Assignment 5
C/C++ Programming II

Exercise 1 *(4 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C2A5E1_SwapObjects.c**. Also add instructor-supplied source code file **C2A5E1_main-Driver.c**. Do not write a `main` function! `main` already exists in the instructor-supplied file and it will use the code you write.

File **C2A5E1_SwapObjects.c** must contain a function named `SwapObjects`.
`SwapObjects` syntax:

```
void SwapObjects(void *pa, void *pb, size_t size);
```

Parameters:
`pa` – a pointer to one of the objects to be swapped
`pb` – a pointer to the other object to be swapped
`size` – the number of bytes in each object
Synopsis:
Swaps the objects in `pa` and `pb`.
Return:
void

Do not use any kind of looping statement or call any function that is not from the standard C library.

If `SwapObjects` dynamically allocates memory it must also free it before returning.

All dynamic allocation results must be tested for success/failure before the memory is used. If allocation fails an error message is output to `stderr` and the program is terminated with an error code.

Submitting your solution

Send both source code files to the Assignment Checker with the subject line **C2A5E1_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

1. Merely swapping pointers `pa` and `pb` does not swap the objects they point to.
2. The only case where dynamically-allocated memory is freed automatically is when a program exits. Good programming practice dictates that dynamically-allocated memory always be explicitly freed by the program code as soon as it is no longer needed. Relying upon a program exit to free it is a bad programming practice.

Exercise 2 *(6 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C2A5E2_Create2D.c**. Also add instructor-supplied source code files **C2A5E2_Type-Driver.h** and **C2A5E2_main-Driver.c**. Do not write a `main` function! `main` already exists in the instructor-supplied implementation file and it will use the code you write.

Regarding data type `Type`, which is used in this exercise…

`Type` is a typedef'd data type that is defined in instructor-supplied header file **C2A5E2_Type-Driver.h**

Any file that uses this data type must include this header file using `#include`.

File **C2A5E2_Create2D.c** must contain functions named `Create2D` and `Free2D`.

`Create2D` syntax:

```
Type **Create2D(size_t rows, size_t cols);
```

Parameters:

`rows` – the number of rows in the 2-dimensional pointer array `Create2D` will create

`cols` – the number of columns in the 2-dimensional pointer array `Create2D` will create

Synopsis:

Creates a 2-dimensional pointer array of data type `Type` having the number of rows and columns specified by `rows` and `cols`. All memory needed for this array is dynamically-allocated at once using a single call to the appropriate memory allocation function. If allocation fails an error message is output to `stderr` and the program is terminated with an error code.

Return:

a pointer to the first pointer in the array

`Free2D` syntax:

```
void Free2D(void *p);
```

Parameters:

`p` – a pointer to the block of memory dynamically-allocated by `Create2D`

Synopsis:

Frees the dynamically-allocated block of memory pointed to by `p`.

Return:

void

Submitting your solution

Send all three source code files to the Assignment Checker with the subject line **C2A5E2_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

Make no assumptions about relationships between the sizes of pointers and the sizes of any other types, including other pointer types. If you attempt this exercise without fully understanding every aspect of note 14.4B you are asking for trouble. If you're having problems step through your code on paper to make sure it creates the memory map shown in Figure 1 on the next page.

Although the block of memory allocated by any call to a standard memory allocation function is guaranteed to be internally contiguous, the blocks allocated by multiple calls cannot be assumed to be contiguous with each other. Such is the case with the multiple memory blocks allocated by the

original Create2D function illustrated in note 14.4B. If these blocks are not contiguous it prevents such arrays from being accessed linearly, which is a significant limitation in some applications. In addition, the fact that the original Create2D function must do multiple dynamic memory allocations makes it inefficient and necessitates a custom Free2D function. These limitations can be overcome if Create2D instead pre-calculates the total amount of memory needed for everything and allocates it all at once. The main disadvantage of this approach is that data alignment problems are possible when multiple data types are mixed. Although this potential issue can be solved with some added complexity, simply ignore it for this exercise.

Your version of Create2D must create a pointer array like the original version except that it must get all needed memory at once. Figure 1, below is a memory map of how the result should look for a 2-by-3 array after your version completes. Compare this with the memory map in Figure 2, produced by the original Create2D function. Notice that both employ the same basic concepts but the new version places everything in one contiguous block of memory rather than in multiple, possibly non-contiguous blocks:

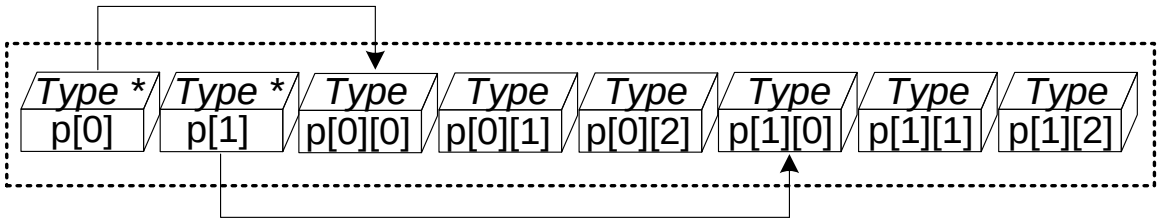


Figure 1 – Memory Map from Your Rewritten Create2D Function for a 2x3 Array

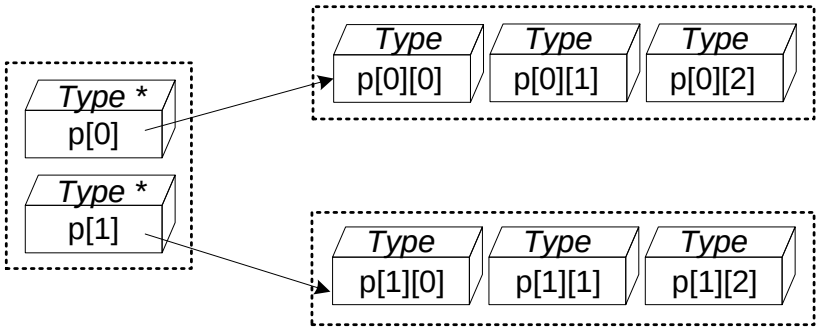


Figure 2 – Memory Map from the Original Create2D Function for a 2x3 Array

As in the original version, you must explicitly initialize each pointer to point to the first element of the corresponding sub-array. Your code (but not my driver file code) must work with any arbitrary data type represented by Type.

Exercise 3 *(5 points – Diagram only – No program required)*

Create the state diagram described below and place it in a PDF file named **C2A5E3_StateDiagram.pdf**. Although using Word, Visio, etc. to create such an illustration and the required PDF file is easy, you may instead do it the hard way by drawing it by hand and scanning it in if you wish as long as it is neat and easily readable.

In standard C and C++ floating literals may be written in either scientific or standard notation and their data types can be any of **float**, **double**, or **long double**, as determined by their suffix or lack thereof. During the parsing phase of program compilation the source code is broken up into separate tokens, some of which may be floating literals.

Produce a State Machine Diagram that analyzes the contents of any string of characters. If the entire string forms a complete floating literal the machine will exit with an indication of its data type (but not its value). If not, the machine will exit with an indication that the string does not represent a floating literal at all. Design your machine such that:

1. it obtains the next character from the string upon entry into every state;
2. it never looks back at previous characters or ahead at future characters and does not rely upon any information other than the character just obtained and the current state. In programming terms this means that the only two variables allowed are the current state variable and the current character variable.

Submitting your solution

Send your PDF file to the Assignment Checker with the subject line **C2A5E3_*ID***, where ***ID*** is your 9-character UCSD student ID.

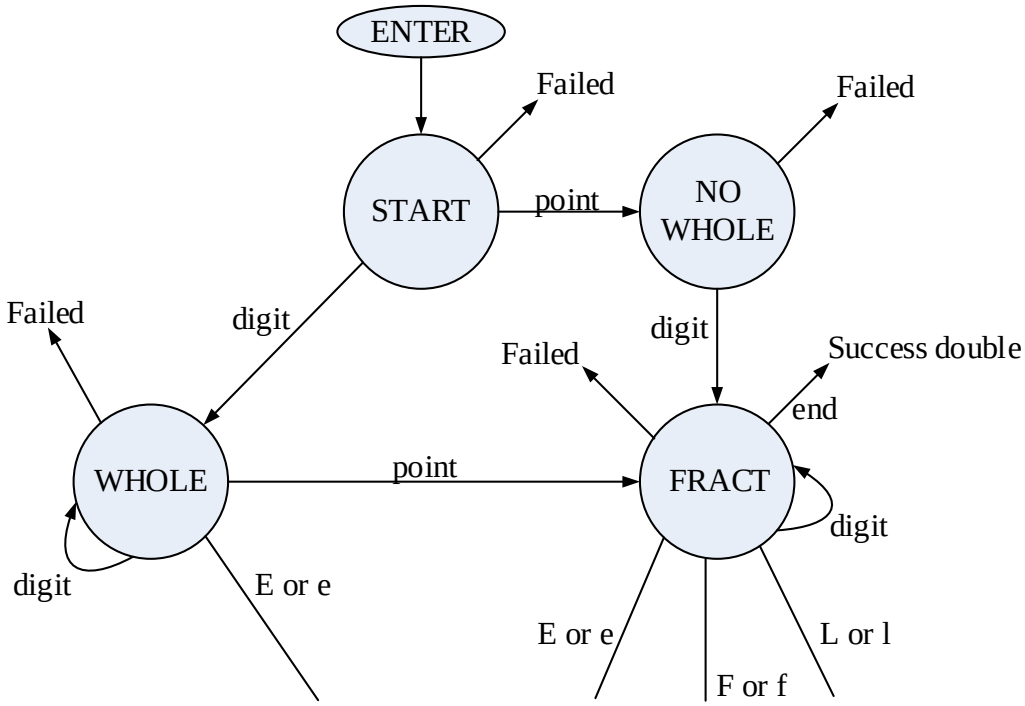
See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

1. 9 and only 9 states are required; your diagram is not correct if it has more or less than 9.
2. The entry and exit points do not count as states.
3. Choose meaningful state names that can be used directly as state identifiers in C/C++ code.
4. Before starting you must have a good understanding of the possible floating literal forms. The language standards use the following recursive adaptation of BNF (Backus Naur Form) to describe them. Note that floating literals may never start with a sign.

floating literal: fractional-constant exponent-part$_{opt}$ floating-suffix$_{opt}$ digit-sequence exponent-part floating-suffix$_{opt}$ fractional-constant: digit-sequence$_{opt}$ **.** digit-sequence digit-sequence **.** exponent-part: **e** sign$_{opt}$ digit-sequence **E** sign$_{opt}$ digit-sequence	sign: one of **+ -** digit-sequence: digit digit-sequence digit floating-suffix: one of **f l F L** digit: one of **0 1 2 3 4 5 6 7 8 9**

5. The first part of the expected state diagram is provided on the next page and shows 4 states and their transitions to other states (which you must design). If the first part of your state diagram is not functionally the same it is wrong. This diagram assumes:
   a. The next character is available as each state is entered;
   b. "Failed" indicates an exit where the string was not a floating literal;
   c. "Success ..." indicates an exit where the string was a floating literal of the specified type;
   d. "end" means the end of the string was reached.


ENTER
Failed
Failed
START
point
NO
WHOLE
Failed
digit
digit
Failed
Success double
end
WHOLE
point
FRACT
digit
digit
E or e
E or e
F or f
L or l

Exercise 4 *(5 points – C++ Program)*

Exclude any existing source code files that may already be in your IDE project and add two new ones, naming them **C2A5E4_OpenFile.cpp** and **C2A5E4_DetectFloats.cpp**. Also add instructor-supplied source code files **C2A5E4_StatusCode-Driver.h** and **C2A5E4_main-Driver.cpp**. Do not write a `main` function! `main` already exists in the instructor-supplied implementation file and it will use the code you write.

Regarding data type `StatusCode`, which is used in this exercise…

`StatusCode` is an enumeration type consisting of members `NOTFLOATING`, `TYPE_FLOAT`, `TYPE_DOUBLE`, and `TYPE_LDOUBLE`. It is defined in instructor-supplied header file

C2A5E4_StatusCode-Driver.h

Any file that uses this enumeration type must include this header file using `#include`.

File **C2A5E4_OpenFile.cpp** must contain a function named `OpenFile`.

`OpenFile` syntax:

```
void OpenFile(const char *fileName, ifstream &inFile);
```

Parameters:

`fileName` – a pointer to the name of a file to be opened

`inFile` – a reference to the `ifstream` object to be used to open the file

Synopsis:

Opens the file named in `fileName` in the read-only text mode using the `inFile` object. If the open fails an error message is output to `cerr` and the program is terminated with an error exit code. The error message must mention the name of the failing file.

Return:

void if the open succeeds; otherwise, the function does not return.

File **C2A5E4_DetectFloats.cpp** must contain a function named `DetectFloats`.

DetectFloats syntax:

```
StatusCode DetectFloats(const char *chPtr);
```

Parameters:

`chPtr` – a pointer to the first character of a string to be analyzed

Synopsis:

Analyzes the string in `chPtr` and determines if it represents a syntactically legal floating literal, and if so, its data type (but not its value).

Return:

one of the following `StatusCode` enumerations representing the result of the string analysis:

`NOTFLOATING`, `TYPE_FLOAT`, `TYPE_DOUBLE`, or `TYPE_LDOUBLE`

DetectFloats must:

1. implement the state machine you diagrammed in the previous exercise exactly, using only one variable other than formal parameter `chPtr`. Using additional variables is not permitted.
2. make all transition decisions based only upon the current character in the string and the current state; storing or looking backward or forward at characters or states is not permitted.
3. be tested with instructor-supplied data file **TestFile5.txt**, which must be placed in the program's "working directory". Do not assume that your program works correctly based strictly upon the results with this file, however. The test strings it contains do not represent all possible character combinations and your program could parse them correctly while still containing one or more significant bugs.

Submitting your solution

Send all four source code files to the Assignment Checker with the subject line **C2A5E4_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

Each state represents a new character to be examined. When you are ready to return out of the state machine function don't go to another state but instead immediately return an appropriate status value. Don't use a separate variable to indicate a potential type **float** or type **long double** string. Instead, use different states to differentiate these findings.

Get a Consolidated Assignment Report (optional)

If you would like to receive a consolidated report containing the results of the most recent version of each exercise submitted for this assignment, send an empty email to the assignment checker with the subject line **C2A5_*ID***, where ***ID*** is your 9-character UCSD student ID. Inspect the report carefully since it is what I will be grading. You may resubmit exercises and report requests as many times as you wish before the assignment deadline.