

C/C++ Programming 1 - Assignment 7

See the course website for detailed information on assignment requirements. All quiz and program submissions must be **MACHINE PRINTED**; handwritten materials are acceptable only for diagrams. Place the quiz first, followed by the program materials, as indicated below. For in-class turn-in please staple all materials together.

Quiz Requirements - 30% of Assignment Score

Quiz answers must be on one sheet in the following format. The choices I have shown are examples only:

Your name & programmer ID

Your email address

Assignment number

Date

Course Title, Section ID, Instructor's Name

1. A
 2. B
 3. C
 4. D
 5. E
 6. A
 7. B
 8. C
-

Exercise Requirements - 70% of Assignment Score

Separate exercises must be on separate sheets. Non-programming exercises (if any) simply require the submission of the answer(s) to the question(s). Programming exercises require that you submit the following for each, in the order shown (do not turn in compiler output):

1. the error-free & warning-free email report from the “Automated Assignment Checker”
2. the working, properly formatted program source code,
3. a screen dump (or equivalent) of the program run(s) and/or any files produced by the program, as appropriate. This **must** include the results of **all** required test cases plus any other appropriate tests.

All C program source code files (including any header files) must begin with a block comment in the form shown below (for C++ programs you may start each line with // instead, if desired). In addition, each individual function in the program (including *main*) must be preceded by a block comment describing its operation, parameters, and return. Appropriate comments related to pertinent lines or sections of code are also required.

```
/*
 * Your name & programmer ID
 * Your email address
 * Assignment and Exercise number
 * Date of development
 * Name of source file (myProgram.c, myProgram.cpp, etc.)
 * Your operating system (WinXP, Mac OS X, UNIX, LINUX, etc.)
 * Your compiler & version (Visual C++ 6.0, Borland 7.0, etc.)
 * Name of this course, section ID, and instructor
 *
 * Detailed description of the purpose of the code and any information pertinent to its
 * maintenance, limitations, peculiarities, user interface, and I/O requirements.
 */
```

Quiz (8 points)

This is a theoretical "paper only" quiz in which you must assume a perfectly implemented ANSI C/C++ compiler. How any particular program runs on your computer only indicates how your computer runs that program and not necessarily how it should run or how portable it is. There is only one correct answer to each question.

1. What is the data type of each element of *test*:

```
char *test[] = {"Hello", " world", "\n"};
```

 - A. **char**
 - B. **char ***
 - C. **char ****
 - D. Array of **const char**
 - E. Pointer to array of **const char**

2. Which additional three *printf* arguments will extract the message *the brown letter is in the mail* from array *p*:

```
static char *p[] =
    {"now's the", "how now brown",
     "my letter is in the mail"};
printf("%s %s %s", 3 arguments);
```

 - A. `&p[0][6], &*(p+1)[8], &p[2][3]`
 - B. `&p[2][16], &*((*(p+2))+3), &p[2][3]`
 - C. `&p[0][6], p[1]+8, &*((*(p+2))[3]`
 - D. `&*(p+0)[6], &p[1][8], &p[2][3]`
 - E. None will do it portably.

3. What is wrong with the following:

```
char ch = 'A';
int *ptr;
ptr = (int *)malloc(128 * sizeof(int));
*ptr = ch;
```

 - A. *malloc* produces an **void** pointer.
 - B. `(int *)malloc` must be `(char *)malloc`.
 - C. The success/failure of *malloc* is not tested.
 - D. **ptr* references uninitialized memory.
 - E. `*ptr = ch` must be `(int)*ptr = ch`.

4. To dynamically create one array of type **int**, another of type `(int *)`, and another of type **double**, where all 64 elements in each array must be initialized during allocation to a value of 0, what is wrong with the following?

```
calloc(64, sizeof(int));
calloc(64, sizeof(int *));
calloc(64, sizeof(double));
```

 - A. *calloc* does not clear memory – *malloc* does.
 - B. All three are not portable.
 - C. The last two are not portable.
 - D. Nothing is wrong.
 - E. *calloc* will only reliably allocate memory for integral types (**char**, **short**, **int**, **long**, etc.).

5. With no prototype present what data type gets passed to function *xyz*:

```
struct Test { int a; } b = { (float)25.0 };
xyz(b);
```

 - A. **int**
 - B. **struct Test**
 - C. pointer to **struct Test**
 - D. **float**
 - E. **b**

6. If a structure is passed as a function argument and the function changes the value of one of the parameter's members,
 - A. the original structure is changed.
 - B. the original is changed only if **static**.
 - C. only the function's local copy is changed.
 - D. a pointer to the structure is returned.
 - E. the effect is compiler dependent.

7. Assume a C function named *fcn* returns type **void** and has one formal parameter, which is a pointer to the structure `struct junk { int x, y; } z`. Which is the prototype for *fcn* and the call to *fcn* that passes a pointer to *z*.
 - A. `struct junk *fcn(void);` and `fcn(&z)`
 - B. `void fcn(struct junk);` and `fcn(z)`
 - C. `void fcn(junk *);` and `fcn(&z)`
 - D. `void fcn(struct junk *);` and `fcn(&z)`
 - E. None of the above.

8. The main problem with passing/returning entire structures rather than passing/returning pointers or references to them is that it
 - A. is considered bad style.
 - B. is not permitted by some ANSI compilers.
 - C. can corrupt some structure members.
 - D. is inefficient.
 - E. There is no reason to avoid doing this.

Exercises (10 points)

The point value of each exercise is shown in parentheses.

A "**magic number**" is defined as a numeric literal (and in some cases a character or string literal) embedded in a program's code or comments. They make programs cryptic and difficult to maintain because their meaning is not obvious and, if they must be changed, the likelihood of missing one or changing the wrong one is high. Instead, the `#define` directive (in C), **const**-qualified variables (in C++), or enumerated data (C and C++) should be used to associate meaningful names with literals. There are a few cases, however, where magic numbers are acceptable, including:

0 & 1 as array indices or loop start/end values, 1 as an increment/decrement value, 2 as way to double/halve a value or as a divisor to check for odd/even (but not in this course), coefficients in some mathematical formulae, some print strings, cases dictated by common sense

1. (5) **C++ Program** - Write and test a function named *DetermineElapsedTime* that computes the time elapsed between the times stored in two structures. For example, if the first structure contains 3:45:15 (3 hours, 45 minutes, 15 seconds) and the second structure contains 9:44:03, *DetermineElapsedTime* computes 5:58:48. The template for each structure is defined as:

```
struct MyTime { int hours, minutes, seconds; };           /* do not change this definition */
```

- *DetermineElapsedTime* has only two parameters - both pointers to **const** *MyTime*.
- *DetermineElapsedTime* must not modify the contents of either structure.
- *DetermineElapsedTime* must not declare any pointers other than its two parameters.
- *DetermineElapsedTime* must return a pointer to a *MyTime* structure containing the elapsed time.
- Use no external variables, including external structure variables. (The structure template definition is not a variable and typically should be defined externally.)
- Use no dynamic storage allocation.
- Do no scanning of user input or printing of any kind inside the *DetermineElapsedTime* function.
- Use military time: 23:59:59 is 1 second before midnight; 00:00:00 is midnight; 12:00:00 is noon.
- If the second time is less than or equal to the first, the second time is for the next day.
- You may (if desired) first convert all times to seconds inside *DetermineElapsedTime*.
- Write a test function that does the following:
 - prompts the user to input two times, each in the format: hours:minutes:seconds
 - inputs these values directly into two time structures
 - calls *DetermineElapsedTime* and prints the elapsed time result it provides
- Beware of potential integer overflow during multiplication!
- Returning a pointer to an automatic object is always wrong, as is dereferencing an uninitialized pointer.
- Don't place spaces around the `->` operator.

Include at least the following *firstTime/secondTime* pairs of input times when testing your program:

00:00:00 00:00:00 12:12:12 13:12:11 13:12:11 12:12:12

----- EXERCISE 2 IS ON THE NEXT PAGE -----

1 2. (5) **C Program** - Without modifying the members of the following structure template definition, write a
2 program following the steps listed. It must dynamically allocate exactly the right amount of memory
3 needed to hold the *name* strings input by the user during run time.

```
4  
5     struct Food  
6     {  
7         char *name;  
8         int weight, calories;  
9     };
```

- 10
11 1. Assume that an apple weighs 4 ounces and contains 100 calories, and a salad weighs 2 ounces
12 and contains 80 calories.
- 13 2. Declare *lunch[LUNCHES]* (where LUNCHES is #defined as 5) and initialize all three members
14 of only elements *lunch[0]* and *lunch[1]* in the same statement that defines the structure template.
15 Initialize to the values listed above about the apple and the salad.
- 16 3. At run time, prompt the user for the required initializers, and initialize all members of the
17 remaining elements of *lunch* to your choice of foods. Beware of using uninitialized pointers that,
18 when dereferenced, often cause core dumps or NULL pointer assignment messages.
- 19 4. Use no more space to store the name of each food than the string actually requires.

20
21 Print out a table showing all available information on the lunch menu, as stored in *lunch*.