Information Processing 2 — Week 08 exercises

Complete the first three exercises in-class on your own laptop. Ask an instructor to check your work during the class as soon as you finish answering each question. A **late penalty of 50%** will be applied to answers checked after class.

Complete the remaining exercises in-class if possible, or before the start of the next exercise class at latest. Exercises checked **more than one week** after the class will have a **50% late penalty** applied.

Declarations, header files, and multiple source files

Functions and variables can be declared before they are defined. This allows them to be called or used before they are defined. Using declarations you can call functions before they are defined (later in the same file) or call functions that are defined in a different source entire.

1 Forward declarations [2 points]

Download the file `08-calculator.c` from Teams. It contains the following function and variable definitions, in this order:

```
int saved = 0;
int getch(void) { ... }
void ungetch(int c) { ... }

int getop(char line[], int limit) { ... }

double stack[32];
int depth = 0;
int stackDepth(void) { ... }
void push(double d) { ... }
double pop(void) { ... }

double atod(char line[]) { ... }

int main() { ... }
```

Move the definition of `main()` to near the start of the program (just before the definition of `saved`). Verify that program no longer compiles properly.

Just before `main()`, add declarations for the functions `getop`, `stackDepth`, `pop`, `push`, and `atod` that are called from `main()`. Verify that program compiles properly again.

Test the program as follows, and verify that the output is correct.

```
$ echo "3 4 * 5 6 * +" | ./08-01-calculator
42
```

▷▷ Ask an instructor to check and record your work.

2 A header file for the entire program [2 points]

Write a single header file `calculator.h` that can be included at the start of the program to declare all the functions needed by `main()`. In the same file, define the symbol `NUMBER`.

Replace the definition of `NUMBER` and the forward function declarations with a single line:

```
#include "calculator.h"
```

Compile the program and verify that it still works.

▷▷ Ask an instructor to check and record your work.

3 Combining multiple source files [2 points]

Split your program into five separate files, as follows:

`getop.c`	contains the functions `getch()`, `ungetch()`, and `getop()`,
`stack.c`	contains the functions `push()`, `pop()`, and `stackdepth()`
`atod.c`	contains the function `atod()`, and
`main.c`	contains `main()`.

Add `#include "calculator.h"` at the top of the three new files. Include any other necessary header files (for example, `getop.c` will need to include <`stdio.h`>).

▷▷ Ask an instructor to check and record your work.

4 Dedicated header file for each source file [1 point]

The functions `atod`, `push`, and `pop` could be useful in may more programs than just the calculator.

Write header files called `atod.h` that declares `atod()`,
`stack.h` that declares `stackDepth()`, `push()`, and `pop()`,
and `getop.h` that declares `getop()`.

Include each header file in its corresponding `.c` file (instead of `calculator.h`). Include all three header files in `main.c` file (instead of `calculator.h`).

Verify that your program still compiles and works.

▷▷ Ask an instructor to check and record your work.

5 File-scope `static` variables and functions [1 point]

At the end of `main()` add '`printf("%i\n", depth);`' just before the `return` statement. This will print the depth of the stack before the program terminates. Verify that it works.

The variable `depth` in the file `stack.c` should really only be accessible to the functions in that same file. Add the keyword `static` to the front of its declaration. Verify that the program no longer compiles, because `main()` can no longer 'see' `depth`.

Remove the final `printf` from `main()`, compile the program, and make sure it works.

There are other functions and variables that can be made `static`, because they are only used by functions in the same source file. Identify as many as you can and make them `static`. Compile the program and verify that it still works.

Hints: `stack.c` contains one more thing that can be made `static`, and `getop.c` contains three more.

▷▷ Ask an instructor to check and record your work.

Macros

Macros can be used to implement your own short-hand notations for common activities. Since macros can represent arbitrary text, you can use them to generate any program text including control constructs and complex behaviour in blocks (compound statements).

6 Making new control constructs with macros [1 point]

The C programming language has a `while` loop (and a `do-while` loop) but it does not have an `until` loop (or a `do-until` loop). (These 'until' loops should behave like the 'while' loops, except that they repeat the body of the loop *until* the condition becomes *true*.)

Define a macro

```
#define until(X) /* your definition here */
```

to implement the `until` (and `do-until`) loop keyword. Check that your definition works using the following program.

```
int main()
{
    int i = 0;

    until (i++ > 8) printf("%d ", i);

    do {
        printf("%d ", i);
        --i;
    } until (i == 0);

    printf("\n");

    return 0;
}
```

Verify that the output from the program is: `1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1`

▷▷ Ask an instructor to check and record your work.

7 Swapping two variables using a macro with a block [1 point]

You can swap the contents of two floating-point `double` variables `p` and `q` like this:

```
{ double _tmp = p;  p = q;  q = _tmp; }
```

Write a macro

```
#define SWAP(T, X, Y)   /* your definition here */
```

that swaps two variables `X` and `Y`, both of which are of type `T`. Hints:

1. You will have to make a temporary variable to hold X while you assign Y to it.
2. Use a block (in the macro definition) to limit the scope of your temporary variable so that it is only in scope while swapping the two variables. (The block will also help your three statements, needed to swap the variables, to behave like a single statement.)

Test your macro using the following program:

```
int main()
{
    int i = 42, j = 666;

    printf("%d %d\n", i, j);
    SWAP(int, i, j);
    printf("%d %d\n", i, j);

    double d = 4.2, e = 6.66;

    printf("%f %f\n", d, e);
    SWAP(double, d, e);
    printf("%f %f\n", d, e);

    return 0;
}
```

Verify that the output is:

```
42 666
666 42
4.200000 6.660000
6.660000 4.200000
```

▷▷ Ask an instructor to check and record your work.

Challenge

Making macro-based short-hand notations completely robust can be difficult. One difficulty arises because of the syntax of `if` statements, which need exactly one statement between `if` and `else`. Since a block is an entire statement, you cannot write an extra semicolon after a block that comes between `if` and `else`. When using a macro to generate the statement controlled by `if` you therefore need to know the implementation of the macro to know whether a semicolon should be placed before an associated `else`.

8 Make SWAP robust [1 bonus point]

The obvious definition of SWAP using a block works fine in the example program above but it does not work in more complicated situations. For example, it causes a syntax error when used in the following code:

```
int main()
{
    int i = 42, j = 666, k = 123;

    if (i < j) SWAP(int, i, j);
    else       SWAP(int, i, k);

    printf("%d %d %d\n", i, j, k);

    return 0;
}
```

Figure out why the `SWAP` macro causes the error in this context and then think of a way to fix it. Use the program fragment shown above to test your macro. The program should print '`666 42 123`'. Change the condition to `i > j` and verify the output is then '`123 666 42`'.

▷▷ Ask an instructor to check and record your work.