COMP281 Principles of C and memory management

lecture 4

Dr. Frans Oliehoek
Department of Computer Science,
University of Liverpool.

Previous Lecture

- some details of the stack
  - arguments != parameters
  - function call protocal

- How to use OJ
  - how do you like it?

- How to submit

Today

- Yet more elements of the C language…
  - Bitwise operators
  - Pre-processor defines and conditions
  - Conditions, short-circuit operators
  - Functions…
    - prototypes, implicit types, variable number of arguments
- More debugging!
  - GDB!
- Intro pointers
  - you'll love them!

More C Language Elements

Operators

- Most arithmetic operators you know (+,-,/,*,--,++, etc.)
  - '%' - modulo (integer remainder)

- Comparison operators; as in Java (==, >, <, >=, <=, !=)

- Logical operators (!, &&, ||)
  - see some of those in a bit

- For complete list, just look online!
  - e.g.: https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

bitwise operations

- C Supports a number of bitwise operations
  - You treat the operands as binary numbers

- | Or
- ^   Xor
- &   And
- <<n   Shift left n bits
- >>n   Shift right n bits

- correspond directly to assembly language instructions
  - Hence they are very efficient

- You can perform many logical operations by bitwise manipulations
  - This is quite common when dealing with I/O, such as graphics or sound

Some useful bitwise operations

- An ‘odd’ binary number always ends in 1! (by definition)
  - e.g. 3 = 11 and -1 = 11111111

- To find if a number is odd; you only need to read the last bit
  - The easy way to do this is what an ‘AND’ operation
  - e.g. (X & 0b000001)
    - is 1 if X is odd
    - and 0 if X is even

- ‘AND’ can be used to ‘mask off’ any bits you don’t want
  - e.g. to keep only the lower 8 bits;

```
char letter;
scanf(“%c”, &letter);
unsigned int ascii_value = letter & 255;
```

(255 = 0xFF = 0b11111111)

Some useful bitwise operations

Some more examples – try yourself!

From: http://www.geeksforgeeks.org/interesting-facts-bitwise-operators-c/

```
/* C Program to demonstrate use of bitwise operators */
#include<stdio.h>
int main()
{
    unsigned char a = 5, b = 9;  // a = (00000101), b = (00001001)
    printf("a = %d, b = %d\n", a, b);
    printf("a&b = %d\n", a&b);   // The result is 00000001
    printf("a|b = %d\n", a|b);   // The result is 00001101
    printf("a^b = %d\n", a^b);   // The result is 00001100
    printf("~a = %d\n", a = ~a); // The result is 11111010
    printf("b<<1 = %d\n", b<<1); // The result is 00010010
    printf("b>>1 = %d\n", b>>1); // The result is 00000100
    return 0;
}
```

Defining constant values

```
const int max_size = 20;

main()
{
    if (check_size(array) > max_size)
        max_size = max_size * 2;
}
```

- This will not compile!
- A variable declared `const` is considered to be ‘read only’, and should not be changed.
- But it still takes up memory as a normal variable does.
  - (And there are, in fact, ways to change it!)

Defining constant values

```
const int max_size = 20;

main()
{
    if (check_size(array) > max_size)
        max_size = max_size * 2;
}
```

- This will not compile!
- A variable declared `const` i
  not be changed.
- But it still takes up memory .
  - (And there are, in fact, ways to

Proper use of constants is good style!
→ it will affect your grade for the lab

#define

```
#define START 0 /*lower limit of table*/
#define END 12 /*upper limit*/
#define STEP 1 /*step size*/
#define TABLE 12

main()
{
  int x;
  for(x=START; x<=END; x=x+STEP)
  printf("%d \t %d \t %d\n",x,TABLE,x*TABLE);
}
```

- #define is, effectively, a text-replacement that occurs BEFORE compilation (a preprocessor macro)
  - convention: use **ALL_CAPITALS** for #defined constants
- It can also be used for more complex purposes, such as generating code…

#define

```
#include<stdio.h>
#define LOOP(X)  for (i=0;i<X;i++)

main()
{
    int i;
    LOOP(10)
        printf("%d\n",i);
}
```

- `#define` is entirely syntactic – it’s easy to get it to do odd things (beware of floating conditionals, brackets, etc.)
- It is always evaluated at compile-time, never when the program is run
  - but it can still be used to refer to program variables
- Uppercase is often used to indicate that this is a macro definition somewhere and not, for example, a function name

Pre-processor Conditions: #if

- What does this code produce?

```
#include<stdio.h>

#define DEBUG_LEVEL 1
main()
{
#if DEBUG_LEVEL == 1
    printf("main starting \n");
#else
    some invalid nonsense
#endif
    printf("main ending \n");
}
```

Conditions

- Remember we don’t have Boolean types!
- Question: so how does an `if` work? (what does the expression evaluate to?)

```
main()
{
    int a = 0;
    if (a > 0 && 1/a==1)
        printf("1/a is 1\n");
    if (a = 1)
        printf("a is now 1\n");
    if (a > 0 & 1/a==1)
        printf("1/a is 1\n");
    if (a == 1)
        printf("a is 1\n");
    if ( (a==1) && (a++==2) )
        printf("a is now 2\n");

    printf("a is %d\n",a);
}
```

Conditions

```
example$ ./conditions
a is now 1
1/a is 1
a is 1
a is 2
```

- Remember we don’t have Boolean types!
- Question: so how does an `if` work? (what does the expression evaluate to?)

```
main()
{
    int a = 0;
    if (a > 0 && 1/a==1)
        printf("1/a is 1\n");
    if (a = 1)
        printf("a is now 1\n");
    if (a > 0 & 1/a==1)
        printf("1/a is 1\n");
    if (a == 1)
        printf("a is 1\n");
    if ( (a==1) && (a++==2) )
        printf("a is now 2\n");

    printf("a is %d\n",a);
}
```

Like Java && is for comparison, if the first part is false the second part is not evaluated

An assignment evaluates to its value

(This is a very common mistake)

& is a bitwise operation, so the second part is always evaluated (and could throw an exception!)

This works correctly

This gets to the second part, but the evaluation is performed before the increment

Conditions (cont'd)

- An expression evaluates to 0 (false) or 1 (true)
  - Hence you can do things like
    `int numdigits = 1+(a > 9)+(a > 99);`
  - But these can be difficult to read

- Beware of = in conditions, instead of ==
  - some people prefer if (1==a) instead

- Can perform operations inside conditions
  - So-called 'side-effects'; you can also call functions.
  - Beware of precedence, particularly with **short circuit operators** (`&&, ||`) some may not be executed

- Beware of bitwise operations, such as `&` or `|` when you mean `&&`
  - They will often work, because `1&1 = 1`, etc., but may fail with numerical input
  - e.g. `(2 && (2 > 1)) = 1` but `(2 & (2 > 1)) =0`
  - All evaluation is performed, so all side effects will be executed

Conditional operator

- A ternary operator (three operands) that can choose between two outcomes
- It takes the form

```
condition ? if_true : if_false
```

- Example – you can write

```
if (y < 0)
   x = -1;
else
   x = 1;
```

- As the following

```
x = (y < 0) ? -1 : 1
```

More about functions…

Reminder declaration vs definition

- A function declaration has no code

```
int square(int);
```

  - this gives the compiler information about the function - its name and the types of the parameters
  - This is not the same as an empty function:

```
int square(int)
{   }
```

- The parameter names are optional in the declaration, **but not in the definition** (or else you couldn’t access them!).
  - It is very common for the names to also be in the declaration

Implicit types in declarations

- If a function is not given a return type it is implicitly assumed to return an int

- If a function doesn’t return anything (when it should), it will still compile
  - e.g.
    `double get_f() { printf(“get_f”)  };`
  - The result is not defined!

- functions should be declared (or defined) before being called
  - This declaration (or definition) is also called a ‘prototype’

- “ `()` ” is valid in a declaration
  - e.g.,
    `double get_f();`
  - This does not mean ‘no parameters’ as it does in Java
    (should write `double get_f(void);`)
  - It means **the called function can deal with ANY parameters passed**
  - This is not encouraged!

function_prototypes.c

Implicit types in declarations

- If a function is not given a return type it is implicitly assumed to return an int
- If a function doesn’t return anything (when it should), it will still compile
  - e.g.
    ```
    double get_f() { printf(“get_f”)  };
    ```
  - The result is not defined!
- functions should be declared (or defined) before being called
  - This declaration (or definition) is also called a ‘prototype’
- “ `()` ” is valid in a declaration
  - e.g.,
    ```
    double get_f();
    ```
  - This does not mean ‘no parameters’ as it does in Java (should write `double get_f(void);`)
  - It means **the called function can deal with ANY parameters passed**
  - This is not encouraged!

Use “`-Wall`” to get a warning about these issues!

function_prototypes.c

Functions: quiz time

Question: Which of the following functions work?

```
1.  main()
{
}

2.  main()
{
  return 0;
}

3.  int main()
{
}

4.  int main()
{
  return 0;
}
5.  int main(void)
{
  return 0;
}
```

Functions: quiz time

Question: Which of the following functions work?

```
1.   main()
{
}

2.   main()
{
  return 0;
}

3.   int main()
{
}

4.   int main()
{
  return 0;
}
5.   int main(void)
{
  return 0;
}
```

They all work!

the return type defaults to int

no return gives an ***undefined*** value!

Specifies that main takes no parameters
() and (void) are different – use (void) if you don't intend to have parameters passed

Variable Number of Parameters

So how does printf work?

There is a special type of parameter passing for variable arguments

```
#include<stdio.h>
#include<stdarg.h>
int getAverage(int number,...)
{
  int sum = 0, count = 0, val = 0;
  va_list argument_pointer;
  va_start(argument_pointer, number);
  while ((val = va_arg(argument_pointer, int)) > 0)
  {
    count++;
    sum = sum + val;
  }
  return (sum / count);
}
```

file:e9.c

Variable Number of Parameters

- How does the program know how many parameters there are?
- It doesn’t – you must do that yourself
  - e.g., passing some number or using some special value to terminate

- How does the program know what type each parameter is?
- It doesn’t – you must do that yourself
  - and things may go badly wrong if you read an incorrect type

- Will I need to use this in my code?
- It is usually best to avoid it
  - Unpredictable things can happen if small mistakes are made
  - e.g., we have seen how easy it is for printf to crash the program
    - on some systems this may even crash the whole operating systems

Summary of Elements

- Bitwise operators
- Pre-processor defines and conditions
- Conditions, short-circuit operators
- Functions…
  - prototypes, implicit types, variable number of arguments

Debugging

Online Judge Says: “No”

- Before you start debugging… Read the notifications!
  - There may be useful information there!
  - e.g., is it a 'run-time' error or 'compilation' error? (and do you know the difference?!)

- Things to do:
  - Read the problem again
    - Are you sure you have interpreted it correctly?

  - Test it yourself: It’s very likely (99%) that the fault is with your program…!
    - Did you try different test cases?
    - What **assumptions** did you make about the test cases?
    - DON’T assume the test case(s) give you all of the possibilities
    - Did you try…. large numbers, small numbers, negative numbers, …, etc.?

  - Check the output
    - Does the output match ***exactly***?
    - Don’t print anything else to the screen!
    - “replicate OJ” by piping a test file into your program! (see previous lecture)

Example Bug

- Can you spot the mistake?

```
void swap_chars(char* char_arr, int i, int j)
{
    char t = char_arr[i];
    char_arr[i] = char_arr[j];
    char_arr[j] = t;
}
int main(void)
{
    char the_chars[SIZE];
    int a=0, b;
    int random_indices[SIZE];
    printf("type input, please:\n");
    scanf("%s", the_chars);
    b = strlen(the_chars);

    /*  generate a random numbers for each position:*/
    do {
        random_indices[a] = rand() % b;
        a++;
    } while (a < b);

    /* now swap the chars with their random index: */
    for(a = 0; a < SIZE; a++)
        swap_chars(the_chars, a, random_indices[a]);
    printf("randomly swapped string: %s\n",the_chars);
}
```

Debugging for run-time errors

- `gdb` – the GNU debugger
  - can be used to track down runtime errors
  - need to compile with debug options: `gcc -O0 -g -Wall`
- (demo)
- important commands to get started:
  - Loading and running: `file, run`
  - examining the stack: `bt, up, down, list`
  - setting up breakpoints: `break <line_number/function_name>, delete, info breakpoints`
  - stepping through code: `step, continue, finish`
- Try and do a tutorial! E.g.:
  http://www.thegeekstuff.com/2010/03/debug-c-program-using-gdb/

Pointers – Part 1

Pointers

- Program code and data are stored in memory

- Every location (e.g., each byte) in memory has an **address**
  - This is just a number telling the processor how to find it.

- In C we can **access** and **manipulate these addresses** directly
  - (similar to assembly language)
  - the variables that store such addresses are called **pointers**
  - Declared using '*'
- E.g.,:

```
int a = 42;
int * ptr_to_int = NULL;
ptr_to_int = &a;
```

- If you want to pass a function a large amount of data, it is much easier to just **pass a pointer to that data**

Pointers

- A **pointer** is a variable that contains the memory address of some item
- `*type` denotes 'a pointer to type'
- `&` denotes 'the address of'

```
int* pointer;  //the variable will contain a pointer to an integer
```

- We can use this, as follows:

```
int variableA;
int* pointer = &variableA;
```

- Pointer now contains the address of variableA…
  - so, how do we access its contents?
  - We also use the `*` notation for this:

```
int value = *pointer;
*pointer = 8;
```

Pointer – example

```
void set_to_10(int* ptr)
{
    *ptr = 10;
}
main()
{
    int v = 5;
    int* pointer = &v;
    int a[5] = {1,2,3,4,5};

    printf("v = %d \n", v);
    set_to_10(pointer);
    printf("v = %d \n", v);

    printf("a[0] = %d \n", a[0]);
    set_to_10(a);
    printf("a[0] = %d \n", a[0]);
    return(0);
}
```

Pointer – example

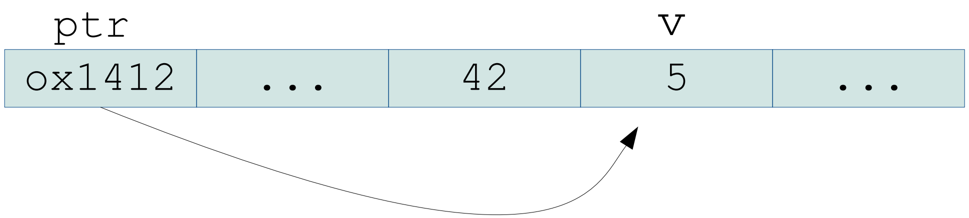
```
void set_to_10(int* ptr)
{
    *ptr = 10;
}
main()
{
    int v = 5;
    int* pointer = &v;
    int a[5] = {1,2,3,4,5};

    printf("v = %d \n", v);
    set_to_10(pointer);
    printf("v = %d \n", v);

    printf("a[0] = %d \n", a[0]);
    set_to_10(a);
    printf("a[0] = %d \n", a[0]);
    return(0);
}
```

pointer is **dereferenced**

i.e., the **value** of the address to which the pointer points is set (or retrieved)



ptr			v	
ox1412	...	42	5	...

Pointer – example

```
void set_to_10(int* ptr)
{
    *ptr = 10;
}
main()
{
    int v = 5;
    int* pointer = &v;
    int a[5] = {1,2,3,4,5};

    printf("v = %d \n", v);
    set_to_10(pointer);
    printf("v = %d \n", v);

    printf("a[0] = %d \n", a[0]);
    set_to_10(a);
    printf("a[0] = %d \n", a[0]);
    return(0);
}
```

Output.:

```
v = 5
v = 10
a[0] = 1
a[0] = 10
```

this is called “(C style) pass by reference”

Pointer – example

```
void set_to_10(int* ptr)
{
    *ptr = 10;
}
main()
{
    int v = 5;
    int* pointer = &v;
    int a[5] = {1,2,3,4,5};

    printf("v = %d \n", v);
    set_to_10(pointer);
    printf("v = %d \n", v);

    printf("a[0] = %d \n", a[0]);
    set_to_10(a);
    printf("a[0] = %d \n", a[0]);
    return(0);
}
```

Output.:

```
v = 5
v = 10
a[0] = 1
a[0] = 10
```

this is called "(C style) pass by reference"

- an array's name **is a pointer**
  - points to first element
  - i.e. `a == &a[0]`

Pointer / References

- In Java, you refer to objects by *'reference'*; same idea!
- except that a **C pointer refers to the actual memory address** used by the system:
  - can do pointer arithmetic (moving the pointer)
  - Java reference cannot be used in that way
- 'reference' tends to mean something slightly different in different languages...

  http://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value

Review

- Yet more elements of the C language…
  - Bitwise operators
  - Pre-processor defines and conditions
  - Conditions, short-circuit operators
  - Function prototypes, implicit types, variable number of arguments
- Debugging using `gdb`
- Pointers
  - variables whose value are memory locations
  - “pass by reference”
- You now know how to:
  - test for even/odd
  - how 'if' works
  - debug with gdb, e.g., examine the stack
  - define a pointer, pass by reference (and why scanf needs the “&” syntax!)