*CSC 141 Introduction to Computer Programming*

Practice Exercises

Hands on Demo

Arrays, pointers and strings

Examples from How to program C 7/e

by Deital chapter -7

Website Reference:

http://www.deitel.com/Books/C/CHowtoProgram7e

1-Ptr-operators

//Fig. 7.4: fig07\_04.c

// Using the & and \* pointer operators.

#include <stdio.h>

int main( void )

{

 int a; // a is an integer

 int \*aPtr; // aPtr is a pointer to an integer

 a = 7;

 aPtr = &a; // set aPtr to the address of a

 printf( "The address of a is %p"

 "\nThe value of aPtr is %p", &a, aPtr );

 printf( "\n\nThe value of a is %d"

 "\nThe value of \*aPtr is %d", a, \*aPtr );

 printf( "\n\nShowing that \* and & are complements of "

 "each other\n&\*aPtr = %p"

 "\n\*&aPtr = %p\n", &\*aPtr, \*&aPtr );

} // end main

Skip

// Fig. 7.6: fig07\_06.c

// Cube a variable using pass-by-value.

#include <stdio.h>

int cubeByValue( int n ); // prototype

int main( void )

{

 int number = 5; // initialize number

 printf( "The original value of number is %d", number );

 // pass number by value to cubeByValue

 number = cubeByValue( number );

 printf( "\nThe new value of number is %d\n", number );

} // end main

// calculate and return cube of integer argument

int cubeByValue( int n )

{

 return n \* n \* n; // cube local variable n and return result

} // end function cubeByValue

2-pass-by-reference

// Fig. 7.7: fig07\_07.c

// Cube a variable using pass-by-reference with a pointer argument.

#include <stdio.h>

void cubeByReference( int \*nPtr ); // function prototype

int main( void )

{

 int number = 5; // initialize number

 printf( "The original value of number is %d", number );

 // pass address of number to cubeByReference

 cubeByReference( &number );

 printf( "\nThe new value of number is %d\n", number );

} // end main

// calculate cube of \*nPtr; actually modifies number in main

void cubeByReference( int \*nPtr )

{

 \*nPtr = \*nPtr \* \*nPtr \* \*nPtr; // cube \*nPtr

} // end function cubeByReference

3-Convert-str-2-upper

// Fig. 7.10: fig07\_10.c

// Converting a string to uppercase using a

// non-constant pointer to non-constant data.

#include <stdio.h>

#include <ctype.h>

void convertToUppercase( char \*sPtr ); // prototype

int main( void )

{

 char string[] = "cHaRaCters and $32.98"; // initialize char array

 printf( "The string before conversion is: %s", string );

 convertToUppercase( string );

 printf( "\nThe string after conversion is: %s\n", string );

} // end main

// convert string to uppercase letters

void convertToUppercase( char \*sPtr )

{

 while ( \*sPtr != '\0' ) { // current character is not '\0'

 \*sPtr = toupper( \*sPtr ); // convert to uppercase

 ++sPtr; // make sPtr point to the next character

 } // end while

} // end function convertToUppercase

4-Print-str-thru-ptr

// Fig. 7.11: fig07\_11.c

// Printing a string one character at a time using

// a non-constant pointer to constant data.

#include <stdio.h>

void printCharacters( const char \*sPtr );

int main( void )

{

 // initialize char array

 char string[] = "print characters of a string";

 puts( "The string is:" );

 printCharacters( string );

 puts( "" );

 system("pause");

} // end main

// sPtr cannot modify the character to which it points, But the pointer is changing by itself

// i.e., sPtr is a "read-only" pointer

void printCharacters( const char \*sPtr )

{

 // loop through entire string

 for ( ; \*sPtr != '\0'; ++sPtr ) { // no initialization

 printf( "%c", \*sPtr );

 } // end for

} // end function printCharacters

5-p-ptr-error.cpp

// Fig. 7.12: fig07\_12.c

// Attempting to modify data through a

// non-constant pointer to constant data.

#include <stdio.h>

void f( const int \*xPtr ); // prototype

int main( void )

{

 int y; // define y

 f( &y ); // f attempts illegal modification

} // end main

// xPtr cannot be used to modify the

// value of the variable to which it points

void f( const int \*xPtr )

{

 \*xPtr = 100; // error: cannot modify a const object

} // end function f

5-p-ptr-error.cpp

Combine with previous

// Fig. 7.13: fig07\_13.c

// Attempting to modify a constant pointer to non-constant data.

#include <stdio.h>

int main( void )

{

 int x; // define x

 int y; // define y

 // ptr is a constant pointer to an integer that can be modified

 // through ptr, but ptr always points to the same memory location

 int \* const ptr = &x;

 \*ptr = 7; // allowed: \*ptr is not const

 ptr = &y; // error: ptr is const; cannot assign new address

} // end main

5-p-ptr-error.cpp

Combine with previous

// Fig. 7.14: fig07\_14.c

// Attempting to modify a constant pointer to constant data.

#include <stdio.h>

int main( void )

{

 int x = 5; // initialize x

 int y; // define y

 // ptr is a constant pointer to a constant integer. ptr always

 // points to the same location; the integer at that location

 // cannot be modified

 const int \*const ptr = &x; // initialization is OK

 printf( "%d\n", \*ptr );

 \*ptr = 7; // error: \*ptr is const; cannot assign new value

 ptr = &y; // error: ptr is const; cannot assign new address

} // end main

6-Bubble-sort-with-ptrs

// Fig. 7.15: fig07\_15.c

// Putting values into an array, sorting the values into

// ascending order, and printing the resulting array.

#include <stdio.h>

#define SIZE 10

void bubbleSort( int \* const array, size\_t size ); // prototype

int main( void )

{

 // initialize array a

 int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

 size\_t i; // counter

 puts( "Data items in original order" );

 // loop through array a

 for ( i = 0; i < SIZE; ++i ) {

 printf( "%4d", a[ i ] );

 } // end for

 bubbleSort( a, SIZE ); // sort the array

 puts( "\nData items in ascending order" );

 // loop through array a

 for ( i = 0; i < SIZE; ++i ) {

 printf( "%4d", a[ i ] );

 } // end for

 puts( "" );

} // end main

// sort an array of integers using bubble sort algorithm

void bubbleSort( int \* const array, size\_t size )

{

 void swap( int \*element1Ptr, int \*element2Ptr ); // prototype

 unsigned int pass; // pass counter

 size\_t j; // comparison counter

 // loop to control passes

 for ( pass = 0; pass < size - 1; ++pass ) {

 // loop to control comparisons during each pass

 for ( j = 0; j < size - 1; ++j ) {

 // swap adjacent elements if they’re out of order

 if ( array[ j ] > array[ j + 1 ] ) {

 swap( &array[ j ], &array[ j + 1 ] );

 } // end if

 } // end inner for

 } // end outer for

} // end function bubbleSort

// swap values at memory locations to which element1Ptr and

// element2Ptr point

void swap( int \*element1Ptr, int \*element2Ptr )

{

 int hold = \*element1Ptr;

 \*element1Ptr = \*element2Ptr;

 \*element2Ptr = hold;

} // end function swap

skip

// Fig. 7.16: fig07\_16.c

// Applying sizeof to an array name returns

// the number of bytes in the array.

#include <stdio.h>

#define SIZE 20

size\_t getSize( float \*ptr ); // prototype

int main( void )

{

 float array[ SIZE ]; // create array

 printf( "The number of bytes in the array is %u"

 "\nThe number of bytes returned by getSize is %u\n",

 sizeof( array ), getSize( array ) );

} // end main

// return size of ptr

size\_t getSize( float \*ptr )

{

 return sizeof( ptr );

} // end function getSize

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

7-Sizeof-data-types

// Fig. 7.17: fig07\_17.c

// Using operator sizeof to determine standard data type sizes.

#include <stdio.h>

int main( void )

{

 char c;

 short s;

 int i;

 long l;

 long long ll;

 float f;

 double d;

 long double ld;

 int array[ 20 ]; // create array of 20 int elements

 int \*ptr = array; // create pointer to array

 printf( " sizeof c = %u\tsizeof(char) = %u"

 "\n sizeof s = %u\tsizeof(short) = %u"

 "\n sizeof i = %u\tsizeof(int) = %u"

 "\n sizeof l = %u\tsizeof(long) = %u"

 "\n sizeof ll = %u\tsizeof(long long) = %u"

 "\n sizeof f = %u\tsizeof(float) = %u"

 "\n sizeof d = %u\tsizeof(double) = %u"

 "\n sizeof ld = %u\tsizeof(long double) = %u"

 "\n sizeof array = %u"

 "\n sizeof ptr = %u\n",

 sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,

 sizeof( int ), sizeof l, sizeof( long ), sizeof ll,

 sizeof( long long ), sizeof f, sizeof( float ), sizeof d,

 sizeof( double ), sizeof ld, sizeof( long double ),

 sizeof array, sizeof ptr );

} // end main

8-Subscripts-and-ptrs

// Fig. 7.20: fig07\_20.cpp

// Using subscripting and pointer notations with arrays.

#include <stdio.h>

#define ARRAY\_SIZE 4

int main( void )

{

 int b[] = { 10, 20, 30, 40 }; // create and initialize array b

 int \*bPtr = b; // create bPtr and point it to array b

 size\_t i; // counter

 size\_t offset; // counter

 // output array b using array subscript notation

 puts( "Array b printed with:\nArray subscript notation" );

 // loop through array b

 for ( i = 0; i < ARRAY\_SIZE; ++i ) {

 printf( "b[ %u ] = %d\n", i, b[ i ] );

 } // end for

 // output array b using array name and pointer/offset notation

 puts( "\nPointer/offset notation where\n"

 "the pointer is the array name" );

 // loop through array b

 for ( offset = 0; offset < ARRAY\_SIZE; ++offset ) {

 printf( "\*( b + %u ) = %d\n", offset, \*( b + offset ) );

 } // end for

 // output array b using bPtr and array subscript notation

 puts( "\nPointer subscript notation" );

 // loop through array b

 for ( i = 0; i < ARRAY\_SIZE; ++i ) {

 printf( "bPtr[ %u ] = %d\n", i, bPtr[ i ] );

 } // end for

 // output array b using bPtr and pointer/offset notation

 puts( "\nPointer/offset notation" );

 // loop through array b

 for ( offset = 0; offset < ARRAY\_SIZE; ++offset ) {

 printf( "\*( bPtr + %u ) = %d\n", offset, \*( bPtr + offset ) );

 } // end for

} // end main

9-Copy-str-using-array-and-ptrs

// Fig. 7.21: fig07\_21.c

// Copying a string using array notation and pointer notation.

#include <stdio.h>

#define SIZE 10

void copy1( char \* const s1, const char \* const s2 ); // prototype

void copy2( char \*s1, const char \*s2 ); // prototype

int main( void )

{

 char string1[ SIZE ]; // create array string1

 char \*string2 = "Hello"; // create a pointer to a string

 char string3[ SIZE ]; // create array string3

 char string4[] = "Good Bye"; // create a pointer to a string

 copy1( string1, string2 );

 printf( "string1 = %s\n", string1 );

 copy2( string3, string4 );

 printf( "string3 = %s\n", string3 );

} // end main

// copy s2 to s1 using array notation

void copy1( char \* const s1, const char \* const s2 )

{

 size\_t i; // counter

 // loop through strings

 for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; ++i ) {

 ; // do nothing in body

 } // end for

} // end function copy1

// copy s2 to s1 using pointer notation

void copy2( char \*s1, const char \*s2 )

{

 // loop through strings

 for ( ; ( \*s1 = \*s2 ) != '\0'; ++s1, ++s2 ) {

 ; // do nothing in body

 } // end for

} // end function copy2

May-Skip

// Fig. 7.24: fig07\_24.c

// Card shuffling and dealing.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SUITS 4

#define FACES 13

#define CARDS 52

// prototypes

void shuffle( unsigned int wDeck[][ FACES ] ); // shuffling modifies wDeck

void deal( unsigned int wDeck[][ FACES ], const char \*wFace[],

 const char \*wSuit[] ); // dealing doesn't modify the arrays

int main( void )

{

 // initialize suit array

 const char \*suit[ SUITS ] =

 { "Hearts", "Diamonds", "Clubs", "Spades" };

 // initialize face array

 const char \*face[ FACES ] =

 { "Ace", "Deuce", "Three", "Four",

 "Five", "Six", "Seven", "Eight",

 "Nine", "Ten", "Jack", "Queen", "King" };

 // initialize deck array

 unsigned int deck[ SUITS ][ FACES ] = { 0 };

 srand( time( NULL ) ); // seed random-number generator

 shuffle( deck ); // shuffle the deck

 deal( deck, face, suit ); // deal the deck

} // end main

// shuffle cards in deck

void shuffle( unsigned int wDeck[][ FACES ] )

{

 size\_t row; // row number

 size\_t column; // column number

 size\_t card; // counter

 // for each of the cards, choose slot of deck randomly

 for ( card = 1; card <= CARDS; ++card ) {

 // choose new random location until unoccupied slot found

 do {

 row = rand() % SUITS;

 column = rand() % FACES;

 } while( wDeck[ row ][ column ] != 0 ); // end do...while

 // place card number in chosen slot of deck

 wDeck[ row ][ column ] = card;

 } // end for

} // end function shuffle

// deal cards in deck

void deal( unsigned int wDeck[][ FACES ], const char \*wFace[],

 const char \*wSuit[] )

{

 size\_t card; // card counter

 size\_t row; // row counter

 size\_t column; // column counter

 // deal each of the cards

 for ( card = 1; card <= CARDS; ++card ) {

 // loop through rows of wDeck

 for ( row = 0; row < SUITS; ++row ) {

 // loop through columns of wDeck for current row

 for ( column = 0; column < FACES; ++column ) {

 // if slot contains current card, display card

 if ( wDeck[ row ][ column ] == card ) {

 printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],

 card % 2 == 0 ? '\n' : '\t' ); // 2-column format

 } // end if

 } // end for

 } // end for

 } // end for

} // end function deal

Not taught yet

Pointers to functions

// Fig. 7.26: fig07\_26.c

// Multipurpose sorting program using function pointers.

#include <stdio.h>

#define SIZE 10

// prototypes

void bubble( int work[], size\_t size, int (\*compare)( int a, int b ) );

int ascending( int a, int b );

int descending( int a, int b );

int main( void )

{

 int order; // 1 for ascending order or 2 for descending order

 size\_t counter; // counter

 // initialize unordered array a

 int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

 printf( "%s", "Enter 1 to sort in ascending order,\n"

 "Enter 2 to sort in descending order: " );

 scanf( "%d", &order );

 puts( "\nData items in original order" );

 // output original array

 for ( counter = 0; counter < SIZE; ++counter ) {

 printf( "%5d", a[ counter ] );

 } // end for

 // sort array in ascending order; pass function ascending as an

 // argument to specify ascending sorting order

 if ( order == 1 ) {

 bubble( a, SIZE, ascending );

 puts( "\nData items in ascending order" );

 } // end if

 else { // pass function descending

 bubble( a, SIZE, descending );

 puts( "\nData items in descending order" );

 } // end else

 // output sorted array

 for ( counter = 0; counter < SIZE; ++counter ) {

 printf( "%5d", a[ counter ] );

 } // end for

 puts( "\n" );

} // end main

// multipurpose bubble sort; parameter compare is a pointer to

// the comparison function that determines sorting order

void bubble( int work[], size\_t size, int (\*compare)( int a, int b ) )

{

 unsigned int pass; // pass counter

 size\_t count; // comparison counter

 void swap( int \*element1Ptr, int \*element2ptr ); // prototype

 // loop to control passes

 for ( pass = 1; pass < size; ++pass ) {

 // loop to control number of comparisons per pass

 for ( count = 0; count < size - 1; ++count ) {

 // if adjacent elements are out of order, swap them

 if ( (\*compare)( work[ count ], work[ count + 1 ] ) ) {

 swap( &work[ count ], &work[ count + 1 ] );

 } // end if

 } // end for

 } // end for

} // end function bubble

// swap values at memory locations to which element1Ptr and

// element2Ptr point

void swap( int \*element1Ptr, int \*element2Ptr )

{

 int hold; // temporary holding variable

 hold = \*element1Ptr;

 \*element1Ptr = \*element2Ptr;

 \*element2Ptr = hold;

} // end function swap

// determine whether elements are out of order for an ascending

// order sort

int ascending( int a, int b )

{

 return b < a; // should swap if b is less than a

} // end function ascending

// determine whether elements are out of order for a descending

// order sort

int descending( int a, int b )

{

 return b > a; // should swap if b is greater than a

} // end function descending

Pointers to functions

// Fig. 7.28: fig07\_28.c

// Demonstrating an array of pointers to functions.

#include <stdio.h>

// prototypes

void function1( int a );

void function2( int b );

void function3( int c );

int main( void )

{

 // initialize array of 3 pointers to functions that each take an

 // int argument and return void

 void (\*f[ 3 ])( int ) = { function1, function2, function3 };

 size\_t choice; // variable to hold user's choice

 printf( "%s", "Enter a number between 0 and 2, 3 to end: " );

 scanf( "%u", &choice );

 // process user's choice

 while ( choice >= 0 && choice < 3 ) {

 // invoke function at location choice in array f and pass

 // choice as an argument

 (\*f[ choice ])( choice );

 printf( "%s", "Enter a number between 0 and 2, 3 to end: " );

 scanf( "%u", &choice );

 } // end while

 puts( "Program execution completed." );

} // end main

void function1( int a )

{

 printf( "You entered %d so function1 was called\n\n", a );

} // end function1

void function2( int b )

{

 printf( "You entered %d so function2 was called\n\n", b );

} // end function2

void function3( int c )

{

 printf( "You entered %d so function3 was called\n\n", c );

} // end function3