Recitation 2: C Primer and Perf

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

1 Getting started

We recommend that you work on course machines. You can access it using ssh

% ssh username@cloudN.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/recitation2/username recitation2

2 Overview of C

- Simple and Powerful: No complicated Object-oriented abstractions like Java/C++; Offers direct access to the machine.

- Fast: No overhead of JIT compilation; 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), that need direct interaction with the machine are written in C.

3 Data types et al

C supports a variety of primitives.

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

Higher level data types can be composed from the primitives.

    typedef struct
    {
       int id ;
       int year ;
    } student ;

    student you ;
    you.id = 12345 ;
    you.year = 3 ;
• **Exercise:** Compile and run a simple program `sizes.c` that prints the byte sizes of each of these types. To make and run, use the following command.

```
% make sizes
% ./sizes
```

## 4 Preprocessor

Preprocessor processes macros and brings in headers included in `# include` directives. Preprocessing precedes the actual compilation. Conditional compilation helps avoid compiling certain portions of the code.

```c
#define ONE 1 // All occurrences of ONE will be replaced by 1.
#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.
  if (something) {}
#endif
```

• **Exercise:** Preprocess a simple program `preprocess.c`, which has the above statements, using the following command. It will output the preprocessed code to the console.

```
% icc -E preprocess.c
```

Now preprocess again with the following command.

```
% icc -E -DNDEBUG preprocess.c
```

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

## 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.

```c
void swap(int i, int j)
{
    i ^= j;
    j ^= i;
    i ^= j;
}
```
int main()
{
    int k = 1, m = 2;
    swap(k, m);
    // What would be the value of k and m printed below?
    printf("k = %d, m = %d\n", k, m);
}

6 Pointers

Pointers are data types that can store addresses of variables. For example, consider the following snippet.

int main(int argc, char * argv[]) // what data type is argv?
{
    int i = 5;
    // pi stores the address of variable i;
    int * pi = &i;
    int j = *pi; // j now has value 5.

    char c[] = "6.172";
    // array names are pointers to the first element
    // of the array.
    char * pc = c; // valid assignment
    char d = *pc; // d now has the value '6'.

    printf("char d = %.c\n", d);

    // pointer to a char pointer
    // parray stores the address of a char pointer
    char ** pcp;

    // Is the below assignment valid?
    pcp = argv;

    // Yes. since argv is an array of char pointers and
    // argv points to the first element of the array.
    const char * pcc = c; // pointer to character constant
    char const * pcc2 = c; // what type is this?

    // Are the following assignments valid?
    *pcc = '7'; // No. Pointer is pointing to a char const
    pcc = *pcp; // Yes. *pcp is a char pointer.
    pcc = argv[0]; // Yes. argv[0] is a char *.

    char * const cp = c; // const pointer to character
    // Are the following assignments valid?
    cp = *pcp; // invalid. cannot change the const pointer
    cp = *argv; // invalid. cannot change the const pointer
Handout 8: Recitation 2: C Primer and Perf

*cp = '; //valid.

const char * const cpc = c ; //const pointer to char const.
//Are the following assignments valid?
cpc = *pcp ; //invalid
cpc = argv[0] ; //invalid
*cpc = '@' ; //invalid

}  

Complete the following Exercise:

1. The above program is in pointer.c. Compile the code with the following command.

   % make pointer

   You will see compilation errors corresponding to the invalid statements mentioned in the above program. Try to understand why they are invalid, comment out those invalid statements and recompile the program.

2. swap.c contains the code to swap two integers as was shown in the section on argument passing. 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 the results are correct.

7 Java and C

We shall briefly go over the material in 'C for Java Programmers' by Reid Kleckner. It serves as a useful reference.

8 Perf

Perf is a static profiling tool. It uses sampling to gather data about important h/w and kernel events, such as cache misses, branch misses, page faults and context switches. It also helps in locating performance bottlenecks in a program. The perf stat command helps find branch-misses, cache misses etc. It is invoked as

   % perf stat -e branches -e branch-misses -e cycles -e instructions\  
   program_name program_arguments

   The perf record and perf annotate commands help identify performance bottlenecks in the program. They are invoked as
% perf record -f program_name program_arguments

% perf annotate -l function_name

- **Exercise** 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. Also identify the performance bottlenecks in the program.