Section 12

The C/C++ Run Time Environment; Recursion

NOTE 12.1

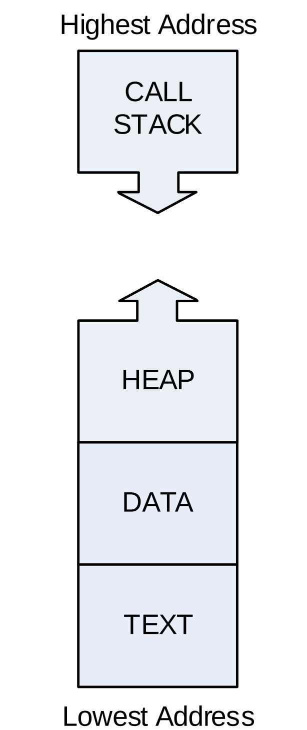
Program Areas

When a program is run the operating system typically allocates a block of memory dedicated to that program. This block, known as the run time environment, is usually divided into four areas known as the *text*, *data*, *heap*, and *call stack* (or simply, *stack*) areas, as shown in the illustration. Whether or not all four areas are needed and their actual order in memory is dependent upon both the program itself and the implementation. The following is a brief description of each area:

Text Area

Contains:
- Executable instructions (run time operations);
- Immediate data (data included as part of machine instructions);
- String literals (if they're not in the data area).

Properties:
- Considered "read only" – cannot be modified by the program (prevents self modifying code);
- Size depends upon number and type of program instructions.



Typical Run-Time Program Areas

Data Area

Contains:
- Load time (non-automatic) variables;
- String literals (if they're not in the text area).

Properties:
- Considered "read/write" – can be modified by the program;
- Size depends upon number and type of non-automatic variables;
- Divided into two parts:
    1. Initialized – non-automatic variables with non-0 initializers;
    2. Uninitialized – non-automatic variables without initializers or with initializers of 0.
        A. Also called the BSS (Block Started by Symbol) area.

The Heap

Contains:
- Dynamically-allocated non-stack storage (*malloc*, *calloc*, *realloc*, *new*, *new[]*).

Properties:
- Accessed only using pointers;
    1. May be used for any data types (including arrays and structures).
- Managed by a "heap manager" program;
- Size determined by programmer;
- Depending upon the implementation, the maximum size may be:
    1. Fixed;
    2. Determined by the current size of the stack;
    3. Virtually unlimited.

The Stack

Contains:
- Automatic variables (including formal parameters);
- Values returned by functions;
- Function housekeeping information;
- Temporary compiler created variables.

Properties:
- Size changes dynamically as a program runs;
- Depending upon the implementation, the maximum size may be:
    1. Fixed;
    2. Determined by the current size of the heap;
    3. Virtually unlimited.

NOTE 12.2

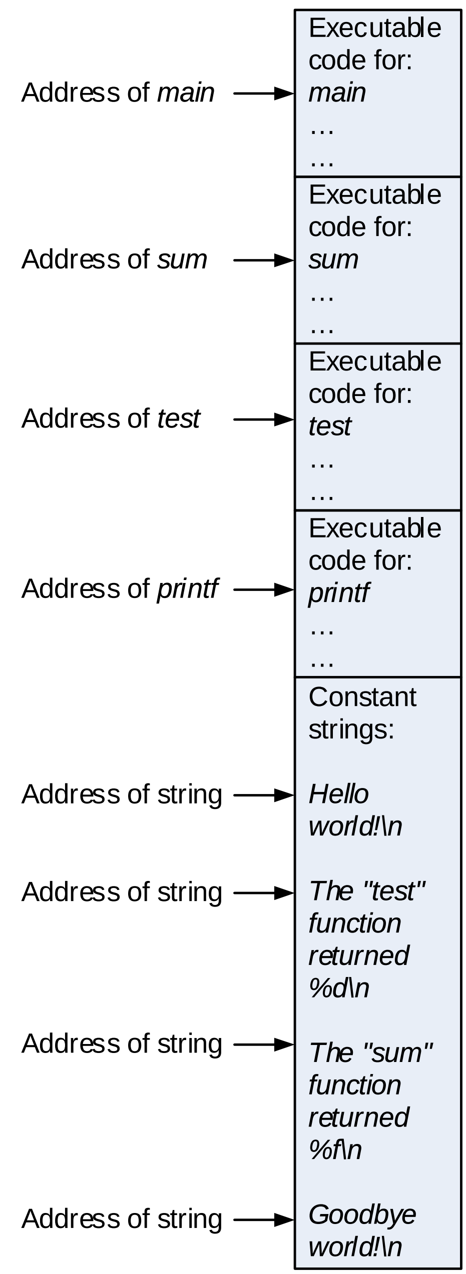
The Text Area

The text area is where a program's executable code resides. This code consists of machine instructions to perform both load time initializations and the various run time computations and assignments contained in the program's functions. Some implementations also place string literals here to prevent modification. Anything residing in the text area is considered "read-only" and the result of an attempted write varies between implementations. Because it is read only, multiple instances of the same program can share it.

Accessing the Text Area

Usually a programmer does not explicitly access the text area but, rather, merely lets the program automatically fetch and execute instructions as needed. Using pointers to functions, however, a programmer can explicitly access addresses in the text area for the purpose of indirectly calling functions located at those addresses:

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

double sum(double x, double y)
{
    return(x + y);
}

int test(int cat, int dog)
{
    return(cat < dog ? cat : dog);
}

int main(void)
{
    char * const SAMPLE_STRING = "Hello world!\n";
    int speed;
    int (*ifnp)(int a, int b) = test;
    double time;
    double (*dfnp)(double a, double b) = sum;

    speed = ifnp(5, 25);
    printf("The \"test\" function returned %d\n", speed);
    time = dfnp(6.5, 12.3);
    printf("The \"sum\" function returned %f\n", time);
    printf(SAMPLE_STRING);
    strcpy((char *)test, "Goodbye world!\n");      /* whoops! */
    return(EXIT_SUCCESS);
}
```



Typical Program Text Area

Modifying the Text Area

The text area must be considered read-only even if a particular implementation does not implement it that way. At best, attempted modification of the text area will result in a non portable program. At worst it will result in erroneous program operation with no indication from the system. In between can lie an assortment of compiler and operating system warnings and/or errors/crashes. In the illustration above the *strcpy* function attempts such a modification.

NOTE 12.3

The Data Area

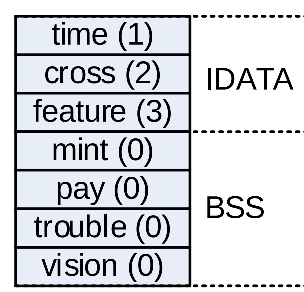
The data area stores all non-automatic variables and any string literals that are not stored in the text area. Variables stored in this area, unlike automatic variables, exist for the life of the program and are initialized only once when the program is loaded into memory prior to running. Since the size and number of all non-automatic variables and string literals do not change during a program run, the size of the data area is usually fixed. The data area is typically divided into two separate areas frequently called the “initialized data” area and the BSS area.

The Initialized Data Area
The initialized data area contains all non-automatic variables having non-0 initializers as well as any string literals not stored in the text area. The initialization values are stored in the executable program file along with the program’s executable code and, thus, increase the size of the file.

The BSS Area
The BSS area contains all non-automatic variables having initializers of 0 as well as all non-automatics without initializers at all (which are still guaranteed to have initial values of 0). Because of this, the only information that must normally be stored in an executable program file pertaining to the BSS area is its size. When a program first starts it automatically does a very efficient “zero-fill” operation on the BSS area. The executable code to do this resides in the text area.

Data Area Example Variables
The following illustrates the data areas used by some typical variables. Note that whenever initializers are present for automatic variables, the executable code to perform those initializations resides in the text area.

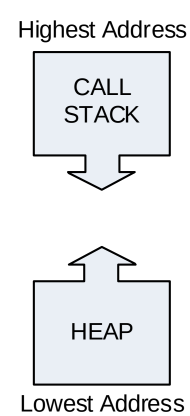
```
double mint;                          // BSS:  initial value of 0
double pay = 0;                       // BSS:  initial value of 0
double time = 1;                      // Initialized Data:  initial value of 1
static double trouble;                // BSS:  initial value of 0
static double cross = 2;              // Initialized Data:  initial value of 2

void DemonstrateMemoryAreas()
{
    double dealing;                   // STACK:  initial value of garbage
    double duty = 0;                  // STACK:  initial value of 0
    static double vision;             // BSS:  initial value of 0
    static double feature = 3;        // Initialized Data:  initial value of 3

    cout << "Hello world!\n";         // String literal:  Initialized Data or Text
}
```



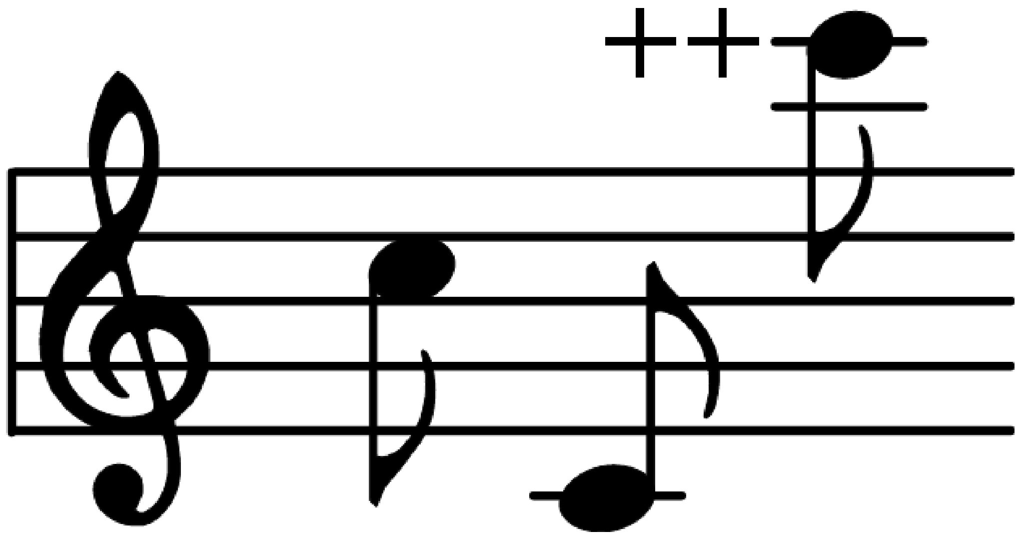
Data Area

The Heap

The heap is used for any dynamically allocated memory needed by a progam. These allocations are requested by *malloc*, *calloc*, *realloc*, *new*, *new*[], etc., which produce pointers to the allocated memory. A “heap manager” program controls heap resources for these functions. The heap is separate from the stack, whose allocations are not under the programmer’s control. On some implementations the heap and the stack share a common block of memory with the heap starting at the lowest memory address and growing upward and the stack starting at the highest memory address and growing downward. In such implementations the maximum size of the heap may be dependent upon the current size of the stack and vice versa. The heap is always accessed using the pointers returned by the various allocation functions.



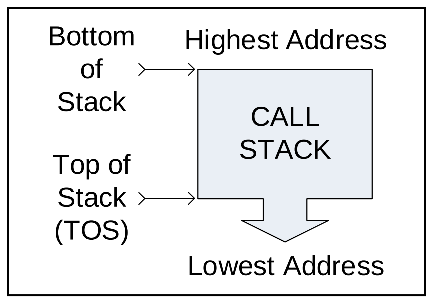
Typical Stack & Heap Program Areas



NOTE 12.4A

The Call Stack

The call stack (or simply, stack) is the memory region typically used to support function operation. It can contain a function's return object, its automatic variables including its formal parameters, and miscellaneous housekeeping information such as return addresses and temporary variables created by the compiler to facilitate expression evaluation. It is a "last-in/first-out" (LIFO) data structure that may be compared to a spring-loaded stack of cafeteria trays. Data is placed onto the top of the stack by what is referred to as a "push" operation and removed from the top by a "pop" operation. The address of the top of the stack is the lowest address occupied by objects currently on the stack and each successive push occurs at a lower address. As a result the top of the stack grows downwards in memory.



Stack Related Concepts

- Push and Pop;
- Stack Pointer (SP);
- C and Pascal Calling Conventions;
- Stack Frame;
- Function Return Address – Program Counter (PC);
- Frame Base Address – Base Pointer (BP) – Previous Frame Address.

Stack Operation During Function Call

- Reserve space for return object;
- Push arguments;
- Push Function Return Address;
- Push Base Pointer, then copy Stack Pointer into Base Pointer;
- Reserve space for local automatic variables.

Stack Operation During Function Return (in whatever order is appropriate)

- Load Base Pointer with Previous Frame Address (BP = *BP);
- Load Program Counter with Function Return Address;
- Pop stack frame.

...............CONTINUED

NOTE 12.4B **...............CONTINUATION**

Explanation Of Stack Related Concepts:

Stack Frame
Every time a function is called, stack space is allocated for the object it returns, its automatic variables including its formal parameters, and miscellaneous housekeeping information. This is collectively known as the function's “stack frame”. When the function returns, its stack frame is popped off the stack. If a function is called recursively each call creates its own unique stack frame. To speed things up programs sometimes pass parameters and return values in internal machine registers instead of on the stack.

Stack Pointer
An internal machine register known as the “Stack Pointer” (SP) keeps track of the current Top Of Stack (TOS), which is the lowest address currently used for the stack. Whenever something must be pushed onto the stack, the Stack Pointer is first decremented by the number of bytes needed to store it and then it is stored at the resulting address. As items are popped off the stack, the Stack Pointer is incremented accordingly.

C and Pascal Calling Conventions
When a function is called, all of its arguments are typically pushed onto the stack. Arguments may be pushed in either right-to-left order, known as the “C Calling Convention”, or in left-to-right order, known as the “Pascal Calling Convention”. Functions that can accept a variable number of arguments require the use of the C calling convention.

Program Counter, Function Return Address
An internal machine register known as the “Program Counter” (PC) keeps track of the address of the next program instruction to be executed. Most of the time this register is merely loaded with the address of next instruction in memory, but in some cases this scenario is altered, such as with a conditional statement, a loop, or a function call. Whenever a function is called, the Program Counter is automatically reloaded with the starting address of that function. When a function returns, however, program execution must resume with the instruction that follows the original function call. To accomplish this, the address of that instruction, called the *function return address*, is pushed onto the stack and used to reload the Program Counter upon function return.

Frame Address, Base Pointer, Previous Frame Address
After the *function return address* has been pushed onto the stack, a pointer to a reference point in the previous stack frame, known as the *previous frame address*, is pushed and the address where it is pushed, known as the *frame base address*, is stored in an internal machine register known as the Base Pointer (BP). The Base Pointer provides a simple way of referencing any object in the current stack frame using a simple offset rather than an absolute memory adderess. The function’s return object, formal parameters, and *function return address* are typically stored at positive offsets from the *frame base address* the while its local automatic variables are stored at negative offsets. The *previous frame address* is the *frame base address* of the previous stack frame and provides a link back to that frame so that when the function returns, the Base Pointer can be reloaded with this address, thereby causing the objects within that frame to become current.

Stack Events During Function Return
When a function returns, the Base Pointer is reloaded with the *frame base address* of the previous frame, the Program Counter is reloaded with the address of the instruction in the previous function where program execution will resume upon the return, and the Stack Pointer is reloaded with the address of the last item in the previous stack frame, thereby popping the entire current stack frame. These events occur in whatever order is appropriate to ensure correct operation.

...............CONTINUED

NOTE 12.4C **...............CONTINUATION**

The following code is used to illustrate some typical stack manipulations that occur as functions are called and return. 2-byte ints, 4-byte pointers, 8-byte doubles are assumed and the starting address of the stack is arbitrary. Diagrams of the resulting stack frame sequence are shown on the following page. The procedures used for manipulating the stack are:

<u>To create a new stack frame:</u>

1. Reserve space for return object, if any;
2. Push arguments, if any;
3. Push Function Return Address;
4. Push the Previous Frame Address:
    a. That is, push BP, then update it by doing *BP = SP*.
5. Reserve space for local automatic variables, if any:
    a. Initialization, if needed, is typically done by the called function itself.

<u>To "push" any object:</u>

1. Determine the number of bytes in the object;
2. Decrement the Stack Pointer by that number;
3. Store the object at the Stack Pointer address.

<u>To return from a function (done in whatever order is appropriate):</u>

1. Point Base Pointer at previous frame by doing *BP = *BP*;
2. Load Function Return Address into Program Counter;
3. Pop Stack Frame (reload Stack Pointer).

```
void fcnB(int cat, int dog)
{
    double wB = 5.6, zB;
}

void fcnA(double pig, double horse)
{
    int xA, yA = 81;

    fcnB(−7, 29);
    xA = 10;                // assume the instruction to do  xA = 10  is at address 0x21F8
}

int main()
{
    fcnA(7.5, 2.8);         // assume that when fcnA is called, both SP and BP == 0x0FB8
    return(0);              // assume the instruction to do return(0) is at address 0x3AC2
}
```

...............CONTINUED

NOTE 12.4D ....**CONTINUATION**

Stack Frame Sequence Resulting
From Previous Program

Figure 1

Stack View prior to *fcnA* call

Pointer	Memory Addresses: Absolute	Stack Values	Description	Frame
	??	??	??	*startup* Stack Frame
	FC2h	??	??	
	FC0h	??	Return Object (int)	*main* Stack Frame
	FBCh	??	Function Return Address	
BP FB8h, SP FB8h	FB8h	??	Previous Frame Address	

Figure 2

Stack View *fcnA* arg push 1

Pointer	Memory Addresses: Absolute	Stack Values	Description	Frame
	??	??	??	*startup* Stack Frame
	FC2h	??	??	
	FC0h	??	Return Object (int)	*main* Stack Frame
	FBCh	??	Function Return Address	
BP FB8h	FB8h	??	Previous Frame Address	
SP FB0h	FB0h	2.8	horse	

Figure 3

Stack View *fcnA* arg push 2

Pointer	Memory Addresses: Absolute	Stack Values	Description	Frame
	??	??	??	*startup* Stack Frame
	FC2h	??	??	
	FC0h	??	Return Object (int)	*main* Stack Frame
	FBCh	??	Function Return Address	
BP FB8h	FB8h	??	Previous Frame Address	
	FB0h	2.8	horse	
SP FA8h	FA8h	7.5	pig	

Figure 4

Stack View *fcnA* FRA push

Pointer	Memory Addresses: Absolute	Stack Values	Description	Frame
	??	??	??	*startup* Stack Frame
	FC2h	??	??	
	FC0h	??	Return Object (int)	*main* Stack Frame
	FBCh	??	Function Return Address	
BP FB8h	FB8h	??	Previous Frame Address	
	FB0h	2.8	horse	
	FA8h	7.5	pig	
SP FA4h	FA4h	3AC2h	Function Return Address	

Figure 5

Stack View *fcnA* PFA push w/ BP update

Pointer	Memory Addresses: Absolute	Stack Values	Description	Frame
	??	??	??	*startup* Stack Frame
	FC2h	??	??	
	FC0h	??	Return Object (int)	*main* Stack Frame
	FBCh	??	Function Return Address	
	FB8h	??	Previous Frame Address	
	FB0h	2.8	horse	
	FA8h	7.5	pig	
	FA4h	3AC2h	Function Return Address	
BP FA0h, SP FA0h	FA0h	FB8h	Previous Frame Address	

Figure 6

Stack View *fcnA* allocation of all local variables & initialization of 1 local variable

Pointer	Memory Addresses: Absolute	Stack Values	Description	Frame
	??	??	??	*startup* Stack Frame
	FC2h	??	??	
	FC0h	??	Return Object (int)	*main* Stack Frame
	FBCh	??	Function Return Address	
	FB8h	??	Previous Frame Address	
	FB0h	2.8	horse	*fcnA* Stack Frame
	FA8h	7.5	pig	
	FA4h	3AC2h	Function Return Address	
BP FA0h	FA0h	FB8h	Previous Frame Address	
	F9Eh	??	xA	
SP F9Ch	F9Ch	81	yA	

Figure 7

Stack View *fcnB* arg push 1

Pointer	Memory Addresses: Absolute	Stack Values	Description	Frame
	??	??	??	*startup* Stack Frame
	FC2h	??	??	
	FC0h	??	Return Object (int)	*main* Stack Frame
	FBCh	??	Function Return Address	
	FB8h	??	Previous Frame Address	
	FB0h	2.8	horse	*fcnA* Stack Frame
	FA8h	7.5	pig	
	FA4h	3AC2h	Function Return Address	
BP FA0h	FA0h	FB8h	Previous Frame Address	
	F9Eh	??	xA	
	F9Ch	81	yA	
SP F9Ah	F9Ah	29	dog	

■ ■ ■

Figure 8

Stack View *fcnB* complete

With Relative Addresses Shown

Pointer	Memory Addresses: Relative	Memory Addresses: Absolute	Stack Values	Description	Frame
	BP+??	??	??	??	*startup* Stack Frame
	BP+??	FC2h	??	??	
	BP+8h	FC0h	??	Return Object (int)	*main* Stack Frame
	BP+4h	FBCh	??	Function Return Address	
	BP	FB8h	??	Previous Frame Address	
	BP+10h	FB0h	2.8	horse	*fcnA* Stack Frame
	BP+8h	FA8h	7.5	pig	
	BP+4h	FA4h	3AC2h	Function Return Address	
	BP	FA0h	FB8h	Previous Frame Address	
	BP-2h	F9Eh	??	xA	
	BP-4h	F9Ch	81	yA	
	BP+Ah	F9Ah	29	dog	*fcnB* Stack Frame
	BP+8h	F98h	-7	cat	
	BP+4h	F94h	21F8h	Function Return Address	
BP F90h	BP	F90h	FA0h	Previous Frame Address	
	BP-8h	F88h	5.6	wB	
SP F80h	BP-10h	F80h	??	zB	

NOTE 12.5A

Automatic Variable Considerations

Automatic Variables in Inner Blocks
Many functions contain block statements other than the function body itself in which automatic variables are declared, such as in the **if** statement in the following example. Although it is possible that such a block might never be entered and, therefore, its automatic variables never needed, most compilers in the interest of reduced complexity allocate all automatics contained in the entire function as part of the initial stack frame. The quantity and types of these variables dictate the amount of space allocated. This permits the programmer to make a reasonable estimate of the amount of stack space that a function will require. For example (assuming 1 byte **char**s, 2 byte **int**s, and 8 byte **double**s):

```
struct Information {int bird; double pay; char rank[80];};

int DemoStackUsage(int z)       // Stack requirements:  Automatic variables  ~912 bytes;  Return object ~2 bytes
{
        int x, y;                       // requires 4 stack bytes
        double a, b;                    // requires 16 stack bytes
        static double big[75000];       // requires 0 stack bytes (statics do not reside in the stack)

        if (SomeFunction())             // enter block only if SomeFunction returns non-0
        {
                double art[100];        // requires 800 stack bytes
                Information employee;   // requires ~90 stack bytes
        }
        return int(double(x + y) + a + b);
}
```

Automatic Variables and the Maximum Stack Size
On some systems the maximum size of the stack is fixed and the program will typically be terminated with a "Run-Time Stack Overflow" error if that size is exceeded during a program run. This potential problem cannot always be detected at compile time. Other more sophisticated systems simply store the contents of the stack on disk if it expands beyond a certain point, thereby permitting this "virtual" stack to expand indefinitely (until the disk gets full). Because large automatic arrays, structures, classes, and unions can require a significant amount of stack space, declaring them **static** or **const** is one technique sometimes used to prevent stack overflow problems. The downside of doing this is that the memory they occupy is not available for any other purpose, whereas stack memory is reusable.

Automatic Variable Initialization
Of the various items allocated in a stack frame when a function is called, the arguments, function return address, and previous stack frame address have known values and are pushed one at a time. However, the function being called is typically responsible for initializing its own local automatics. As a result, the Stack Pointer, which always points to the top of the stack and is equal to the value of the Base Pointer when it is time to allocate the local automatics, is simply decremented by the amount of storage needed for them. Because of this any values that happen to already be in this part of storage are not disturbed. For automatic variables that do have initializers, initialization occurs after the called function starts running, utilizing compiler generated run-time assignment instructions just as if the programmer had written separate assignment statements as part of the code in that function. For those not having initializers, no initialization occurs and they are left with whatever values are already in the locatations in memory allocated for them.

Automatic Variable Initialization Considerations
Although no variables should ever be initialized arbitrarily, this is especially important for automatic variables since their initialization occurs at run-time, thereby decreasing program execution speed. This slowdown is even more drastic for the run-time initialization large automatic aggregates such as arrays, structures, & classes.

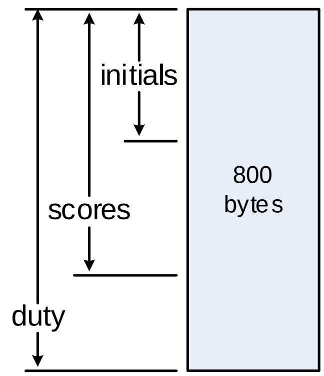
...............CONTINUED

NOTE 12.5B ....**CONTINUATION**

Automatic Variable Overlap

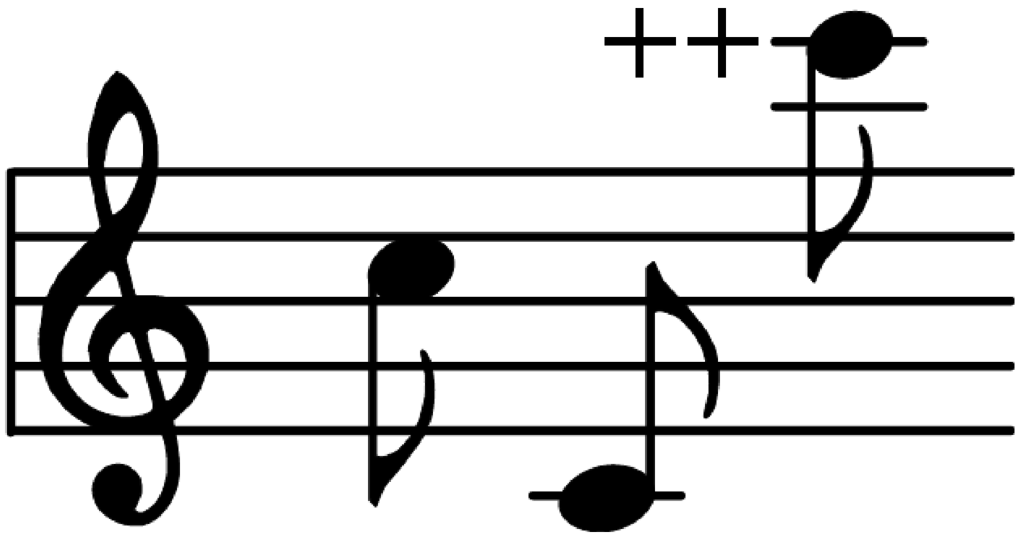
The following function contains three block statements, each of which declares an array. Assume 1 byte **char**s, 2 byte **int**s, and 8 byte **double**s:

```
void DemonstrateVariableOverlap(void)       /* ~800 bytes needed for all automatic variables */
{
    if (SomeFunction()) == 5)
    {
        double duty[100];                   /* 800 bytes required */
    }
    else
    {
        int scores[300];                    /* 600 bytes required */
    }

    while (AnotherFunction() <= 234)
    {
        char initials[300];                 /* 300 bytes required */
    }
}
```



Stack View of Automatic Variable Overlap

A programmer might at first assume that the approximate stack space required for all of the automatic variables would be 800 + 600 + 300 == 1700 bytes. While the compiler is within its rights to allocate separate locations on the stack for each of the arrays, this would be a very inefficient use of memory. Note that none of the three blocks containing the declarations are located within any of the others. Since one of the characteristics of automatic variables is that their values must be considered lost when the block in which they are declared is exited, there is no case in which more than one at a time of the three arrays must hold valid data. For this reason the same stack space can be used to represent them all and the total storage required will only be the amount needed for the largest array, approximately 800 bytes.



NOTE 12.6A

Iteration and Recursion

A function is said to be iterative if it implements its algorithm by simply looping through the code, whereas it is said to be recursive if it calls itself either directly or indirectly. Stated another way, recursion occurs whenever any function is called again before returning from a previous call. Some algorithms can be implemented either iteratively or recursively and the method chosen can affect a program's performance and maintainability. All C/C++ functions, if properly constructed, can be used recursively. In general, recursive functions should be used selectively and:

- are more compact and sometimes more easily understood than iterative functions;
- require more machine overhead and thus, run slower than iterative functions;
- can exhaust a machine's stack space;
- should use as few automatic variables as possible to minimize stack space usage;
- must not use non-automatic variables if reentrancy is required.

Both of the following functions are initially called from another function such as *main* with a positive integer passed in as an argument. Each will convert this number to individual characters representing the number, then output those characters in correct order.

```
void Iterate(int value)                          /* version 1:   Iterative version */
{
    char digit[MAX_DIGITS];                      /* array to hold each digit of value */
    int index = 0;                               /* index to starting array element */

    do                                           /* save digits in array */
        digit[index++] = value % 10 ;            /* save value of current LSD */
    while (value /= 10);                         /* form new LSD */
    while (--index >= 0)                         /* output digit values from array */
        printf("%d", digit[index]);
}

void Recur(int value)                            /* version 2:   Recursive version */
{
    static int nextValue;                        /* static - is this a good idea? */

    nextValue = value / 10;                      /* shift digits to the right (new LSD) */
    if (nextValue)                               /* if nextValue != 0 */
        Recur(nextValue);                        /* call current function again (recursively) */
    printf("%d", value % 10);                    /* output LSD of value */
}
```

...............**CONTINUED**

NOTE 12.6B **...............CONTINUATION**

The following code and corresponding stack sequence diagrams on the next page illustrate the operation of the stack, the Base Pointer, and the Stack Pointer during the execution of a recursive program. For simplicity stack frames for *printf* and any of its possible "helper functions" are not shown. 2-byte **int**s and 3-byte pointers are assumed, and arbitrary addresses are used for the "text" memory instructions and the start of the stack.

```
int main(void)        "startup function" calls main
{                         and main returns to it.
    int x;

    Ready();
    x = 3;
    printf("%d: Return from main", x);
    return(EXIT_SUCCESS);
}

void Ready(void)
{
    Recur(397);
    printf("Return from Ready");
    return;
}

void Recur(int value)
{
    static int nextValue;

    nextValue = value / 10;
    if (nextValue)
        Recur(nextValue);
    printf("%d", value % 10);
    return;
}
```

Function *main* (At Text Memory Address 1F0*h*)	
Operation	Instruction Address
Call to *Ready*	1F0*h*
x = 3	200*h*
Call to *printf*	220*h*
return	240*h*

Function *Ready* (At Text Memory Address 108*h*)	
Operation	Instruction Address
Call to *Recur*	108*h*
Call to *printf*	116*h*
return	12A*h*

Function *Recur* (At Text Memory Address 79C*h*)	
Operation	Instruction Address
nextValue = value / 10;	79C*h*
the **if** statement	7A0*h*
Call to *Recur*	7B2*h*
Call to *printf*	7BE*h*
return	7C0*h*

As always, the procedures used for manipulating the stack are:

<u>To create a new stack frame:</u>

1. Reserve space for return object, if any;
2. Push arguments, if any;
3. Push Function Return Address;
4. Push the Previous Frame Address;
    a. That is, push BP, then update it by doing *BP = SP*.
5. Reserve space for local automatic variables, if any.
    a. Initialization, if needed, is typically done by the called function itself.

<u>To "push" any object:</u>

1. Determine the number of bytes in the object;
2. Decrement the Stack Pointer by that number;
3. Store the object at the Stack Pointer address.

<u>To return from a function (done in whatever order is appropriate):</u>

1. Point Base Pointer at previous frame by doing *BP = *BP*;
2. Load Function Return Address into Program Counter;
3. Pop Stack Frame (reload Stack Pointer).

...............CONTINUED

NOTE 12.6C **...............CONTINUATION**

Stack Frame Sequence Resulting
From Previous Program

Figure 1

Stack View prior to *Ready* call

Register	Memory Addresses Absolute	Stack Values	Description	Frame
	??	??	??	*startup* Stack Frame
	FA9*h*	??	??	
	FA7*h*	??	Return Object (int)	*main* Stack Frame
	FA4*h*	??	Function Return Address	
BP FA1*h*	FA1*h*	??	Previous Frame Address	
SP F9F*h*	F9F*h*	??	x	

Figure 2

Stack View prior to *1st Recur* call

Register	Memory Addresses Absolute	Stack Values	Description	Frame
	??	??	??	*startup* Stack Frame
	FA9*h*	??	??	
	FA7*h*	??	Return Object (int)	*main* Stack Frame
	FA4*h*	??	Function Return Address	
	FA1*h*	??	Previous Frame Address	
	F9F*h*	??	x	
	F9C*h*	200*h*	Function Return Address	*Ready* Stack Frame
BP F99*h*, SP F99*h*	F99*h*	FA1*h*	Previous Frame Address	

Figure 3

Stack View prior to *2nd Recur* call

Register	Memory Addresses Absolute	Stack Values	Description	Frame
	??	??	??	*startup* Stack Frame
	FA9*h*	??	??	
	FA7*h*	??	Return Object (int)	*main* Stack Frame
	FA4*h*	??	Function Return Address	
	FA1*h*	??	Previous Frame Address	
	F9F*h*	??	x	
	F9C*h*	200*h*	Function Return Address	*Ready* Stack Frame
	F99*h*	FA1*h*	Previous Frame Address	
	F97*h*	397	value	*Recur* Stack Frame 1
	F94*h*	116*h*	Function Return Address	
BP F91*h*, SP F91*h*	F91*h*	F99*h*	Previous Frame Address	

Figure 4

Stack View prior to *3rd Recur* call

Register	Memory Addresses Absolute	Stack Values	Description	Frame
	??	??	??	*startup* Stack Frame
	FA9*h*	??	??	
	FA7*h*	??	Return Object (int)	*main* Stack Frame
	FA4*h*	??	Function Return Address	
	FA1*h*	??	Previous Frame Address	
	F9F*h*	??	x	
	F9C*h*	200*h*	Function Return Address	*Ready* Stack Frame
	F99*h*	FA1*h*	Previous Frame Address	
	F97*h*	397	value	*Recur* Stack Frame 1
	F94*h*	116*h*	Function Return Address	
	F91*h*	F99*h*	Previous Frame Address	
	F8F*h*	39	value	*Recur* Stack Frame 2
	F8C*h*	7BE*h*	Function Return Address	
BP F89*h*, SP F89*h*	F89*h*	F91*h*	Previous Frame Address	

Figure 5

Stack View -- Final State

Register	Memory Addresses Relative	Absolute	Stack Values	Description	Frame
	BP+??	??	??	??	*startup* Stack Frame
	BP+??	FA9*h*	??	??	
	BP+6*h*	FA7*h*	??	Return Object (int)	*main* Stack Frame
	BP+3*h*	FA4*h*	??	Function Return Address	
	BP	FA1*h*	??	Previous Frame Address	
	BP-2	F9F*h*	??	x	
	BP+3	F9C*h*	200*h*	Function Return Address	*Ready* Stack Frame
	BP	F99*h*	FA1*h*	Previous Frame Address	
	BP+6*h*	F97*h*	397	value	*Recur* Stack Frame 1
	BP+3*h*	F94*h*	116*h*	Function Return Address	
	BP	F91*h*	F99*h*	Previous Frame Address	
	BP+6*h*	F8F*h*	39	value	*Recur* Stack Frame 2
	BP+3*h*	F8C*h*	7BE*h*	Function Return Address	
	BP	F89*h*	F91*h*	Previous Frame Address	
	BP+6*h*	F87*h*	3	value	*Recur* Stack Frame 3
	BP+3*h*	F84*h*	7BE*h*	Function Return Address	
BP F81*h*, SP F81*h*	BP	F81*h*	F89*h*	Previous Frame Address	

NOTE 12.7A

Recursion and String Reversal

The following program inputs a line of text and outputs it backwards:

```
7  #include <iostream>
8  #include <cstdlib>
9
10 void ReverseLine()
11 {
12     int myInput;                              // new instance of this at each recursive level
13
14     myInput = std::cin.get();                 // get the next character
15     if (myInput != '\n' && myInput != EOF)    // if not at end of this line or at end of file…
16         ReverseLine();                        // …make a recursive call
17     if (myInput != EOF)
18         std::cout.put((char)myInput);         // output current character
19 }
20 int main()
21 {
22     ReverseLine();
23     return EXIT_SUCCESS;
24 }
```

-- Simplified stack frames from previous program - only local automatics are shown. --
Text input is: ***ABC\n***

0 — *main* call
No automatics on stack:

<none>	*main* stack frame

1 — 1st *ReverseLine* call
Local automatics on stack just after line 14:

<none>	*main* stack frame
'A'	*myInput* - 1st *ReverseLine* stack frame

2 — 2nd *ReverseLine* call
Local automatics on stack just after line 14:

<none>	*main* stack frame
'A'	*myInput* - 1st *ReverseLine* stack frame
'B'	*myInput* - 2nd *ReverseLine* stack frame

3 — 3rd *ReverseLine* call
Local automatics on stack just after line 14:

<none>	*main* stack frame
'A'	*myInput* - 1st *ReverseLine* stack frame
'B'	*myInput* - 2nd *ReverseLine* stack frame
'C'	*myInput* - 3rd *ReverseLine* stack frame

4 — 4th *ReverseLine* call
Local automatics on stack just after line 14:

<none>	*main* stack frame
'A'	*myInput* - 1st *ReverseLine* stack frame
'B'	*myInput* - 2nd *ReverseLine* stack frame
'C'	*myInput* - 3rd *ReverseLine* stack frame
'\n'	*myInput* - 4th *ReverseLine* stack frame

...............CONTINUED

NOTE 12.7B **...............CONTINUATION**

Recursion and String Reversal, cont'd.

The following program reads a line of text and outputs it backwards, capitalizing the third letter read. Note how the variable *recursiveLevel* is used to keep track of the current level of recursion. Since **static** variables are not kept on the stack but, rather, in data memory, there is only one instance of *recursiveLevel*, common to all levels. To save stack space local variables may be declared **static** in recursive functions whenever it is not necessary that new instances of them be created for each recursive call. However, functions containing non-automatic variables are non-reentrant.

```
#include <iostream>
#include <cstdlib>
#include <cctype>

void PrintBackwards()
{
    static int recursiveLevel;               // this variable is common to all recursive levels
    int myInput;                             // new instance of this at each recursive level

    ++recursiveLevel;                        // recursive level about to be (possibly) called
    myInput = std::cin.get();                // get the next character
    if (myInput != '\n' && myInput != EOF)   // if not at end of this line or at end of file…
        PrintBackwards();                    // …make a recursive call
    --recursiveLevel;                        // recursive level just returned from
    if (recursiveLevel == 2)                 // if this is the 3rd letter from beginning of line…
        myInput = toupper(myInput);          // …capitalize it
    if (myInput != EOF)
        std::cout.put((char)myInput);        // output current character
}
int main()
{
    PrintBackwards();
    return EXIT_SUCCESS;
}
```

NOTE 12.8

Recursion and Factorials

The factorial of a nonnegative integer *n*, written *n!*, may be defined as $(n-0) * (n-1) * (n-2) * \ldots$ for $n > 1$ and values in parentheses > 0. For example, *4!* is $4 * 3 * 2 * 1$ is 24. By definition the value of both *1!* and *0!* is 1. The following program computes factorials recursively:

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

#define MAXFACTORIAL 10L          /* maximum factorial to compute */

long ComputeFactorial(long nbr)
{
        if (nbr <= 1)
                return(1L);
        return(nbr * ComputeFactorial(nbr - 1));     /* recursive call */
}
int main(void)
{
        long number;

        for (number = 0L; number <= MAXFACTORIAL; ++number)
                printf("%ld! is %ld\n", number, ComputeFactorial(number));
        return(EXIT_SUCCESS);
}
```

ComputeFactorial stack frame

??	Return Object (long)
??	*nbr* (long)
??	Function Return Address
??	Previous Frame Address

main stack frame

??	Return Object (int)
??	Function Return Address
??	Previous Frame Address
??	*number* (long)

Recursion and the Greatest Common Divisor

The following steps implement Euclid's algorithm for computing the greatest common divisor of two positive integral values, that is, the largest positive integral value that evenly divides both:

A. Input the two values as *x* and *y*;
B. If *y* is 0 then output *x* as the answer and stop;
C. Divide *x* by *y* and let *remain* be the remainder;
D. Replace *x* with *y* and *y* with *remain* and continue at step A.

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

long gcd(long x, long y)              /* implement step A */
{
        if (y == 0)                   /* implement step B */
                return(x);            /* implement step B */
        return(gcd(y, x % y));        /* implement steps C and D */
}
int main(void)
{
        printf("gcd(128, 32777) is %ld\n", gcd(128L, 32777L));
        printf("gcd(128000, 32777000) is %ld\n", gcd(128000L, 32777000L));
        printf("gcd(43672, 92) is %ld\n", gcd(43672L, 92L));
        return(EXIT_SUCCESS);
}
```

gcd stack frame

??	Return Object (long)
??	*y* (long)
??	*x* (long)
??	Function Return Address
??	Previous Frame Address

main stack frame

??	Return Object (int)
??	Function Return Address
??	Previous Frame Address

NOTE 12.9A

Tail Recursion

A recursive function is said to be "Tail Recursive" when the recursive call is the last action to take place before the function returns. This is desirable because it only needs one stack frame no matter how many levels of recursion are required, thereby reducing overhead and preventing stack overflows resulting from deep recursion.

Non-"Tail Recursive" function *Factorial* below must always multiply the value of parameter *nbr* saved in its current stack frame by the value returned by the recursive call to itself before returning. Because of this a separate stack frame is required for each previous value of *nbr*.

```
int main(void)
{
    int result = Factorial(3);              /* Assume assignment instruction is at address 0x1289 */
    return 0;
}
int Factorial(int nbr)                      /* non-"Tail Recursive" */
{
    if (nbr <= 1)
        return(1);
    return(nbr * Factorial(nbr - 1));       /* Assume multiplication instruction is at address 0x4892 */
}
```

Stack frame illustration for **non-"Tail Recursive"** implementation of function *Factorial*
(assumes 2-byte **int** and 4-byte pointers)

EA4*h*	0*	Return Object (int)	*main*
EA0*h*	??	Function Return Address	
E9C*h*	??	Previous Frame Address	
E9A*h*	??	*result* (int)	
E9E*h*	6*	Return Object (int)	*Factorial* (Level 1)
E9C*h*	3	*nbr* (int)	
E98*h*	*1289*h	Function Return Address	
E94*h*	E9C*h*	Previous Frame Address	
E92*h*	2*	Return Object (int)	*Factorial* (Level 2)
E90*h*	2	*nbr* (int)	
E8C*h*	4892*h*	Function Return Address	
E88*h*	E94*h*	Previous Frame Address	
E86*h*	1*	Return Object (int)	*Factorial* (Level 3)
E84*h*	1	*nbr* (int)	
E80*h*	4892*h*	Function Return Address	
E7C*h*	E88*h*	Previous Frame Address	

* Value of return object is not determined until function actually returns.

..............**CONTINUED**

NOTE 12.9B **...............CONTINUATION**

Tail Recursion, Cont'd

"Tail Recursive" function *Factorial* below does not need values from previous stack frames to compute its return value and can, thus, reuse a single stack frame by simply overwriting the values of its parameters for each recursive call. Since the final return value will be computed during the final recursive call, it can return that value directly to the original caller without the need for returning back up through multiple stack frames. It is the responsibility of the compiler to recognize that a function is tail recursive and, thus, generate optimized code.

```
int main(void)
{
    int result = Factorial(3, 1);             /* Assume assignment instruction is at address 0x1289 */
    return 0;
}
int Factorial(int nbr, int a)                 /* "Tail Recursive" */
{
    if (nbr < 1)
        return 1;
    else if (nbr == 1)
        return a;
    return Factorial(nbr - 1, nbr * a);       /* Assume return instruction is at address 0x4892 */
}
```

Stack frame illustration for **"Tail Recursive"** implementation of function *Factorial*
(assumes 2-byte **int** and 4-byte pointers)

Non-Optimized

EA4*h*	0*	Return Object (int)	*main*
EA0*h*	??	Function Return Address	
E9C*h*	??	Previous Frame Address	
E9A*h*	??	*result* (int)	
E9E*h*	6*	Return Object (int)	*Factorial* (Level 1)
E9C*h*	3	*nbr* (int)	
E9A*h*	1	*a* (int)	
E96*h*	*1289*h	Function Return Address	
E92*h*	E9C*h*	Previous Frame Address	
E90*h*	6*	Return Object (int)	*Factorial* (Level 2)
E8E*h*	2	*nbr* (int)	
E8C*h*	3	*a* (int)	
E88*h*	4892*h*	Function Return Address	
E84*h*	E92*h*	Previous Frame Address	
E82*h*	6*	Return Object (int)	*Factorial* (Level 3)
E80*h*	1	*nbr* (int)	
E7E*h*	6	*a* (int)	
E7A*h*	4892*h*	Function Return Address	
E76*h*	E84*h*	Previous Frame Address	

* Value of return object is not determined until function actually returns.

Optimized

EA4*h*	0*	Return Object (int)	*main*
EA0*h*	??	Function Return Address	
E9C*h*	??	Previous Frame Address	
E9A*h*	??	*result* (int)	
E9E*h*	6*	Return Object (int)	*Factorial* (Levels 1, 2, 3)
E9C*h*	3, 2, 1	*nbr* (int)	
E9A*h*	1, 3, 6	*a* (int)	
E96*h*	*1289*h	Function Return Address	
E92*h*	E9C*h*	Previous Frame Address	

* Value of return object is not determined until function actually returns.

Section 12 Practice Exercises (not for submission or grading)

12-1. For the code shown below, replace the question marks in the list that follows to indicate the program areas where the indicated objects/instructions are kept. Be sure to differentiate between the initialized data and BSS areas. Ignore any compiler optimizations.

```
int g_test = 0, result;
char *g_title = "Goodbye world!";
static double g_value, g_height = 5;

int DemonstrateMemoryAreas(float speed, register int time)
{
        register double roll = 6;
        int flip;
        static signed char ch;
        char *cptr = "Hello world!";
        extern int g_test;

        flip = 25;
        printf("%d", g_test);
        return(flip);
}
```

g_test	??
g_test = 0	?? (Where are the instructions for *g_test = 0* kept?)
result	??
g_title	??
g_title = "Goodbye world!"	?? (Where are the instructions for *g_title = "Goodbye world"* kept?)
g_value	??
g_height	??
g_height = 5	?? (Where are the instructions for *g_height = 5* kept?)
speed	??
time	??
roll	??
roll = 6	?? (Where are the instructions for *roll = 6* kept?)
flip	??
ch	??
cptr	??
cptr = "Hello world!"	?? (Where are the instructions for *cptr = "Hello world!"* kept?)
"Hello world!"	??
flip = 25	?? (Where are the instructions for *flip = 25* kept?)
printf("%d", g_test)	?? (Where are the instructions for *printf("%d", g_test)* kept?)
return(flip)	?? (Where are the instructions for ***return**(flip)* kept?)

12-2. For the program shown below draw an illustration similar to Figure 5 in Note 12.6C, using the same *startup* stack frame shown there. However, assume the following data type sizes and make any changes required by these assumptions:

type **long** is 4 bytes;
all other stack items are 2 bytes.

<u>Do not show library function stack frames.</u> Your illustration may be drawn by hand as long as it is neat, evenly arranged, and easily readable. This is a paper exercise and should not be compared to any values obtained from actually running the program.

```
int main(void)          "startup function" calls main
{                           and main returns to it.
    int x;

    Ready();
    x = 3;
    printf("Return from main");
    return(EXIT_SUCCESS);
}

void Ready(void)
{
    long result;

    gcd(128L, 96L);
    printf("Return from Ready");
    return;
}

long gcd(long x, long y)
{
    if (y == 0)
        return(x);
    return(gcd(y, x % y));
}
```

Function *main* (At Text Memory Address 1F0*h*)	
Operation	Instruction Address
x = 3	200*h*

Function *Ready* (At Text Memory Address 108*h*)	
Operation	Instruction Address
Call to *printf*	108*h*

Function *gcd* (At Text Memory Address 79C*h*)	
Operation	Instruction Address
last **return**	7C0*h*

12-3. Assume word separators are arbitrarily defined as *whitespace* (as defined by the *isspace* function), *period*, *question mark*, *exclamation point*, *comma*, *colon*, *semicolon*, and *EOF*. Write a program that accepts input and reverses the characters in each word, capitalizing the last character of each reversed word if it happens to be a letter, then displays the result. Do not reverse any separators. For example:

```
Input:   What!  Another useless, stupid, and unnecessary program?
         Yes;  What else?:  Try input redirection.  /*.*/  /*.!?,;:=+*/
Output:  tahW!  rehtonA sselesU, diputS, dnA yrassecennU margorP?
         seY;  tahW eslE?:  yrT tupnI noitcerideR.  */./*  */.!?,;:/*+=
```

The program must work for words of arbitrary length and terminate at EOF. Do not use arrays and do not attempt to read an entire file into your program at once. You must use a recursive solution. Test your program on some arbitrary text files.

HINT: Redirection of input/output eliminates the need for explicit file opening. The programs in Notes 12.7A and 12.7B illustrate using recursion to reverse all characters on a line. The second program also shows how to keep track of which recursive level is active. Note 12.8 contains programs that illustrate returning values from lower recursive levels.