C programming
CPS104 - Computer Organization and Programming
Alexandru Duţu
Resources

- Recommended book:
  - “The C Programming Language” by Brian W. Kernighan and Dennis M. Ritchie
  - http://www.howstuffworks.com/c.htm
Why C?

- compiled directly to machine code
- memory management - allocation and deallocation
- explicit access to references, pointers
- is not a Object Oriented Programming language - C has modules
C program

#include <stdio.h>

int main()
{
    printf("Hello world!\n");
    return 0;
}
Making a program

- compilation (gcc)
  - assembly code

- assembling (as)
  - objects (.o or .obj, do not confuse with objects from OOP)
  - libraries (static or dynamic)

- linking (ld)
  - binaries
  - libraries

- gcc is directly producing the binary
compiling

- GNU compiler collection - gcc (http://gcc.gnu.org/)
- How to compile?
  - gcc -c example.c
- Is the binary ready?
- How to link?
  - gcc -o example example.o
Can we automate the process of binary creation?

make - GNU utility for managing multiple commands

make files
- rules - <name>: <source files> <command>
- macros - <macro name> = <command>
debugging

- `gdb <binary>` (gdb example)
  - but first produce debug symbols for that binary
  - use `-g` flag when you compile `gcc -g -c example.c`

- `next (n)`, `step (s)`

- `print (p) <variable name>`

- `breakpoints - break (b)`
  - `b <line number or function name>`
  - `b <file name>:<line number>`
  - `info b` - display all breakpoints

- `continue (c)` - continues execution until next breakpoint
reverse engineering

- fast disassembly - gdb disas
  - `gdb <binary name>`
  - `info line main` - returns the address of main function if the binary has a main function
  - `diasas <start address for main>`
  - you don’t need debug symbols for this

- objdump
  - `objdump -i`
  - `objdump -D <binary name> > binary.s`
  - complete disassembly: `objdump -Dslx <binary name> > binary.s`
Summary

- C basics
- Pointers and
  - Arrays
  - Strings
- Data Structures
Variables

- Variables (_)*[a-zA-Z0-9]+(_)*[a-zA-Z0-9]+(_)*
- Declare before use
- Global
  - Outside functions
  - All functions can use them
  - Program execution lifetime
- Local
  - Within function
  - Only that function can use them
  - Function execution lifetime
Types, operators and expressions

- char, int, float, double
- short, long
- signed, unsigned
- sizeof(<variable>);

- arithmetic operators
  - + - * / %

- relational operators
  - > >= <= == !=

- Logical operators
  - ! && ||

- Bitwise operators
  - & | ^ << >> ~
Expressions

- \( \text{expr2} = (\text{expr2}) \langle \text{operator} \rangle (\text{expr1}) \)
- \( \text{expr2} \langle \text{operator} \rangle = \text{expr1} \)
- \( \text{expr3} \ ? \ \text{expr1} : \text{expr2} \)

eg:

```c
int x, y, z;
x = x*y;
x *= y;
z = (x>3)?x:y;
```
I/O

- `#include <stdio.h>
- `printf("%d\n", sum);
- `scanf("%d", &x);
- `%c, %d, %u, %x, %f, %e, %g
- character, decimal, unsigned, hexadecimal, normal floating point, exponential floating point, either normal or exponential floating point
Control Flow

if

eg:

if (h >= 6 && h <= 10)
printf("Good morning!\n");

if (height > 6.3)
printf("You were drafted in NBA!");
else
printf("Carefully prepare the ");

switch

eg:

switch (input)
{
    case 'h':
        printf("This is a demo!\n");
        break;
    case 'q':
        printf("Program will exit!\n");
        break;
    default:
        printf("invalid");
}
Control Flow

while

eg:

while ( i<100 )
{
    sum += i;
    i++;
}

do-while

eg:

do
{
    sum += i;
    i++;
} while ( i<100 )
Control Flow

for

for (<initialization>; <termination condition>; <expression>)
{
}
}

eg:

for (i = 0; i < 20; i++)
{
    sum += i;
}

Control Flow

**break**

Provides an early exit to the inner most enclosing loop or switch.

eg:

/* infinite loop */
for (;;)
{
    ...
    printf(“Press q to exit”);
    scanf(“%c”, &ch);
    if (ch == ‘q’) // break
        break;
}  

**continue**

Provides a way to skip the current iteration.

eg:

i = 0;
do
{
    if (i%2 == 1)
        {
            i++;  // continue
        }

    sum += i;
i++;
i++;
} while ( i<100 )


Structure of C programs

- preprocessor statements
  - `#include <stdio.h>`
- variable declarations
  - `int x;`
  - `int s=0;`
- functions
  - `int main() { ...}`
- constants
  - `#define pi 3.1415`
  - `enum boolean {NO, YES};`
  - `int x;`
  - `const int y = 234;`
  - `int s = 0;`
  - `x = 123;`
  - `s = x + y;`
  - `printf("Sum: %d\n", s);`
  - `return 0;`
Functions

- function declaration - can be in .h files
  - `void <function_name>(void);`
  - `void <function_name>(int <variable>, …);`
- function definition - has to be in .c files
  ```c
  int <function_name>(int <variable>, …)
  {
    <body>
    return <int value>
  }
  ```
Parameter passing

- Pass by value
  
  int sum(int x, int y)
  {
    return x + y;
  }

- Pass by reference
  
  void swap(int *x, int *y) {
    int temp = *x;
    *x = *y; *y = temp;
  }
  
  int x = 8239, y = 2375;
  int *px = &x, *py = &y;
  swap(&x, &y);
  swap(px, py);
Arrays

- array declaration
  - int array[10];
  - how many elements?

- arrays are indexed from 0

- initialization
  - int array[5] = {1, 2, 3, 4, 5};

- 2-dimensional array (matrix)
  - int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
  - matrix[1][2] = ?
Pointers

- Pointers - are memory locations that store the address of another memory location

- Operators for pointers:
  - * - dereferencing
  - & - address of (not to be confused with bitwise AND)

Eg:
```c
int *p;       //pointer to integer
int x = 10;   //integer variable
p = &x;       // address of x
printf("%d", *p);
```
Pointers and Arrays

- name of the array represents the address of the first element in the array

- `int *p; p = a;`

- or initialize at declaration:
  - `int *p = a;`
  - `int *p = &a[0]`

Eg:

```c
int *p;       // pointer to integer
int a[2] = {10, 7}; // array of integers
p = &a[0];     // address of a[0]
p = a;          // name of the array
```
Pointers and Arrays

- int a[10]; int *p = a;
- what is p+1 and a+1?
- p+1 - &a[1]
- what is *(p + 1)?
- p + 1 - a[1]

Eg:
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int *p, i;
for (p = a; p < (a+10); p++) {
    printf("%d ", *p);
}
Pointers and Strings

- Strings = array of chars
- char c[3] = {'e', 'n', 'j', 'o', 'y'};
- char string[] = “enjoy”;
- char string[] = {“enjoy”};
- char string[] = {'e', 'n', 'j', 'o', 'y', '\0'};
- char *string = “enjoy”;


Pointers and Strings

String functions in C
- `puts(str);` - writes to stdout
- `gets(str);` - reads from stdin
- `strcat(str1, str2);` - concatenates 2 strings
- `strcpy(str1, str2);` - copies str2 in str1
- `strcmp(str1, str2);` - compares strings
- `strlen(str);` - length of the string
- `strlwr(str);` - to lower case
- `strupr(str);` - to upper case
main()
{
  char a[ ]="I am happy", b[20], *p1, *p2; int i;
p1=a; p2=b;
  for(; *p1!='\0'; p1++, p2++)
      *p2=*p1;
  *p2='\0';
}
Structures

- heterogeneous datatypes

- struct <name> {
  - type1 element;
  - type2 element;
  - ...
} <variables>;

- accessing elements inside a structure
  - printf("%s", <struct_variable>.name);
  - printf("%f", <struct_variable>.score);
  - printf("%f", <pointer_to_struct_variable>-score);
structures and array of structures

struct student
{
    int id;
    char name[20];
    float score;
} stu = {5265045, “Joe”, 99.9};


struct student
{
    int id;
    char name[20];
    float score;
Pointers and Structures

main()
{

    struct student{
        int id;
        char name[20];
        float score;
    };

    struct student stu;
    struct student *p;
    p=&stu;
    stu.id=208;
    printf("%d\n", stu.id);
    printf("%d\n", (*p).id);
    printf("%d\n", p->id);
}

Pointers and Structures

- Linked lists
  - have a pointer to a struct of identical type

- Binary trees
  - have 2 pointers to a struct of identical type

```c
struct node{
    int value;
    struct node *next;
};
```

```c
struct node{
    int value;
    struct node *right;
    struct node *left;
};
```
Pointers and Structures

```c
struct node{
    int value;
    struct node *next;
};
struct node ar[10];
main(){
    int k;
    struct node *p; p=ar;
    for(k=0; k<9; k++)
    {
        p->value=k; p->next=&ar[k+1]; p++;
    }
    p->value=9;
    p->next =NULL; p=&ar[0];
    while(p!=NULL)
    {
        printf("%d 0x%x 0x%x\n", p->value, p, p->next);
        p=p->next;
    }
}
```