Chapter 14 - Advanced C Topics

Outline

14.1 Introduction

- Several advanced topics in this chapter
- Many capabilities are specific to operating systems (especially UNIX and/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 C++ feature
  - UNIX and DOS
  - **$** or **%** represents command line

  Example: `$ myProgram < input`
  - Rather than inputting values by hand, read them from a file
- Pipe command (**|**)
  - Output of one program becomes input of another

  `$ firstProgram | secondProgram`
  - Output of `firstProgram` goes to `secondProgram`

14.2 Redirecting Input/Output on UNIX and DOS Systems (II)

- Redirect output (**>**)
  - Determines where output of a program goes
  - `$ myProgram > myFile`
    - Output goes into `myFile` (erases previous contents)
- Append output (**>>**)
  - Add output to end of file (preserve previous contents)
  - `$ myOtherProgram >> myFile`
    - Output goes to the end of `myFile`

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

```
double myfunction (int i, ...);
```

  - Prototype with variable length argument list
  - Example: prototype of **`printf`**

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


14.3 Variable-Length Argument Lists (II)

- Macros and declarations in function definition

`va_list`
- Type specifier, required (**`va_list arguments;`**)

`va_start(arguments,` *other variables***`)`**
- Intializes parameters, required before use

`va_arg(arguments,` *type***`)`**
- Returns a parameter each time **`va_arg`** is called
- Automatically points to next parameter

`va_end(arguments)`
- Helps function have a normal return

Example of variable-length argument lits

```
/* Fig. 14.2: fig14_02.c
   Using variable-length argument lists */
#include <stdio.h>
#include <stdarg.h>

double average( int, ... );

int main()
{
   double w = 37.5, x = 22.5, y = 1.7, z = 10.2;

   printf( "%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n",
           "w = ", w, "x = ", x, "y = ", y, "z = ", z );
   printf( "%s%.3f\n%s%.3f\n%s%.3f\n",
           "The average of w and x is ",
           average( 2, w, x ),
           "The average of w, x, and y is ",
           average( 3, w, x, y ),
           "The average of w, x, y, and z is ",
           average( 4, w, x, y, z ) );

   return 0;
}
```



```
double average( int i, ... )
{
   double total = 0;
   int j;
   va_list ap;

   va_start( ap, i );

   for ( j = 1; j <= i; j++ )
      total += va_arg( ap, double );

   va_end( ap );
   return total / i;
}
```

Outline

```
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
```


14.4 Using Command-Line Arguments

- Pass arguments to **`main`** in DOS and UNIX

  `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: **`$ copy input output`**

  `argc: 3`

  `argv[ 0 ]: "copy"`

  `argv[ 1 ]: "input"`

  `argv[ 2 ]: "output"`

Outline

```
/* Fig. 14.3: fig14_03.c
   Using command-line arguments */
#include <stdio.h>

int main( int argc, char *argv[] )
{
   FILE *inFilePtr, *outFilePtr;
   int c;

   if ( argc != 3 )
      printf( "Usage: copy infile outfile\n" );
   else
      if ( ( inFilePtr = fopen( argv[ 1 ], "r" ) )

         if ( ( outFilePtr = fopen( argv[ 2 ], "w" ) ) != NULL )

            while ( ( c = fgetc( inFilePtr ) ) != EOF )
               fputc( c, outFilePtr );

         else
            printf( "File \"%s\" could n

      else
         printf( "File \"%s\" could not be opened\n", argv[ 1 ] );

   return 0;
}
```

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`**.

1. Initialize variables

y open type rite)

3. Copy file

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 they are used
  - Example:
    - Integer **`myGlobal`** defined in one file
    - To use in another file:

      `extern int myGlobal;`

      - **`extern`** - states that variable defined elsewhere (i.e., not in that file)

14.5 Notes on Compiling Multiple-Source-File Programs (II)

- Programs with multiple source files (continued)
  - Function prototypes can be used in other files, **`extern`** not needed
    - Have a prototype in each file that uses the function
  - Example: loading header files
    - **`#include <cstring>`**
    - Contains prototypes of functions
    - We do not know where definitions are

14.5 Notes on Compiling Multiple-Source-File Programs (III)

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

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

- Dynamic memory allocation
  - Can create dynamic arrays
- **`void *calloc(size_t nmemb, size_t size)`**
  - **`nmemb`** - number of members
  - **`size`** - size of each member
  - Returns pointer to dynamic array
- **`void *realloc(void *ptr, size_t size)`**
  - **`ptr`** - pointer to the object being reallocated
  - **`size`** - new size of the object
  - Returns pointer to reallocated memory
  - Returns **`NULL`** if cannot allocate space
  - If **`newSize = 0`**, object freed
  - If **`pointerToObject = 0`**, 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
  - Label: identifier and colon (i.e. **`start:`**)
  - Quick escape from deeply nested loop
  - **`goto start;`**

```c
/* Fig. 14.9: fig14_09.c
   Using goto */
#include <stdio.h>

int main()
{
   int count = 1;

   start:                        /* label */
      if ( count > 10 )
         goto end;

      printf( "%d  ", count );
      ++count;
      goto start;

   end:                          /* label */
      putchar( '\n' );

   return 0;
}
```

Notice how **`start:`**, **`end:`** and **`goto`** are used

Outline

1. Initialize variable

1.1 Labels

2. Loop

3. Print

Program Output

```
1  2  3  4  5  6  7  8  9  10
```


14.6 Program Termination with `exit` and `atexit`

- Function **`exit`**
  - Forces a program to terminate
  - Parameters - symbolic constants **`EXIT_SUCCESS`** or **`EXIT_FAILURE`**
  - Returns implementation-defined value
  - **`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


```c
/* Fig. 14.4: fig14_04.c
   Using the exit and atexit functions */
#include <stdio.h>
#include <stdlib.h>

void print( void );

int main()
{
   int answer;

   atexit( print );       /* register function print */
   printf( "Enter 1 to terminate program with function exit"
           "\nEnter 2 to terminate program normally\n" );
   scanf( "%d", &answer );

   if ( answer == 1 ) {
      printf( "\nTerminating program with "
              "function exit\n" );
      exit( EXIT_SUCCESS );
   }

   printf( "\nTerminating program by reaching"
           " the end of main\n" );
   return 0;
}

void print( void )
{
   printf( "Executing function print at program "
           "termination\nProgram terminated\n" );
}
```

Outline

1. Register function `print` using `atexit`

2. User input

3. Output

3.1 Function definition

Outline

Program Output

```
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
```

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.

  Integer - **`u`** or **`U`** (**`unsigned`** integer)
  `long` integer - **`l`** or **`L`**
  `unsigned long` integer - **`ul`** or **`UL`**
  `float` - **`f`** or **`F`**
  `long double` - **`l`** or **`L`**

  Examples:
  ```
  174u
  467L
  3451ul
  ```
- Defaults
  - Integers: lowest type that holds them (**`int`**, **`long int`**, **`unsigned long int`**)
  - Floating point numbers: **`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 when rigorous speed, storage, and compatibility conditions demand it
  - Otherwise, text files preferred
    - Inherent portability, can use standard tools to examine data
  - 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

14.9 More on Files (II)

- Function **`tmpfile`**
  - In standard library
  - 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

14.10 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>`**
  - Two arguments: signal number, pointer to function to handle it
- Function **`raise`**
  - Takes in integer signal number and creates signal

14.10 Signal Handling (II)

Signal	Explanation
SIGABRT	Abnormal termination of the program (such as a call to abort).
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.


```c
/* Fig. 14.8: fig14_08.c
   Using signal handling */
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <time.h>

void signal_handler( int );

int main()
{
   int i, x;

   signal( SIGINT, signal_handler );
   srand( clock() );

   for ( i = 1; i <= 100; i++ ) {
      x = 1 + rand() % 50;

      if ( x == 25 )
         raise( SIGINT );

      printf( "%4d", i );

      if ( i % 10 == 0 )
         printf( "\n" );
   }

   return 0;
}

void signal_handler( int signalValue )
```

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

Outline

1. Function prototype

Generate random mber

2.1 `raise` signal if `x == 25`

3. Function definition

```c
{
   int response;

   printf( "%s%d%s\n%s",
      "\nInterrupt signal ( ", signalValue, " ) received.",
      "Do you wish to continue ( 1 = yes or 2 = no )? " );

   scanf( "%d", &response );

   while ( response != 1 && response != 2 ) {
      printf( "( 1 = yes or 2 = no )? " );
      scanf( "%d", &response );
   }

   if ( response == 1 )
      signal( SIGINT, signal_handler );
   else
      exit( EXIT_SUCCESS );
}
```

User given option of terminating program

Signal handler reinitialized by calling **`signal`** again

Outline

3. Function definition

```
   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
```

Program Output

Outline

Program Output

```
  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 )? 1
  97  98  99 100
```