Recitation 2: C Primer and Profiling

(Last Updated: September 12, 2014)

This recitation provides a quick C primer and introduces the perf profiling tool.

1 Getting started

We recommend that you work on the cloud machines. You can access them using ssh:

```
$ ssh <username>@cloud.csail.mit.edu
```

To get a local copy of the repository for your work, you need to use git to clone it:

```
$ git clone /afs/csail.mit.edu/proj/courses/6.172/student-repos/fa14/recitations/\>recitation2/<username>.git recitation2
```

You can also get the recitation code from GitHub:

```
$ git clone https://github.com/mit6172/recitation2 recitation2
```

2 Why use C?

- Simple: No complicated object-oriented abstractions like Java/C++.
- Powerful: Offers direct access to memory (but does not offer protection in accessing memory).
- Fast: No overhead of a runtime or JIT compiler, and no behind-the-screen runtime features (like garbage collection) that use machine resources.
- Preferred low level language: Most low level software (e.g. device drivers, operating system kernels), that need direct interaction with the machine are written in C.

3 Preprocessing

The C preprocessor modifies source code before it is passed to the compilation phase. Specifically, it performs string substitution of `#define` macros, conditionally omits sections of code, processes `#include` directives to import entire files’ worth of code, and strips comments from code.

As an example, consider the code in `preprocess.c`, which is replicated here:
// All occurrences of ONE will be replaced by 1.
#define ONE 1

// Macros can also behave similar to inline functions.
// Note that parentheses around arguments are required to preserve order of
// operations. Otherwise, you can introduce bugs when substitution happens.
#define MIN(a,b) ((a) < (b) ? (a) : (b))

int c = ONE, d = ONE + 5;
int e = MIN(c,d);

#ifndef NDEBUG
// This code will be compiled only when
// the macro NDEBUG is not defined.
// Recall that if gcc is passed -DNDEBUG on the command line,
// then NDEBUG will be defined.
if (something) {} #endif

Exercise: Direct gcc to preprocess preprocess.c.

$ gcc -E preprocess.c

The preprocessed code will be output to the console. Now rerun the C preprocessor with the
following command:

$ gcc -E -DNDEBUG preprocess.c

You will notice that the if statement won’t appear in the preprocessor output.

4 Data types and their sizes

C supports a variety of primitives types, including the following:

short s;
unsigned int i;
long l;
char c;
float f;
double d;

The use of long is considered suspect by many programmers because its size is architecture-
dependent. It is better to explicitly write code with explicit data types instead:

#include <stdint.h>

uint64_t unsigned_64_bit_int;
int16_t signed_16_bit_int;

You can define more complex data types by composing primitives types into a struct, as
follows:
typedef struct {
    int id;
    int year;
} student;

student you;
you.id = 12345;
you.year = 3;

student *me = (student *)malloc(sizeof(student));
me->id = 12346;
me->year = 4;

Exercise: Edit sizes.c to print the sizes of each of the following types: int, short, long, char, float, double, unsigned int, long long, uint8_t, uint16_t, uint32_t, uint64_t, and student. To compile and run this code, use the following command:

$ make sizes && ./sizes

You may find it useful to avoid creating repetitive code to define a macro and call it for each of the types, rather than copying and pasting code.

5 Argument passing

Formal arguments of a function and the actual arguments of the function call are separate variables. When a function is called, formal arguments are initialized with the corresponding actual arguments. In other words, the arguments are passed by value.

For instance, consider this code that swaps two integers using the bit trick from lecture. What goes wrong with the bit trick here?

```c
void swap(int i, int j) {
    i ^= j;
    j ^= i;
    i ^= j;
}

int main() {
    int k = 1;
    int m = 2;
    swap(k, m);
    // What does this print?
    printf("k = %d, m = %d\n", k, m);
}
```

There are two ways to fix this code. One such way is to change swap() to be a macro, causing the operations to be evaluated in the scope of the macro invocation. Finally, you could change swap() to use pointers, which we’ll ask you to do in the next section.
6 Pointers

Pointers are first-class data types that store addresses into memory. A pointer can store the address of anything in memory, including another pointer. In other words, it is possible to have a pointer to a pointer.

As you will see in Homework 2, arrays behave very similarly to pointers. For example, `int foo[]` and `int *foo` behave very similarly. There are a few gotchas involved with treating pointers and arrays equivalently, however.

Consider the following (buggy) snippet of code from `pointer.c`:

```c
int main(int argc, char * argv) { // What is the data type of argv?
    int i = 5;
    int * pi = &i; // The & operator here gets the address of i and
    // stores it into pi.
    int j = *pi; // The * operator here dereferences pi and stores the
    // value -- 5 -- into j.

    char c [] = "6.172";
    char * pc = c; // Valid assignment: c acts like a pointer to c[0] here.
    char d = *pc;
    printf("char d = %c\n", d); // What does this print?

    // pcp is a pointer to a char pointer, meaning that
    // pcp stores the address of a char pointer
    char ** pcp;
    pcp = argv; // Why is this assignment valid?

    const char * pcc = c; // pcc is a pointer to a char constant
    char const * pcc2 = c; // What is the type of pcc2?

    // For each of the following, why is the assignment:
    *pcc = '7'; // invalid
    pcc = *pcp; // valid
    pcc = argv [0]; // valid

    char * const cp = c; // cp is a const pointer to char
    // For each of the following, why is the assignment:
    cp = *pcp; // invalid
    cp = *argv; // invalid
    *cp = '!'; // valid

    const char * const cpc = c; // cpc is a const pointer to char const.
    // For each of the following, why is the assignment:
    cpc = *pcp; // invalid
    cpc = argv [0]; // invalid
    *cpc = '@'; // invalid
}
```

1For further reading on this, try out after class the challenge at https://blogs.oracle.com/kssplice/entry/the_kssplice_pointer_challenge.
Exercises:

1. Compile `pointer.c` using the following command:

   ```
   $ make pointer
   ```

   You will see compilation errors corresponding to the invalid statements mentioned in the above program. Why are these statements invalid? Comment out those invalid statements and recompile the program. (Do not worry if you see additional warnings about unused variables.)

2. File `swap.c` contains the code to swap two integers from Section 5. Rewrite the `swap()` function using pointers and make appropriate changes in `main()` function so that the values are swapped with a call to `swap()`. Compile the code with `make swap` and run the program with `./swap`. Verify that the results are correct.

7 Perf

Perf is a static profiling tool. It uses sampling to gather data about important hardware and kernel events, such as cache misses, branch misses, page faults and context switches. It also helps in locating performance bottlenecks in a program.

First, we are going to use the `stat` subcommand, which is generally invoked as follows:

```
$ perf stat <program_name> <program_arguments>
```

You can choose specific events, such as `branch-misses`, with the `-e` option. You can see a full list of events by running `perf list`.

Here is an example on how to identify branch misses, clock cycles, and instructions executed by your program on Lanka:

```
$ lexec perf stat -e branches -e branch-misses -e cycles -e instructions
> <program_name> <program_arguments>
```

You can use `perf record` and `perf report` to analyze the performance of your program on Lanka. Here is an example of how to use `perf record` on Lanka:

```
$ lexec perf record -o perf.data <program_name> <program_arguments>
```

You can then view the results of `perf record` using `perf report` as follows:

```
$ perf report -i perf.data --symfs=.debug
```

Note: To run `perf record` on Lanka, we use `lexec`. To view the results locally on a cloud machine, we run `perf report` without `lexec`.

Exercise: File `isort.c` contains an insertion sort routine. Compile the program with

```
$ make isort
```
and then run the `isort` executable to identify branch misses, clock cycles and instructions. Identify the performance bottlenecks in the program. You can read the disassembly of `isort` annotated with performance data using `perf report`.

You can see both the C code and the assembly instructions in the annotated output. The performance counter events are usually associated with the instruction right after or a few instructions below the one that causes extra stalls.

8 Finished!

Commit your changes to your local repository, then verify your work using `verifier.py` and check your code quality by running `clint.py`. If these scripts pass, show your work to a TA or UTA to complete the checkoff for this recitation. If you worked with someone else, you should grant permission to access the repository to your peer.