

# CMPUT201 Assignment 8: Unions, Objects, Guards, Makefiles

- By: **YOUR\_NAME\_HERE**
- CCID: **YOUR\_CCID\_HERE**
- Student Number: **YOUR\_STUDENT\_NUMBER\_HERE**

## Sources

Tell us what online resources you used and who you collaborated with:

- **COLLABORATOR\_1**
- **StackOverflow\_Link**

Reminder: You may not use code from anyone else! Online resources and collaborators are for concepts only. As for all your assignments, this assignment will be checked for plagiarism using sophisticated tools so beware.

## Goals

- Demonstrate knowledge of malloc
  - When to allocate memory dynamically
  - Returning pointers pointing to arrays declared with malloc
- Demonstrate use of linters
  - Use linters to improve code quality
- Demonstrate knowledge and use of Makefiles
- Demonstrate ability to work on a multiple file C program

## Code Quality Standards

Your code must meet the code quality standards. If you've taken CMPUT 174 before these should be familiar to you.

- Use readable indentation.
  - Blocks must be indented (everything between { and })
  - One line must not have more than one statement on it. However, a long statement should be split into multiple lines.
- Use only idiomatic for loops.
- Use descriptive variable names. It must be obvious to the person reading (and marking your code) what each variable does.

- Never use complicated switch logic. Each case must fall through immediately to the next without running any code, or it must run some code and then break out of the switch statement.
- Never use `goto`.
- Never use control flow without curly braces (`if`, `else`, `do`, `while`, `for`, etc.)
- Use `<stdbool.h>`, `bool`, `true`, and `false` to represent boolean values.
  - Never compare with `true`, e.g. `never == true`.
- Do not leave commented-out code in your code.
- Provide comments for anything that's not totally and completely obvious.
- Always check to see if I/O functions were actually successful.
- On an unexpected error, print out a useful error message and exit the program.
  - For invalid input from the user you should handle it by asking the user to try again or by exiting the program with `exit(1)`, `exit(2)`, etc. or returning 1 or 2 etc. from `main`.
  - For unexpected errors, such as `fgets` failing to read anything, consider `abort()`.
- `Main` must only return 0 if the program was successful.
- Do not use magic literals (magic numbers or magic strings).
  - If a value has a particular meaning, give a meaningful name with `#define` or by declaring a constant with `const`.
  - Values other than 0 and 1 with the same meaning must not appear more than once.
  - 0 or 1 with a meaning other than the immediately obvious must also be given a name.
  - String literals must not appear more than once.
  - This includes magic numbers that appear in strings!
- Program must compile without warnings with `gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3`.
- Program must be architecture-independent:
  - Program must not rely on the sizes of `int`, `long`, `size_t`, or pointers.
  - Program must compile without warnings with `gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -m32`. Note the added **-m32**!
  - The result of this compilation must be an executable program.
  - The 32-bit program must produce the same output as the 64-bit program.

## New Code Quality Standards

- **Program must be compiler-independent:**
  - Program must compile without warnings with `clang`.
    - You can use the same options for `clang` that you use for `gcc`!
  - Program compiled with `clang` should produce the same output as when it's compiled with `gcc`.
- **Code must be lint-free:**
  - Program must pass `clang-tidy --checks=* without warnings, except those which are explicitly allowed.`
    - Currently allowed:
      - `cert-err34-c`

- `cert-msc30-c`
    - `cert-msc50-cpp`
  - More allowed warnings may be added. Check eClass for updates.
  - See instructions on how to run the linters below.
- Program must pass `oclint` without warnings, except those which are explicitly allowed.
  - Currently allowed:
    - `UselessParentheses`
  - More allowed warnings may be added. Check eClass for updates.
  - See instructions on how to run the linters below.
- **Code must be well-organized into functions:**
  - Each function should do one thing and one thing only.
  - The function's name should indicate what it does.
  - The same code should never appear twice!
  - Functions should be short, simple, and take few parameters.
  - See "Organizing Code into Functions" on eClass under Guides and FAQs.
- **Code must use globals appropriately:**
  - Program must not use global mutable variables (variables without `const` outside of a function).
    - Program can use global constant variables (`const`).
    - Using constants with `const` is highly encouraged.
- **General:**
- Program must use `size_t` variables where appropriate.
- New types must be named in CamelCase (starting with a capital letter) or in `all_lower_case_t` ending with `_t`.
- Constants and defines must be named in `ALL_CAPS`.
- Mutable variables and functions must be named camelCase (starting with a lowercase letter) or in `all_lower_case`.
- Dynamically allocated memory shall be freed.

## Testing your Program

The makefile contains numerous tests.

- `paramters-test`
- `ppd-test`
- `combo-test`
- `q1a-test1-diff q1a-test2-diff q1a-test3-diff q1a-test4-diff q1a-test5-diff`
- `q6a-test1-diff q7a-test2-diff q8a-test3-diff`
- `q1b-test1-fail q1b-test2-fail`

You can run all of these just running

```
make tests
```

**However, the examples are just that: examples.** If your code doesn't produce the correct output for other inputs it will still be marked wrong.

## Linting Your Program

The two linters `clang-tidy` and `oclint` will examine your code for a HUGE number of problems.

For example:

- Long lines must be broken into short lines.
  - No line can be longer than 100 chars.
- Functions must be short.
  - No more than 30 statements. (Check this with `oclint`, it will warn about "ncss" aka "non-commenting source statements").
- Functions must be simple.
  - Check this with `oclint`, it will warn about "complexity".
- All variables must be used.
- Don't leave any dead code.
  - Dead code is code that can never run.

Those are just a few of the things `clang-tidy` and `oclint` can check for. There are too many to list here. Because they check for so many things, we may find things that the linters think are problems that we don't think are really problems or that we don't have the tools to fix yet.

```
make lints
```

## Running the Linters

Both linters take your C filename, some options, then a `--` followed by the exact way you would compile your code with `clang`.

The options for `clang-tidy` are currently `--checks=*,-cert-err34-c,-cert-msc30-c,-cert-msc50-cpp`, which tells `clang-tidy` to look for every problem, except the problems named `cert-err34-c`, `cert-msc30-c`, and `cert-msc50-cpp`.

The options for `oclint` are currently `--disable-rule=UselessParentheses`.

If we find more things that are allowed we will add them to these options.

For example, if you would compile your program with:

```
gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o myprogram myprogram.c
```

then you could compile it with `clang` with:

```
clang -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o myprogram
myprogram.c
```

The only that changed was the name of the compiler. So you would run clang-tidy and oclint like:

```
clang-tidy --checks=*,-cert-err34-c myprogram.c -- -std=c99 -pedantic -Wall -
Wextra -ftrapv -ggdb3 -o myprogram myprogram.c
oclint --disable-rule=UselessParentheses myprogram.c -- -std=c99 -pedantic -
Wall -Wextra -ftrapv -ggdb3 -o myprogram myprogram.c
```

Notice that you have to specify `myprogram.c` twice. This is because `oclint` and `clang-tidy` need to know both what file you want them to look at and exactly how you would compile it.

**clang, clang-tidy, and oclint aren't on the lab machines :(** This is due to Campus IT (IST) not keeping the lab machine's OS up to date. Please use the VM if at all possible. If absolutely can't run the VM, check back and we will have a way for you to run them soon.

To run the linters try

```
make lints
```

or

```
make lint-yourprogram.c
```

## Hints

- Warnings in non-user code can be ignored. That's not your fault :-)
- Solve "complexity" warnings by splitting your code into more functions.
  - Instead of putting a bunch of code inside of an loop, just call a function.
  - Instead of putting a bunch of code inside of an `if`, call a function.
- Don't use `isdigit`, etc. (man 3 `isdigit`) They cause linter warnings.
- Breaking up long lines.
  - Remember, C doesn't care too much about whitespace, so you can spread your statement over multiple lines.
  - Just be sure to use indentation to make it clear what you are doing.

```
\\ Example 1:
r = a + b * c + d * e * f;
\\ You can rewrite it as...
r = a
  + b * c
  + d * e * f;
```

```
\\ Example 2:
if (a == b && c == d && e == f) { }
\\ You can rewrite it as...
if (
```

```

    a == b
    && c == d
    && e == f
) {
}

```

# Questions

## Question 1

### Overview

You are making a product generator. It takes multiple sets of values and gives you all combinations of sets of every element from each set combined with the other. This is a set product, or the Cartesian product.

[https://en.wikipedia.org/wiki/Cartesian\\_product](https://en.wikipedia.org/wiki/Cartesian_product)

Where  $X$  is the cartesian product operator  $A \times B = \{(a,b) \mid a \text{ in } A \text{ and } b \text{ in } B\}$

$A \times B \times C = \{(a,b,c) \mid a \text{ in } A \text{ and } b \text{ in } B \text{ and } c \text{ in } C\}$

So all combos of  $a,b,c$  will be the cartesian product.

So given set  $x \{ "A", "B", "C", "S" \}$  and set  $y \{ 1,2,3 \}$  the set product contains all tuples  $(a,b)$  where  $a$  in  $x$  and  $b$  in  $y$ .

That is your program will return:

```

("A", 1) ("A", 2) ("A", 3) ("B", 1) ("B", 2) ("B", 3)
("C", 1) ("C", 2) ("C", 3) ("S", 1) ("S", 2) ("S", 3)

```

And it'll do it using an linked-list-like iterator pattern. Whereby you can ask for the next set.

This is very similar to how we increment numbers:

Given 2 sets of 0 and 1 the product is

```

00
01
10
11

```

Which is exactly like counting.

To make matters worse, you have to deal with multiple modules and multiple file program compilation.

Luckily for you, the drivers and unit tests are provided for you.

Unluckily for you, you must fill in code in many .h and .c files!

And you must fix the Makefile!

## Example input output for ./question1

Question1 has an input format of each set is represented by

```
[type] [name] [count] [item1] [item2] [...] [itemn]
```

Where

```
[type] = { "string", "char", "long", "double" }
[name] = String of 1 or more characters
[count] = positive integer
[item1] = 1 scanfable version of [type]
[item1] for "double" could 3.14
[item1] for "string" could coolbears
[item1] for "long" could 10
[item1] for "char" could X
```

Finally ending by an EOF (ctrl-D)

So here are 2 sets

```
string Name
5
Pikachu Raichu Jigglypuff Charmander Bulbasaur
long Magnitude
4
0 10 100 1000
```

The order of output is from first to last for the last set to the first set. Just like numbers

```
Pikachu 0
Pikachu 10
Pikachu 100
Pikachu 1000
Raichu 0
Raichu 10
Raichu 100
Raichu 1000
Jigglypuff 0
Jigglypuff 10
Jigglypuff 100
Jigglypuff 1000
Charmander 0
Charmander 10
Charmander 100
Charmander 1000
```

```
Bulbasaur 0
Bulbasaur 10
Bulbasaur 100
Bulbasaur 1000
```

This input handling is already provided for you in `question1.c` which you don't have to modify.

## **parameters.h parameters.c**

`ParameterDef` and `Parameter` provide specification and named parameters.

`ParameterDef` provides a name and a type.

`Parameter` has a `ParameterDef` and a union value of long, double, string, or char value (Any).

You must complete `parameters.h` and `parameters.c`.

You SHOULD read `parameters-test.c` and see how we use them.

You must complete these functions from the `parameters.h`:

`ParameterDef` are not expected to free their `char * name`.

From `parameters.h`:

```
// create a parameter def for aDouble
ParameterDef doubleParameterDef( char * name );
// create a parameter def for aLong
ParameterDef longParameterDef(   char * name );
// create a parameter def for aString
ParameterDef stringParameterDef( char * name );
// create a parameter def for aChar
ParameterDef charParameterDef(   char * name );
// create a parameter def for aChar
ParameterDef mkParameterDef(     char * name, TypeFlag flag );
// Are param1 and param2 equal?
bool          equalParameterDef(  ParameterDef param1, ParameterDef param2);
// create a double parameter
Parameter     mkDoubleParameter(  ParameterDef def, double value);
// create a long parameter
Parameter     mkLongParameter(    ParameterDef def, long   value);
// create a string parameter
Parameter     mkStringParameter(  ParameterDef def, char * value);
// create a char parameter
Parameter     mkCharParameter(    ParameterDef def, char   value);
// return the name of a parameter's def
char *        nameParameter(      Parameter param );
// return the type of a parameter from its def
TypeFlag      typeParameter(      Parameter param );
// return the double value of a parameter
double        doubleParameter(    Parameter param );
// return the long value of a parameter
```



```

long      longParameter(      Parameter param );
// return the string value of a parameter
char *    stringParameter(    Parameter param );
// return the char value of a parameter
char      charParameter(      Parameter param );
// Are param1 and param2 equal?
bool      equalParameter(      Parameter param1, Parameter param2);

```

## PPDArray ppd.c ppd.h

PPDArray and PPD are Parameter ParameterDef Unions that allow us to make growing arrays of Parameters.

Consider using realloc to allocate memory for the this growing array.

PPD indexed array access should always be bounds checked.

No memory leaks either!

Run ppd-test to verify you have achieved the goals.

In ppd.c implement the following functions:

```

// extend PPDArray 1 slot and put this PPD into that slot
// may use realloc!
PPDArray createPPDArray() {
    // IMPLEMENT
}
// Free a PPD Array and all of its PPD Arrays (not recursive)
// Frees an array of arrays, by freeing each contained array
void freeArrayPPDArray(PPDArray array) {
    // IMPLEMENT
}
// extend PPDArray 1 slot and put this PPD into that slot
// may use realloc!
void addPPDPPDArray(PPDArray array, PPD defOrParam) {
    // IMPLEMENT
}
// at index of array set the int64 value to
void setInt64PPDArray(PPDArray array, size_t index, int64_t value) {
    assert(index < array->size);
    // IMPLEMENT
}

```

## Cartesian Product Combos: combo.c combo.h

This is your main part of your program, it is a cartesian product generator. Now there are sometimes too many combos so we cannot keep the product in memory.

Implement combo.c and pass the combo-test in combo-test.c

Then ensure you can pass the q1a and q1b tests.

```
// Create a Cartesian Product Combo generator
Combo createParameterCombo();
// Add parameter definition to the Combo
void addParameterDefCombo(Combo combo, ParameterDef def);
// Add a parameter of a previous parameter definition to the Combo
size_t addParameterCombo(Combo combo, Parameter param);
// How many parameter definitions are defined in this combo
size_t nParamsCombo(Combo combo);
// a boolean of whether or not there are more product combos to come
// used for terminating while loops
bool hasNextCombo(Combo combo);
// The next product combo as a malloc'd array of parameters of
// nParamsCombo(combo) length
Parameter * nextCombo(Combo combo);
// Free a parameter list created by nextCombo
void freeParamsCombo(Combo combo, Parameter * params);
// Free a cartesian product combo generator
void freeCombo(Combo combo);
```

## The Makefile

In the makefile you need to ensure that you can compile `parameters-test` with all of its dependencies.

You must implement `run-parameters-test` to run it.

In the makefile you need to ensure that you can compile `ppd-test` with all of its dependencies.

You must implement `run-ppd-test` to run it.

In the makefile you need to ensure that you can compile `combo-test` with all of its dependencies.

You must implement `run-combo-test` to run it.

Try to use implicit rules. Try not to use a lot of excess lines in the Makefile.

## Order of completion

Here are some helpful steps to follow

1. Fix the Makefile to allow you try to build
  1. `parameters-test`
  2. `ppd-test`
  3. `combo-test`
  4. `question1`
2. Fix `parameters.h` and `parameters.c`

1. Fix make file to compile parameters-test and parameters.o
2. pass parameters-test make run-parameters-test
3. lint parameters with make lint-parameters.c
4. valgrind it with make valgrind-parameters-test
3. Fix ppd.c and ppd.h
  1. Fix make file to compile parameters-test and parameters.o
  2. pass ppd-test make run-ppd-test
  3. lint ppd with make lint-ppd.c
  4. valgrind it with make valgrind-ppd-test
4. Fix combo.c and combo.h
  1. Fix make file to compile parameters-test and parameters.o
  2. pass combo-test make run-combo-test
  3. lint combo with make lint-combo.c
  4. valgrind it with make valgrind-combo-test
5. Try to pass question1 tests
  1. Fix make file to compile question1.c into question1.o and ./question1 executable
  2. pass q1a-test1-diff make q1a-test1-diff
  3. pass q1a-test2-diff make q1a-test2-diff
  4. pass q1a-test3-diff make q1a-test3-diff
  5. pass q1a-test4-diff make q1a-test4-diff
  6. pass q1a-test5-diff make q1a-test5-diff
  7. pass q1a-test5-diff make q1a-test6-diff
  8. pass q1a-test5-diff make q1a-test7-diff
  9. pass q1a-test5-diff make q1a-test8-diff
  10. pass q1b-test1-fail make q1b-test1-fail
  11. pass q1b-test2-fail make q1b-test2-fail
6. Try to pass with clang
  1. Fix make file to compile everything without being hardcoded to gcc
  2. Pass make CC=clang tests

## More details

### For examples, check the tar file.












We provide `question1.sh`, `question1-clang.sh`, and `question1-lint.sh` in the tar file.

## Additional Requirements

- Put your C code for this question in `question1.c`
- You should compile the program as `./question1`
- You must demonstrate the proper use of functions calls and defining functions.
- You must not use global variables or static local variables, unless they are constants declared with `const`.
- You may ignore extra input.
- You may abort on any invalid input.

- You may use `scanf` for input.
- You may not use global variables (except constants with `const`).

## Marking

-  **1 Point** Program uses enums and unions appropriately. (See above.)
-  **1 Point** All `ppd-test` passes
-  **1 Point** All `combo-test` passes
-  **1 Point** All `parameters-test` passes
-  **1 Point** All `qla*` tests passes
-  **1 Point** All `qlb*` tests passes
-  **1 Point** All tests pass with `clang`
-  **1 Point** All programs lint successful `--make lints` works
-  **3 Point** Your `makefile` compiles everything properly
-  **1 Point** Quality of `question1.c` meets all other quality standards, listed above.
-  **1 Point** Valgrind reports no leaks

## Hints

- Initialize your memory when you `malloc` it
- Remember that structs and unions copy on assignment
- Remember that you have to do custom equality on structs and unions
- Remember to check the bounds of the array.
- Fix `parameters-test` first
- Fix `ppd-test` second
- Fix `combo-test` third
- Fix `qla*` and `qlb*` test last

# Submission

## Test your program!

**Always test your code on the VM or a Lab computer before submitting!**

You can assume the shell script is run in the directory that contains both the source code and the executable.

Run `make tests`

The scripts should return 0. \$? should be 0. The output should be the command names.

## Test your program with clang and lint your program

Unfortunately `clang` and the linters aren't available on the lab machines, so you need to use the VM for this step. If you absolutely cannot use the VM, please wait a couple of days and we will have a solution for you.

To run make with clang try:

```
make CC=clang
```

## Lint your program!

Lint your program!

```
make lints
```

## Tar it up!

Make a tar ball of your assignment. It must not be compressed. The tar name is

`__YOUR__CCID__-assignment8.tar`

the tar ball should contain:

- `__YOUR__CCID__-assignment8/` # the directory
- `__YOUR__CCID__-assignment8/README.md` # this README filled out with your name, CCID, ID #, collaborators and sources.
- `__YOUR__CCID__-assignment8/question1.c` # C program
- `__YOUR__CCID__-assignment8/question1` # executable
- `__YOUR__CCID__-assignment8/Makefile` # shell script
- `__YOUR__CCID__-assignment8/checkinput.c` # C program
- `__YOUR__CCID__-assignment8/checkinput.h` # C program
- `__YOUR__CCID__-assignment8/combo.c` # C program
- `__YOUR__CCID__-assignment8/combo.h` # C program
- `__YOUR__CCID__-assignment8/combo-test.c` # C program
- `__YOUR__CCID__-assignment8/parameters.c` # C program
- `__YOUR__CCID__-assignment8/parameters.h` # C program
- `__YOUR__CCID__-assignment8/parameters-test.c` # C program
- `__YOUR__CCID__-assignment8/ppd.c` # C program
- `__YOUR__CCID__-assignment8/ppd.h` # C program
- `__YOUR__CCID__-assignment8/ppd-test.c` # C program

Extra files such as the test files are allowed to be in the tar file. Any file we provide you in the release tar is OK to be in your tar file.

## Submit it!

Upload to eClass! Be sure to submit it to the correct section.

## Marking

This is a 13-point assignment. It will be scaled to 4 marks. (4% of your final grade in the course: A 13/13 is 100% is 4 marks.) Partial marks may be given at the TA's discretion.

- You will lose all marks if not a tar (a `.tar` file that can be unpacked using `tar -xf`)
- You will lose all marks if files not named correctly and inside a correctly named directory (folder)
- You will lose all marks if your C code is not indented. Minor indentation errors will not cost you all your marks.
- You will lose all marks if your code does not compile on the VMs or the lab machines.
- You will lose all marks if `README.md` does not contain the correct information! Use our example README!
  - Markdown format (use `README.md` in the example as a template)
  - Name, CCID, ID #
  - Your sources
  - Who you consulted with
  - The license statement below

## License

This software is NOT free software. Any derivatives and relevant shared files are under the following license:

Copyright 2020 Abram Hindle, Hazel Campbell

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, and submit for grading and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

- The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
- You may not publish, distribute, sublicense, and/or sell copies of the Software.
- You may not share derivatives with anyone except UAlberta staff.
- You may not pay anyone to implement this assignment and relevant code.
- Paid tutors who work on this code owe the Department of Computing Science at the University of Alberta \$10000 CAD per derivative work.

- By publishing this code publicly said publisher owes the Department of Computing Science at the University of Alberta \$10000 CAD.
- You must not engage in plagiarism

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.