F28HS2 Hardware-Software Interface

Lecture 1: Programming in C 1

Introduction

- in this half of the course we will study:
  - system level programming in C
  - assembly language programming for the ARM processor
  - the relationship between high level programming language constructs and low level realisations
  - high and low level approaches to manipulating information

C Overview

- strict, strongly typed, imperative system programming language
- combines high-level constructs with low level access to type representations and memory
- Reference: B. Kernighan & D. Ritchie, *The C Programming Language (2nd Ed)*, Prentice-Hall, 1988

Overview

- C looks like Java ***BUT IS VERY DIFFERENT!***
- Java has high-level objects
- C exposes low-level memory formats & addresses
- must manage memory explicitly
- very easy to make programming errors
  - almost invariably address errors

Running C programs

- `gcc` - GNU c compiler
  - open source
- generates code for just about every conceivable platform

$ gcc -o $name_2$ $name_1$.c

- generate code for $name_1$.c
- put executable in $name_2$

$ $name_2$

- run program in $name_2$

Running C programs

```
$ gcc … -O …
```

- generate optimised code

$ `gcc -c` *name*$_1$`.c` … *name*$_N$`.c`

- generate object files *name*$_1$.o … *name*$_N$.o only

$ `gcc -o` *name* *name*$_1$`.o` … *name*$_N$`.o`

- link object files *name*$_1$.o … *name*$_N$.o and put executable in *name*

Running C programs

`$ gcc` *`name`*`.c`

- forgot `-o` *`name`* ?
- puts executable in `a.out` !

`$ man gcc`

- Linux manual entry for GNU C compiler

- can often use `cc` instead of `gcc`
  - proprietary C compiler for host OS/platform

Debugging C programs

`$ gcc -g -o` *name*$_2$ *name*$_1$`.c`

`$ gdb` *name*$_2$

- runs GNU debugger with *name*$_2$
- can now:
  - step through program
  - display values of variables
  - set break points

`$ man gdb`

- Linux manual entry for GNU debugger

Raspberry Pi

- usually runs in *user mode*
  - restricts what user can do
- *superuser* has full access
  - precede <u>every</u> command with: `sudo`
  - or
  - run in superuser shell: `sudo su`

Program layout

1. `#include ...`
2. `#define ...`
3. `extern ...`
4. *declarations*
5. *function declarations*
6. `main(int argc,char ** argv)`
7. `{ ... }`

Program layout

- 1. include files
  - #include "..." == look in current directory
  - #include <...> == look in system directories
  - import libraries via header files: *name*.h

e.g. <stdio.h> for I/O

```
1.      #include ...
2.      #define ...
3.      extern ...
4.      declarations
5.      function declarations
6.      main(int argc,
7.           char ** argv)
8.      { ... }
```

Program layout

- 2. macro and constant definitions
- 3. names/types of variables/functions used in this file but declared in linked files

```
1.      #include ...
2.      #define ...
3.      extern ...
4.      declarations
5.      function declarations
6.      main(int argc,
7.           char ** argv)
8.      { ... }
```

Program layout

- 4. & 5. declare all variables before all functions
- 6. & 7. main function with optional command line argument count and array
- declarations and statements terminated with a ;

```
1.      #include ...
2.      #define ...
3.      extern ...
4.      declarations
5.      function declarations
6.      main(int argc,
7.           char ** argv)
8.      { ... }
```

Display output 1

`printf(`“*text*”`)`

- system function
- sends *text* to the display
- `\n` == newline
- `\t` == tab

Display output 1

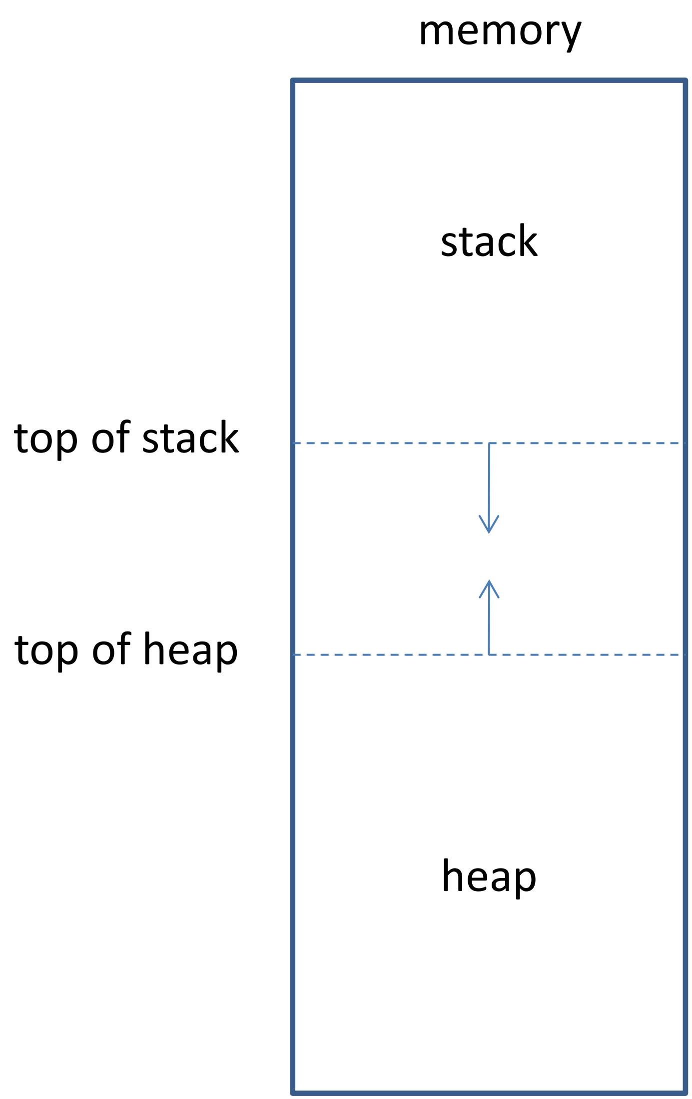
- **e.g.** `hello.c`

```
#include <stdio.h>

main(int argc,char ** argv)
{  printf("hello\n");
}
...
$ gcc -o hello hello.c
$ hello
hello
$
```

Memory organisation

- stack
  - allocated automatically
  - global declarations
  - local declarations
  - function parameters
- heap
  - allocated by program
  - c.f. Java `new`
  - no garbage collection



Declarations

- basic types
  - `char` – character
  - `int` – integer
  - `short` – short integer
  - `long` – long integer
  - `float` – floating point number
  - `double` – double size floating point number

Declarations

type name;

- allocates space for new variable of *type* called *name* on stack

name

- *letter*s + *digit*s + _ starting with a *letter*
- C convention
  - lower case = variable name
  - UPPER CASE = symbolic constant

Declarations

- can group declarations for same type

type $name_1$;

type $name_2$;

…

type $name_N$;

➔

type $name_1, name_2 \ldots name_N$;

Expressions

- *constant* ➔ value of constant
- *name* ➔ value from memory
  - NB value may differ depending on what type context *name* is used in
- unary/binary expression
- function call
  - NB C function == Java method

Constants

- signed integer
  - e.g. `4231 -2579`
- signed decimal
  - e.g. `886.754`
  - e.g. `-3.9E11` == `-3.9 * 10`^{11}
- character: '*letter*'
- e.g. `'a'  '\n'`

Operator expressions

- unary

op exp

  - evaluate *exp*
  - apply *op* to value

- binary infix

exp_1 *op* exp_2

  - evaluate exp_1
  - evaluate exp_2
  - apply *op* to values

Arithmetic

- unary minus: –
- binary infix

  \+ == add

  – == subtract

  * == multiply

  / == division

  % == integer modulo/remainder

Arithmetic

- (. . .) – brackets
- precedence

  (. . .) before...

  unary - before...

  * or / or % before...

  + or - before ...

  function call
- expressions evaluated from left to right

Arithmetic

- mixed mode arithmetic permitted
- always works at maximum precision needed
- for a binary operator:
  - `char` & `short` converted to `int`
  - `float` always converted to `double`
  - either operand is `double` then the other is converted to `double`
  - either operand is `long` then the other is converetd to `long`

Function call

- function called as:

name ($exp_1 ... exp_N$)

- evaluate actual parameters exp_1 to exp_N
  - pushing values onto stack
- function will access formal parameters via stack
- result of function execution is returned

Display output 2

`printf`("*format*" , $exp_1 ... exp_N$)

- displays the values of expressions $exp_1 ... exp_N$ depending on the *format characters*

`%d` == decimal integer

`%f` == floating point

`%x` == hexadecimal

`%s` == string

Display output 2

- NB variable number of arguments
- must have one format for each argument
- any non-format information in string is displayed as text

Address operator: &

- declaration: *type name*
- associates *name* with address of enough memory for type

&*name*

- address of 1st byte allocated to variable *name*
- *lvalue*
- on PC Linux/Raspbian: address == 4 bytes

Keyboard input

`int scanf(`“*format*” , $addr_1$…$addr_N$`)`

- inputs from keyboard to memory at specified addresses
  - depending on format characters
- typically, $addr_i$ is `&`$name_i$
- i.e. **address associated with variable** $name_i$
- `int` return value for success or end of input or failure

Example: polynomial evaluation

- evaluate AX^2+BX+C

```
#include <stdio.h>

main(int argc,char ** argv)
{  int a,b,c,x;
   printf("a: "); scanf("%d",&a);
   printf("b: "); scanf("%d",&b);
   printf("c: "); scanf("%d",&c);
   printf("x: "); scanf("%d",&x);
   printf("%d\n",a*x*x+b*x+c);
}
```

```
$ poly
a: 3
b: 8
c: 4
x: 6
160
```

Example: polynomial evaluation

- evaluate AX^2+BX+C

```
#include <stdio.h>

main(int argc,char ** argv)
{  int a,b,c,x;
   printf("a: "); scanf("%d",&a);
   printf("b: "); scanf("%d",&b);
   printf("c: "); scanf("%d",&c);
   printf("x: "); scanf("%d",&x);
   printf("%d\n",a*x*x+b*x+c);
}
```

put value at address in memory for variable

```
$ poly
a: 3
b: 8
c: 4
x: 6
160
```

Indirection operator: `*`

`*`*expression* ➔

- evaluate *expression* to integer
- use integer as address to get value from memory

- so *name* in *expression* ➔ `*(&`*name*`)`

1. get address associated with *name*
2. get value from memory at that address

Assignment

$expression_1 = expression_2$;

- evaluate $expression_1$ to give an address
  - *lvalue* – on left of assignment
- evaluate $expression_2$ to give a value
  - *rvalue* – on right of assignment
- put the value in memory at the address

Assignment

- assignments are expressions
- returns the value of *expression*$_2$
- value ignored when assignment used as a statement

Logic and logical operators

- no boolean values
- `0` ➔ false
- any other value➔true
- unary

`!` - not

- binary

`&&` - logical AND

`||` - logical OR

Comparison operators

- binary

`==` - equality

`!=` - inequality

`<` - less than

`<=` - less than or equal

`>` - greater than

`>=` - greater than or equal

Precedence

(...) before

&& before

|| before

! before

comparison before

arithmetic before

function call

Block

{ *declarations*

statements

}

- *declarations* are optional
- space allocated to *declarations*
  - on stack
  - for life of block

Iteration: `while`

`while(`*expression*`)`

statement ➔

1. evaluate *expression*
2. if non-zero then
   i. execute *statement*
   ii. repeat from 1.
3. if zero then end iteration

- `break` in *statement* ends enclosing iteration

Example: sum and average

- `sumav.c`

```
include <stdio.h>

main(int argc,char ** argv)
{   int count;
    int sum;
    int n;

    count = 0;
    sum = 0;
```

Example: sum and average

```
  printf("next> ");
  scanf("%d",&n);
  while(n!=0)
  {   count = count+1;
      sum = sum+n;
      printf("next> ");
      scanf("%d",&n);
  }
  printf("count: %d, sum: %d, average:
 %d\n",count,sum,sum/count);
}
```

Example: sum and average

```
$ sumav
next> 1
next> 2
next> 3
next> 4
next> 5
next> 0
count: 5, sum: 15, average: 3
```

Iteration: `for`

`for(`*exp*$_1$`;`*exp*$_2$`;`*exp*$_3$`)`
  statement ➔

exp$_1$;
`while(`*exp*$_2$`)`
`{` *statement*
  exp$_3$;
`}`

1. execute statement *exp*$_1$
2. repeatedly test *exp*$_2$
3. each time *exp*$_2$ is true
   1. execute *statement*
   2. execute statement *exp*3

- all *exps* and *statement* are optional

Iteration: `for`

`for` (*exp_1*; *exp_2*; *exp_3*)
 statement

- usually:
  - *exp_1* initialises loop control variable
  - *exp_2* checks if termination condition met for control variable
  - *exp_3* changes control variable
- NB must declare control variable before `for`

Condition: `if`

`if` (*expression*)

statement$_1$

`else`

statement$_2$ ➔

1. evaluate *expression*
2. if non-zero then execute *statement*$_1$
3. if zero then execute *statement*$_2$

- `else` *statement*$_2$ is optional
  - if *expression* is zero then go on to next statement

Condition: `switch`

```
switch(expression)
{   case constant_1: statements_1
    case constant_2: statements_2
    ...
    default: statements_N
}
```

1. evaluate *expression* to a value
2. for first $constant_i$ with same value, execute $statements_i$
3. if no *constant*s match *expression,* evaluate `default` $statements_N$

Condition: `switch`

```
switch(expression)
{   case constant1: statements1
    case constant2: statements2
    ...
    default: statementsN
}
```

- only matches `char` & `int/short/long` constants
- `break;` → end `switch`
- NB no `break` at end of *statements*$_i$ → execute statements$_{i+1}$!

Example: guessing number

- player thinks of a number between 1 and 100
- computer has to guess number
- each time, player tells computer if guess is:
  - correct
  - high
  - low
- computer uses divide and conquer to halve search space each time

Example: guessing number

- keep track of high and low boundaries
  - initially high is 100 and low is 1
- guess number between boundaries
  - if high then set high to guess
  - if low then set low to guess
- at end, output count of guesses

Example: guessing number

```
#include <stdio.h>

main(int argc, char ** argv)
{   int low, high, guess, response, count;

    low = 1;
    high = 100;
    count = 0;
```

Example: guessing number

```
while(1)
{   guess = (high+low)/2;
    count = count+1;
    printf("I guess %d.\n",guess);
    printf("Am I correct (0), high (1) or
            low (2)? ");
    scanf("%d", &response);
    if(response==0)
     break;
```

Example: guessing number

```
    switch(response)
    {  case 1: high = guess; break;
       case 2: low = guess; break;
       default: printf("I don't understand
                                  %d.\n",response);
                count = count-1;
    }
  }
  printf("I took %d guesses.\n",count);
}
```

Example: guessing number

```
[greg@mull 101]$ guess
I guess 50.
Am I correct (0), high (1) or low (2)? 1
I guess 25.
Am I correct (0), high (1) or low (2)? 2
I guess 37.
Am I correct (0), high (1) or low (2)? 9
I don't understand 9.
I guess 37.
Am I correct (0), high (1) or low (2)? 1
I guess 31.
Am I correct (0), high (1) or low (2)? 0
I took 4 guesses.
```