

Chapter 14 - Advanced C Topics

Outline

- 14.1 Introduction
- 14.2 Redirecting Input/Output on UNIX and DOS Systems
- 14.3 Variable-Length Argument Lists
- 14.4 Using Command-Line Arguments
- 14.5 Notes on Compiling Multiple-Source-File Programs
- 14.6 Program Termination with `exit` and `atexit`
- 14.7 The `volatile` Type Qualifier
- 14.8 Suffixes for Integer and Floating-Point Constants
- 14.9 More on Files
- 14.10 Signal Handling
- 14.11 Dynamic Memory Allocation with `calloc` and `realloc`
- 14.12 The Unconditional Branch: `goto`



Objectives

- In this chapter, you will learn:
 - To be able to redirect keyboard input to come from a file.
 - To be able to redirect screen output to be placed in a file.
 - To be able to write functions that use variable-length argument lists.
 - To be able to process command-line arguments.
 - To be able to assign specific types to numeric constants
 - To be able to use temporary files.
 - To be able to process unexpected events within a program.
 - To be able to allocate memory dynamically for arrays.
 - To be able to change the size of memory that was dynamically allocated previously.



14.1 Introduction

- Several advanced topics in this chapter
- Operating system specific
 - Usually UNIX or DOS



14.2 Redirecting Input/Output on UNIX and DOS Systems

- Standard I/O - keyboard and screen
 - Redirect input and output
- Redirect symbol(<)
 - Operating system feature, not a C feature
 - UNIX and DOS
 - \$ or % represents command line
 - Example:
`$ sum < input`
 - Rather than inputting values by hand, read them from a file
- Pipe command(|)
 - Output of one program becomes input of another
`$ random | sum`
 - Output of random goes to sum



14.2 Redirecting Input/Output on UNIX and DOS Systems

- Redirect output (>)
 - Determines where output of a program goes
 - Example:

```
$ random > out
```

 - Output goes into out (erases previous contents)
- Append output (>>)
 - Add output to end of file (preserve previous contents)
 - Example:

```
$ random >> out
```

 - Output is added onto the end of out



14.3 Variable-Length Argument Lists

- Functions with unspecified number of arguments
 - Load `<stdarg.h>`
 - Use ellipsis(`...`) at end of parameter list
 - Need at least one defined parameter
 - Example:

```
double myfunction ( int i, ... );
```
 - The ellipsis is only used in the prototype of a function with a variable length argument list
 - `printf` is an example of a function that can take multiple arguments
 - The prototype of `printf` is defined as

```
int printf( const char* format, ... );
```



14.3 Variable-Length Argument Lists

Identifier	Explanation
<code>va_list</code>	A type suitable for holding information needed by macros <code>va_start</code> , <code>va_arg</code> and <code>va_end</code> . To access the arguments in a variable-length argument list, an object of type <code>va_list</code> must be declared.
<code>va_start</code>	A macro that is invoked before the arguments of a variable-length argument list can be accessed. The macro initializes the object declared with <code>va_list</code> for use by the <code>va_arg</code> and <code>va_end</code> macros.
<code>va_arg</code>	A macro that expands to an expression of the value and type of the next argument in the variable-length argument list. Each invocation of <code>va_arg</code> modifies the object declared with <code>va_list</code> so that the object points to the next argument in the list.
<code>va_end</code>	A macro that facilitates a normal return from a function whose variable-length argument list was referred to by the <code>va_start</code> macro.
Fig. 14.1 The type and the macros defined in header <code>stdarg.h</code> .	





Outline



fig14_02.c (Part 1 of 2)

```
1  /* Fig. 14.2: fig14_02.c
2     Using variable-length argument lists */
3  #include <stdio.h>
4  #include <stdarg.h>
5
6  double average( int i, ... ); /* prototype */
7
8  int main()
9  {
10     double w = 37.5; /* initialize w */
11     double x = 22.5; /* initialize x */
12     double y = 1.7;  /* initialize y */
13     double z = 10.2; /* initialize z */
14
15     printf( "%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n",
16            "w = ", w, "x = ", x, "y = ", y, "z = ", z );
17     printf( "%s%.3f\n%s%.3f\n%s%.3f\n",
18            "The average of w and x is ", average( 2, w, x ),
19            "The average of w, x, and y is ", average( 3, w, x, y ),
20            "The average of w, x, y, and z is ",
21            average( 4, w, x, y, z ) );
22
23     return 0; /* indicates successful termination */
24
25 } /* end main */
26
```




[Outline](#)



fig14_02.c (Part 2 of 2)

```
27 /* calculate average */
28 double average( int i, ... )
29 {
30     double total = 0; /* initialize total */
31     int j;           /* counter */
32     va_list ap;      /* for storing information needed by va_start */
33
34     va_start( ap, i ); /* initialize ap for use in va_arg and va_end */
35
36     /* process variable length argument list */
37     for ( j = 1; j <= i; j++ ) {
38         total += va_arg( ap, double );
39     } /* end for */
40
41     va_end( ap ); /* end the va_start */
42
43     return total / i; /* calculate average */
44
45 } /* end function average */
```

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2
```

```
The average of w and x is 30.000
The average of w, x, and y is 20.567
The average of w, x, y, and z is 17.975
```

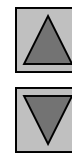
Program Output

14.4 Using Command-Line Arguments

- Pass arguments to `main` on DOS or UNIX
 - Define `main` as

```
int main( int argc, char *argv[] )
```
 - `int argc`
 - Number of arguments passed
 - `char *argv[]`
 - Array of strings
 - Has names of arguments in order
 - `argv[0]` is first argument
 - Example: `$ mycopy input output`
 - `argc`: 3
 - `argv[0]`: "mycopy"
 - `argv[1]`: "input"
 - `argv[2]`: "output"





Outline

fig14_03.c (Part 1 of 2)

```
1  /* Fig. 14.3: fig14_03.c
2     Using command-line arguments */
3  #include <stdio.h>
4
5  int main( int argc, char *argv[] )
6  {
7     FILE *inFilePtr; /* input file pointer */
8     FILE *outFilePtr; /* output file pointer */
9     int c;           /* define c to hold characters input by user */
10
11    /* check number of command-line arguments */
12    if ( argc != 3 ) {
13        printf( "Usage: copy infile outfile\n" );
14    } /* end if */
15    else {
16
17        /* if input file can be opened */
18        if ( ( inFilePtr = fopen( argv[ 1 ], "r" ) ) != NULL ) {
19
20            /* if output file can be opened */
21            if ( ( outFilePtr = fopen( argv[ 2 ], "w" ) ) != NULL ) {
22
23                /* read and output characters */
24                while ( ( c = fgetc( inFilePtr ) ) != EOF ) {
25                    fputc( c, outFilePtr );
26                } /* end while */
27
28            } /* end if */

```

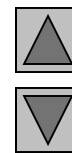
Notice argc and argv[] in main

argv[1] is the second argument, and is being read.

argv[2] is the third argument, and is being written to.

Loop until End Of File. fgetc a character from inFilePtr and fputc it into outFilePtr.

```
29     else { /* output file could not be opened */
30         printf( "File \"%s\" could not be opened\n", argv[ 2 ] );
31     } /* end else */
32
33 } /* end if */
34 else { /* input file could not be opened */
35     printf( "File \"%s\" could not be opened\n", argv[ 1 ] );
36 } /* end else */
37
38 } /* end else */
39
40 return 0; /* indicates successful termination */
41
42 } /* end main */
```



Outline

fig14_03.c (Part 2 of 2)

14.5 Notes on Compiling Multiple-Source-File Programs

- Programs with multiple source files
 - Function definition must be in one file (cannot be split up)
 - Global variables accessible to functions in same file
 - Global variables must be defined in every file in which they are used
 - Example:
 - If integer `flag` is defined in one file
 - To use it in another file you must include the statement
`extern int flag;`
 - `extern`
 - States that the variable is defined in another file
 - Function prototypes can be used in other files without an `extern` statement
 - Have a prototype in each file that uses the function



14.5 Notes on Compiling Multiple-Source-File Programs

- Keyword `static`
 - Specifies that variables can only be used in the file in which they are defined
- Programs with multiple source files
 - Tedious to compile everything if small changes have been made to only one file
 - Can recompile only the changed files
 - Procedure varies on system
 - UNIX: `make` utility



14.6 Program Termination with `exit` and `atexit`

- Function `exit`

- Forces a program to terminate
- Parameters – symbolic constants `EXIT_SUCCESS` or `EXIT_FAILURE`
- Returns an implementation-defined value
- Example:

```
exit( EXIT_SUCCESS );
```

- Function `atexit`

```
atexit( functionToRun );
```

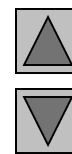
- Registers `functionToRun` to execute upon successful program termination
 - `atexit` itself does not terminate the program
- Register up to 32 functions (multiple `atexit()` statements)
 - Functions called in reverse register order
- Called function cannot take arguments or return values



```

1  /* Fig. 14.4: fig14_04.c
2     Using the exit and atexit functions */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  void print( void ); /* prototype */
7
8  int main()
9  {
10     int answer; /* user's menu choice */
11
12     atexit( print ); /* register function print */
13     printf( "Enter 1 to terminate program with function exit"
14            "\nEnter 2 to terminate program normally\n" );
15     scanf( "%d", &answer );
16
17     /* exit if answer is 1 */
18     if ( answer == 1 ) {
19         printf( "\nTerminating program with function exit\n" );
20         exit( EXIT_SUCCESS );
21     } /* end if */
22
23     printf( "\nTerminating program by reaching the end of main\n" );
24

```



Outline

fig14_04.c (Part 1 of 2)


```
25     return 0; /* indicates successful termination */
26
27 } /* end main */
28
29 /* display message before termination */
30 void print( void )
31 {
32     printf( "Executing function print at program "
33           "termination\nProgram terminated\n" );
34 } /* end function print */
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 1
```

```
Terminating program with function exit
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 2
```

```
Terminating program by reaching the end of main
Executing function print at program termination
Program terminated
```



Outline



fig14_04.c (Part 2 of 2)

Program Output

14.7 The volatile Type Qualifier

- volatile qualifier
 - Variable may be altered outside program
 - Variable not under control of program
 - Variable cannot be optimized



14.8 Suffixes for Integer and Floating-Point Constants

- C provides suffixes for constants
 - unsigned integer – u or U
 - long integer – l or L
 - unsigned long integer – ul, Ul, UL or LU
 - float – f or F
 - long double – l or L
 - Examples:
 - 174u
 - 467L
 - 3451ul
 - If integer constant is not suffixed type determined by first type capable of storing a value of that size (int, long int, unsigned long int)
 - If floating point not suffixed of type double



14.9 More on Files

- C can process binary files
 - Not all systems support binary files
 - Files opened as text files if binary mode not supported
 - Binary files should be used when rigorous speed, storage, and compatibility conditions demand it
 - Otherwise, text files are preferred
 - Inherent portability, can use standard tools to examine data
- Function `tmpfile`
 - Opens a temporary file in mode `"wb+"`
 - Some systems may process temporary files as text files
 - Temporary file exists until closed with `fclose` or until program terminates
- Function `rewind`
 - Positions file pointers to the beginning of the file

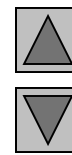


14.9 More on Files

- File open modes:

Mode	Description
rb	Open a binary file for reading.
wb	Create a binary file for writing. If the file already exists, discard the current contents.
ab	Append; open or create a binary file for writing at end-of-file.
rb+	Open a binary file for update (reading and writing).
wb+	Create a binary file for update. If the file already exists, discard the current contents.
ab+	Append; open or create a binary file for update; all writing is done at the end of the file





Outline

fig14_06.c (Part 1 of 2)

```
1  /* Fig. 14.6: fig14_06.c
2     Using temporary files */
3  #include <stdio.h>
4
5  int main()
6  {
7     FILE *filePtr;      /* pointer to file being modified */
8     FILE *tempFilePtr; /* temporary file pointer */
9     int c;              /* define c to hold characters input by user */
10    char fileName[ 30 ]; /* create char array */
11
12    printf( "This program changes tabs to spaces.\n"
13           "Enter a file to be modified: " );
14    scanf( "%29s", fileName );
15
16    /* fopen opens the file */
17    if ( ( filePtr = fopen( fileName, "r+" ) ) != NULL ) {
18
19        /* create temporary file */
20        if ( ( tempFilePtr = tmpfile() ) != NULL ) {
21            printf( "\nThe file before modification is:\n" );
22
23            /* read characters */
24            while ( ( c = getc( filePtr ) ) != EOF ) {
25                putchar( c );
26                putc( c == '\t' ? ' ': c, tempFilePtr );
27            } /* end while */
28
```



Outline

fig14_06.c (Part 2 of 2)

```
29     rewind( tempFilePtr );
30     rewind( filePtr );
31     printf( "\n\nThe file after modification is:\n" );
32
33     /* read characters */
34     while ( ( c = getc(tempFilePtr) ) != EOF ) {
35         putchar( c );
36         putc( c, filePtr );
37     } /* end while */
38
39     } /* end if */
40     else { /* if temporary file could not be opened */
41         printf( "Unable to open temporary file\n" );
42     } /* end else */
43
44     } /* end if */
45     else { /* if file could not be opened */
46         printf( "Unable to open %s\n", fileName );
47     } /* end else */
48
49     return 0; /* indicates successful termination */
50
51 } /* end main */
```

```
This program changes tabs to spaces.  
Enter a file to be modified: data.txt
```

```
The file before modification is:
```

```
0      1      2      3      4  
      5      6      7      8      9
```

```
The file after modification is:
```

```
0 1 2 3 4  
 5 6 7 8 9
```



Outline



Program Output

14.10 Signal Handling

- Signal
 - Unexpected event, can terminate program
 - Interrupts (`<ctrl> c`), illegal instructions, segmentation violations, termination orders, floating-point exceptions (division by zero, multiplying large floats)
- Function `signal`
 - Traps unexpected events
 - Header `<signal.h>`
 - Receives two arguments a signal number and a pointer to the signal handling function
- Function `raise`
 - Takes an integer signal number and creates a signal



14.10 Signal Handling

- Signals defined in `signal.h`

Signal	Explanation
SIGABRT	Abnormal termination of the program (such as a call to <code>abort</code>).
SIGFPE	An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow.
SIGILL	Detection of an illegal instruction.
SIGINT	Receipt of an interactive attention signal.
SIGSEGV	An invalid access to storage.
SIGTERM	A termination request sent to the program.





```
1  /* Fig. 14.8: fig14_08.c
2     Using signal handling */
3  #include <stdio.h>
4  #include <signal.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  void signal_handler( int signalValue ); /* prototype */
9
10 int main()
11 {
12     int i; /* counter */
13     int x; /* variable to hold random values between 1-50 */
14
15     signal( SIGINT, signal_handler );
16     srand( clock() );
17
18     /* output numbers 1 to 100 */
19     for ( i = 1; i <= 100; i++ ) {
20         x = 1 + rand() % 50; /* generate random number to raise SIGINT */
21
22         /* raise SIGINT when x is 25 */
23         if ( x == 25 ) {
24             raise( SIGINT );
25         } /* end if */
26
```

signal set to call function
signal_handler when a signal
of type SIGINT occurs.





Outline



fig14_08.c (Part 2 of 3)

```
27     printf( "%4d", i );
28
29     /* output \n when i is a multiple of 10 */
30     if ( i % 10 == 0 ) {
31         printf( "\n" );
32     } /* end if */
33
34 } /* end for */
35
36 return 0; /* indicates successful termination */
37
38 } /* end main */
39
40 /* handles signal */
41 void signal_handler( int signalValue )
42 {
43     int response; /* user's response to signal (1 or 2) */
44
45     printf( "%s%d%s\n%s",
46           "\nInterrupt signal ( ", signalValue, " ) received.",
47           "Do you wish to continue ( 1 = yes or 2 = no )? " );
48
49     scanf( "%d", &response );
50
```

User given option of terminating program



Outline



fig14_08.c (Part 3 of 3)

```
51  /* check for invalid responses */
52  while ( response != 1 && response != 2 ) {
53      printf( "( 1 = yes or 2 = no )? " );
54      scanf( "%d", &response );
55  } /* end while */
56
57  /* determine if it is time to exit */
58  if ( response == 1 ) {
59
60      /* call signal and pass it SIGINT and address of signalHandler */
61      signal( SIGINT, signal_handler );
62  } /* end if */
63  else {
64      exit( EXIT_SUCCESS );
65  } /* end else */
66
67 } /* end function signalHandler */
```

Signal handler reinitialized by calling
signal again

```
 1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93
```

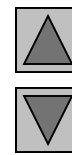
Interrupt signal (2) received.

Do you wish to continue (1 = yes or 2 = no)? 1

```
94 95 96
```

Interrupt signal (2) received.

Do you wish to continue (1 = yes or 2 = no)? 2



Outline

Program Output

14.11 Dynamic Memory Allocation with `calloc` and `realloc`

- Dynamic memory allocation
 - Can create dynamic arrays
- `calloc(nmembers, size)`
 - *nmembers* – number of elements
 - *size* – size of each element
 - Returns a pointer to a dynamic array
- `realloc(pointerToObject, newSize)`
 - *pointerToObject* – pointer to the object being reallocated
 - *newSize* – new size of the object
 - Returns pointer to reallocated memory
 - Returns NULL if cannot allocate space
 - If *newSize* equals 0 then the object pointed to is freed
 - If *pointerToObject* equals 0 then it acts like `malloc`



14.12 The Unconditional Branch: goto

- Unstructured programming
 - Use when performance crucial
 - `break` to exit loop instead of waiting until condition becomes `false`
- `goto` statement
 - Changes flow control to first statement after specified label
 - A label is an identifier followed by a colon (i.e. `start:`)
 - Quick escape from deeply nested loop
`goto start;`





[Outline](#)



fig14_09.c

Notice how start: , end: and goto are used

```
1  /* Fig. 14.9: fig14_09.c
2     Using goto */
3  #include <stdio.h>
4
5  int main()
6  {
7     int count = 1; /* initialize count */
8
9     start: /* label */
10
11     if ( count > 10 ) {
12         goto end;
13     } /* end if */
14
15     printf( "%d ", count );
16     count++;
17
18     goto start; /* goto start on line 9 */
19
20     end: /* label */
21     putchar( '\n' );
22
23     return 0; /* indicates successful termination */
24
25 } /* end main */
```

1 2 3 4 5 6 7 8 9 10

Program Output