EX 3.2 Because of no recursion, Fortran 77 cannot be more than one live instance of the local variables for a given subroutine. However, Algol need a stack to accommodate multiple copies. The ***sum*** on the code below need to know the value of ***low*** for the recursive call. Therefore it is not correct for statically allocated local variables.

```
function sum(f, high, low)
        if high = low
                return f(high)
        else
                return f(hight) - sum(f, high - 1, low)
```

Because Algol has limited extent for local variables, the values of variables are lose when control leaves scope in which they are declared. Lisp needs allocation in the heap to accommodate unlimited extend. The code below doesn't work correct for stack allocated variables. It returns a function which add to its argument f the value m originally passed to add_n. The value m remain accessible when the function returned by add_n remain accessible.

```
function add_n(m)

        return { function(f) return m + f }
```

EX 3.4 (1) For C, a static variable declared inside a function could live but no in scope if the execution is not inside the function.

(2) For C++, non-public fields of an object of class C could live but not in scope if execution is not inside a method of C.

(3) For Modula-2, a global variable declared in a module could live but not in scope if execution is not inside the module.

EX 3.5 (1) C
3, 1 inner
3, 1 middle
1, 2 main

(2) C#
3, 1 inner
3, 1 middle
3, 1 main

(3) Modula-3
1, 2 inner
1, 2 middle
1, 2 main

EX 3.7 (1) There is a reverse_list to make a new list but no track for the old list nodes when Brad assigns the return value to L. Therefore, the program has a memory problem for the main loop because the iteration on the loop run out of memory.

(2) When the delete_list reclaims old list nodes, it reclaim the widgets as well. There is revered list to contain reference for the location in the heap which may be used for new allocation data. It may corrupt to use the element in the reversed list.

EX 3.8 (1) For figure 3.6, the header file stack.h:

```
#define STACK_SIZE 100
typedef int element;

extern void push(element);
extern element pop();
```

Implementation file stack.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

typedef short stack_index;

element s[STACK_SIZE];
stack_index top = 0; /* first unused slot */
static void error(const char* err) {
        fprintf(stderr, "stack error: %s\n", err);
        exit(1);
}

void push(element elem) {
        if (top == STACK_SIZE) error("overflow");
        else s[top++] = elem;
}

element pop() {
        if (top == 0) error("underflow");
        else return s[--top];
}
```

Client file:

```
#include <stdio.h>
#include "stack.h"

int main() {
        element x, y;

        x = 3;
        push(x);
```

```
        y = pop();
        printf("y == %d\n", y);
}
```

(2) For figure 3.8, Header file stack_manager.h:

```
#define STACK_SIZE 100
typedef int element;

typedef short stack_index;

typedef struct {
        element s[STACK_SIZE];
        stack_index top;
} stack;

extern void init_stack(stack*);
extern void push(stack*, element);
extern element pop(stack*);
```

Implementation file stack_manager.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "stack_manager.h"

static void error(const char* err) {
        fprintf(stderr, "stack error: %s\n", err);
        exit(1);
}

extern void init_stack(stack* stk) {
        stk->top = 0;
}

extern void push(stack* stk, element elem) {
        if (stk->top == STACK_SIZE) error("overflow");
        else stk->s[stk->top++] = elem;
}

extern element pop(stack* stk) {
        if (stk->top == 0) error("underflow");
        else return stk->s[--stk->top];
}
```

Client file:

```
#include <stdio.h>
#include "stack_manager.h"
```

```
int main() {
        element x, y;

        stack A, B;

        init_stack(&A);
        init_stack(&B);

        x = 3;
        push(&A, x);
        push(&B, pop(&A));

        y = pop(&B);
        printf("y == %d\n", y);
}
```

EX 3.14 For static scoping, it is 1 1 2 2. For dynamic scoping, it is 1 1 2 1. What make it different is about *set_x* sees the global x or the x declared in *second* when it is called by *second*.

EX 3.18 For shallow binding, *set_x* and *print_x* always access foo's local x. It prints 1 0 2 0 3 0 4 0. For deep binding, if n is even, *set_x* accesses the global x. If n is odd, *set_x* accesses foo's local x. That is, print_x accesses the global x when n is 3 or 4 and foo's local x when n is 1 or 2. It prints 1 0 5 2 0 0 4 4.