+c+d+e;
;       return f;
;     }
;
;     d+=func2(1);
;     int y=func2(2);
;     if(b<4) y+=func1(y);
;
;     return d+y;
;   }
;
;   int z=func1(0);
;   return z+1;
; }
;

NEST_TEXT      segment byte public use16 'CODE'
        assume  cs:NEST_TEXT

; display layout
; --------------
;
; callers frame 0   ; callers stack frame
dsp0    equ     -2  ; outermost scope
dsp1    equ     -4
dsp2    equ     -6

_func0  proc      far
        public  _func0              ; extern "C" name
a       equ     6
; return addr   4
; callers frame 0
; display me    -2
b       equ     -4
z       equ     -6

        enter   4,1                 ; level 1

        mov     b[bp],word ptr 1    ; b=1

        push    0                   ; z=func1(0)
        call    near ptr func1
        mov     z[bp],ax

        mov     ax,z[bp]            ; return z+1
```

```
        inc     ax
        leave
        ret

func1   proc    near
c       equ     4
; return addr   2
; callers frame 0
; func0 display -2
; display me    -4
d       equ     -6
y       equ     -8

        enter 4,2                   ; level 2 nested function

        mov   bx,dsp0[bp]           ; func0 activation record

        mov   ax,ss:b[bx]           ; ++b
        inc   ax
        mov   ss:b[bx],ax           ; ax holds b

        add   ax,ss:a[bx]           ; d=a+b+c
        add   ax,c[bp]
        mov   d[bp],ax

        push  1                     ; d+=func2(1)
        call  near ptr func2
        add   d[bp],ax

        push  2                     ; y = func2(2)
        call  near ptr func2
        mov   y[bp],ax
                                    ; if (b<4)
        mov   bx,dsp0[bp]           ; func0 activation record
        cmp   ss:b[bx],word ptr 4
        jae   lab1

        push  ax                    ; y+=func1(y)
        call  func1                 ; ax holds y
        add   y[bp],ax
lab1:
        mov   ax,d[bp]              ; return d+y
        add   ax,y[bp]
        leave
        ret   2

func2   proc  near
e       equ   4
; return addr 2
; callers frame 0
; func0 frame is -2
; func1 frame is -4
; display me is -6

        enter 0,3                   ; level 3 nested function

                                    ; return a+b+c+d+e;

        mov   bx,dsp0[bp]           ; frame of func0
        mov   ax,ss:a[bx]           ; func0::a
        add   ax,ss:b[bx]           ; func0::b

        mov   bx,dsp1[bp]           ; frame of func1
        add   ax,ss:c[bx]           ; func1::c
        add   ax,ss:d[bx]           ; func1::d

        add   ax,e[bp]
        leave
        ret   2

func2   endp
func1   endp
_func0  endp
NEST_TEXT ends
        end
```