UNIX and C Programming (COMP1000)

Lecture 9: Miscellaneous C

Updated: 22nd October, 2019

Department of Computing
Curtin University

Textbook Reading (Hanly and Koffman)

For more information, see the weekly reading list on Blackboard.

- **Section 7.7** covers enumerated types.
- **Section 10.6** covers unions.
- **Appendix C** covers bitwise operators.

Outline

Timing

- C has a range of functions and data types for manipulating times and dates.
- We'll only cover one function here — `time()`.
- `time()` returns the number of seconds since an arbitrary date.
- On UNIX, the date is the "Epoch" — 1 January 1970 (midnight UTC) .
- You can use `time()` to measure an interval of time, or determine the calendar date.

Using time()

- time() takes one pointer parameter, which you can safely set to NULL.
- It will return the time in seconds.
- (If the parameter is not NULL, the time will also be stored at the given address.)
- The return type is "time_t", which is just a specially-sized integer.

Example

This code outputs the time taken to execute a function:

```
time_t startTime = time(NULL);
doStuff();
printf("%d secs elapsed\n", time(NULL) - startTime);
```

Random Numbers

- A lot of software relies on randomness or unpredictability (e.g. games, screensavers, password generators, etc.)
- Randomness is achieved by generating random numbers.
- In practice, these are usually “pseudorandom” numbers.
- They’re not truly random, but they’re close enough that nobody will notice.
  - Based on encryption algorithms.

Random Number Seed — srand()

- Before you can generate random numbers, you need a "seed".
- This is used to obtain a sequence of random numbers.
- If you always use the same seed, you'll always get the same sequence of "random" numbers!
- Set the seed with `srand()`.
- Takes one `int` parameter (the seed) and returns `void`.

Example

Use `srand()` with the `time()` function:

```
srand(time(NULL)); /* At start of your program */
...
```

Generating Random Numbers — `rand()`

- Use `rand()` to obtain random numbers (after `srand()`).
- Takes no parameters and returns a random `int`.
- Returns a different value each call.

Example

Prints some random numbers:

```
int i;
srand(time(NULL));
for(i = 0; i < 4; i++) {
    printf("%d ", rand());
}
```

Output: 1034290004 1846603216 1185721333 1246567316

Seed Once (Usually)

- In most cases, run `srand()` *once only*, when your program starts.
- Do *not* do this:

```
int i;
for(i = 0; i < 4; i++) {
    srand(time(NULL));      /* <-- Very silly! */
    printf("%d ", rand());
}
```

You'll get the *same* value repeated 4 times! Why?

  - `srand()` resets the sequence.
  - And we're giving it the same seed each time.
  - (Why? Because if `time()` is called multiple times per second, it must return the same value.)

- Only run `srand()` more than once if you need to *repeat* a particular sequence of pseudorandom numbers.

Random Numbers Within a Range

- `rand()` returns values between 0 and `RAND_MAX`.
- `RAND_MAX` is a platform-specific constant.
- It is not a very useful limit.
- You almost always need to constrain the range of numbers.
- Use division or the modulus (%) operator to do this.

Examples

To get a random `int` between 0 and `limit` - 1:

```
rand() % limit
```

To get a random `double` between 0.0 and 1.0:

```
(double)rand() / (double)RAND_MAX
```

Const

- Until now, you've used `#define` to create constants:

```
#define PI 3.14159
```

  - Done by the preprocessor, through by substitution.
  - Has no particular datatype.
- You can also use the `const` keyword:

```
const double PI = 3.14159;
```

  - Done by the C language itself, *not* the preprocessor.
  - Has a specific datatype.

Const Storage

- `const` values are (generally) stored in memory, like variables.
- In memory, a "`const int`" looks just like an "`int`".
- The "`const`" keyword is just a pledge that you won't modify the value.
- (You *can* modify a "const" value, by abusing pointers and typecasting, but don't!)

Pointers to Constants

- Since const values are stored in memory, you can have pointers to them.
- This creates some complex situations!

```
const int x = 100;
const int y = 200;
const int * ptr; /* Pointer to an int constant */

ptr = &x;
ptr = &y;
*ptr = 300;  /* Invalid (*ptr is constant) */
```

- Here, you can't modify x, y or *ptr — they're constants.
- You *can* modify the pointer address ptr, making it point somewhere else.

Variables as Constants

- You can effectively turn a variable into a constant, temporarily:

```
int x = 100;
const int * ptr = &x; /* Ptr to an int constant */

x = 200;
*ptr = 300;  /* Invalid (*ptr is constant) */
```

- Here, `x` is a variable, but `*ptr` is a constant.
- They both represent the same location in memory!

Functions and Constants

Functions can accept constant parameters:

```
void func(const int * ptr) {...}
```

- `*ptr` is constant *within* the function.
- We can still pass in an ordinary, non-constant variable:

```
int x = 100; /* Here, x is a normal variable */
func(&x);
/* x is guaranteed to still be 100 */
```

- So, we're only restricting the function. Why bother?
- When you want to call the function, you instantly know that it's not going to modify the values you pass in.

Constant Pointers

- Pointers themselves can be constant:

```
int x = 100;
int y = 200;
int *const ptr = &x; /* Constant ptr to an int */

*ptr = 200;
ptr = &y;     /* Invalid (ptr is constant) */
```

- Here, we can modify `*ptr` freely.
- `ptr` itself cannot be changed — it will always point to `x`.

Constant Pointers to Constants

- You can fix both the pointer and the thing it points to:

```
int x = 100;
int y = 200;

/* Constant pointer to a constant int. */
const int *const ptr = &x;

ptr = &y;     /* Invalid (ptr is constant)  */
*ptr = 300;  /* Invalid (*ptr is constant) */
```

Const Declarations

- Read declarations **right to left**.
- For example:

```
/* Variable pointers to constant ints. */
const int * ptr1a;
int const * ptr1b;

/* Constant pointer to a variable int. */
int *const ptr2;

/* Constant pointers to constant ints. */
const int *const ptr3a;
int const *const ptr3b;
```

(The order of `int` and `const` at the start makes no difference.)

Arrays of Constants

- You can have an entire array of constants:

```
const int array[] = {5, 10, 15, 20, 25};
```

- This works like a normal array, but you can't modify its contents.
- You can also have an array of constant strings:

```
const char *const months[12] = {
    "January", "February", "March", ...};
```

Global Constants

- Constants can be local or global, like variables.
- Global constants are *not* (necessarily) bad practice — they can be useful.

```
const int daysInMonth[12] = {31, 28, 31, ...};

/* These functions all use 'daysInMonth' */
void printCalendar(int month);
int isValidDate(int day, int month, int year);
...
```

Volatile

- Volatile variables are used in hardware programming.
- A volatile variable can:
  - cause side-effects when accessed; and
  - be modified by something external to the program.
- Consequently, the compiler cannot perform optimisations
- Volatile variables use the "`volatile`" keyword:

```
volatile int a;
volatile int * b;
int *volatile c;
const int *volatile d;
const volatile int e;
volatile int *const *volatile f;
```

- `volatile` can be used in the same way as `const`.
- `volatile` and `const` can be used together.

Enumerations

- An enumeration is a data type whose values are labels or categories.
- Enumerations are programmer-defined — there can be many different types (like structs).
- Enumerations only have one value at a time (like `ints`).
- Enumerations are *not* strings or `chars`!

Examples

You could define enumeration types to store the following data:

- A compass direction (north, south, east or west).
- A menu option (add, delete, search, etc.).

Enumeration Syntax

To define an enumeration:

```
/* Type declaration */
enum Direction {NORTH, EAST, SOUTH, WEST};
...

/* Variable declaration */
enum Direction dir = WEST;
```

OR

```
/* Type declaration */
typedef enum {NORTH, EAST, SOUTH, WEST} Direction;
...

/* Variable declaration */
Direction dir = WEST;
```

Using Enumerations

- Alternatives that have no natural numeric value.
- Most operators (+, /, >=, etc) don't make sense for enums.
- Often used with `switch` statements:

```
typedef enum {NORTH, EAST, SOUTH, WEST} Direction;
...
Direction dir;
int x, y;
...
switch(dir) {
    case NORTH: y++; break;
    case EAST:  x++; break;
    case SOUTH: y--; break;
    case WEST:  x--; break;
}
```

Enumeration Examples

```
typedef enum {CLUBS, HEARTS, SPADES, DIAMONDS} Suit;
```

```
typedef enum {
    OOPD, EP, UCP, ...
} Unit;
```

```
typedef struct {
    int numUnits;
    Unit* units;
    enum {FULLTIME, PARTTIME} studyMode;
    enum {
        GOOD, CONDITIONAL, TERMINATED, GRADUATED
    } standing;
} Student;
```

Enumerations as Integer Constants

- Behind the scenes, enum labels are really just integer constants.
- By default, the C compiler assigns values 0, 1, 2, etc. to each label in turn.
- For instance:

```
typedef enum {NORTH, EAST, SOUTH, WEST} Direction;

/* NORTH == 0
   EAST == 1
   SOUTH == 2
   WEST == 3 */
...
printf("%d\n", SOUTH);  /* Prints "2" */
```

Choosing Enumeration Values

- Normally, you don't care about the actual values — just an implementation detail.
- By default:
  - The first label is zero.
  - Each other label is one more than the previous.
- However, you can choose some or all of them yourself:

```
typedef enum {
    NORTH, EAST = 3, SOUTH = 10, WEST} Direction;

/* NORTH == 0
   EAST == 3
   SOUTH == 10
   WEST == 11  */
```

(You can only choose integer values.)

Unions

- A "`union`" looks like a `struct`, except...
- All the fields are stored in the same memory location!
- Obviously, you can only use one field at a time.
- Allows you to re-use the same memory for different things.
- Not used very often.

Union Syntax

You declare and access a `union` just like a `struct`:

```
typedef union {
    int intValue;
    char strValue[10];
} Ident;

...

Ident id;
id.intValue = 42; /* Overwrites id.strId! */
```

- `&id.intValue == id.strValue`
- Writing to one will corrupt the other!

Union Examples

```
typedef union {
    int *intp;
    double *doublep;
    char *charp;
} MultiPointer;
```

```
typedef struct {
    enum {CIRCLE, RECTANGLE} type;
    union {
        double radius;
        struct {float width; float height;} rect;
    } dimensions;
} Shape;
```

Bit Manipulation

- All data is (of course) an array of bits.
- However, bits cannot be read or written to directly.
- Here we discuss some tricks to manipulate them.

Visualisation

This is binary form of the 8-bit `char` value `'Z'`:

7	6	5	4	3	2	1	bit 0
0	1	0	1	1	0	1	0

(`ints`, `doubles`, etc. are much longer, of course.)

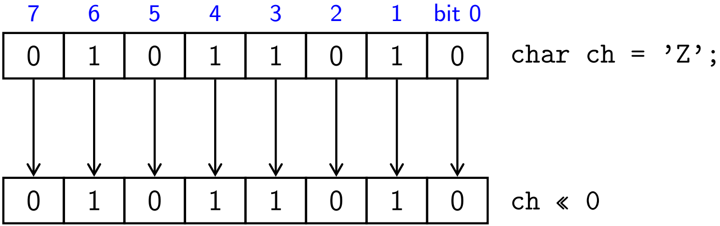
Shift Left and Shift Right

- << is the shift-left operator.
- >> is the shift-right operator.
- These shift a bit pattern left or right, by a given amount.
- For `int`s, this effectively multiplies or divides by powers of two.

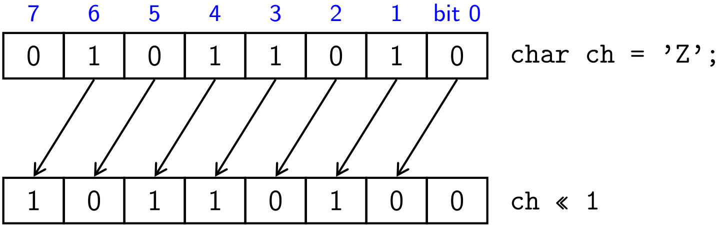
Example

```
int var = 100;

printf("%d\n", var << 1); /* 200 */
printf("%d\n", var << 2); /* 400 */
printf("%d\n", var << 3); /* 800 */

printf("%d\n", var >> 1); /*  50 */
printf("%d\n", var >> 2); /*  25 */
```

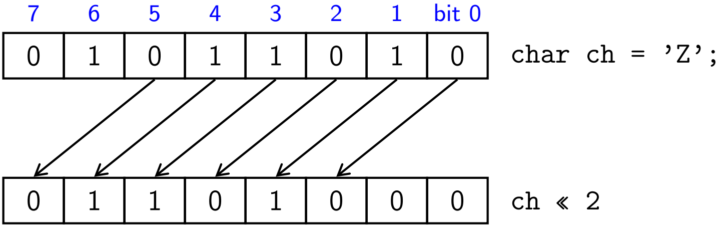
Shift Left — Graphical Example



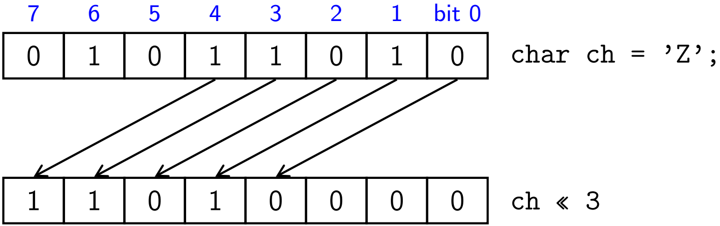
Shift Left — Graphical Example



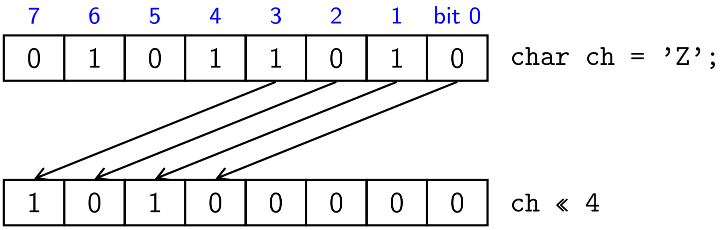
Shift Left — Graphical Example



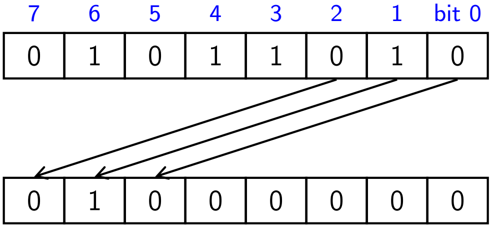
Shift Left — Graphical Example



Shift Left — Graphical Example



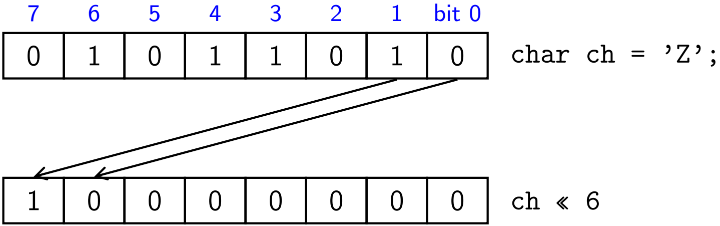
Shift Left — Graphical Example



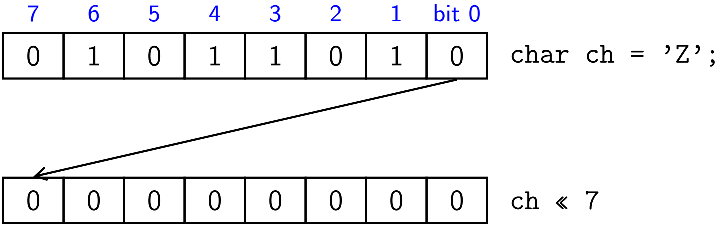
`char ch = 'Z';`

`ch « 5`

Shift Left — Graphical Example



Shift Left — Graphical Example



Bitwise Operators

- Remember the "logical" operators: `&&`, `||` and `!`.
- The "bitwise" operators are: `&`, `|`, `^` and `~`.
- Logical and bitwise operators do the same thing, except:
  - Logical operators work on whole variables.
  - Bitwise operators work on each bit separately.
- With *n*-bit data, bitwise operators will perform *n* separate AND/OR/XOR/NOT operations.
- The *n* resulting bits are returned as one value.

Bitwise NOT (~)

- The simplest bitwise operator.
- A unary operator (like "`!`") — takes only one operand.
- Inverts each bit in the operand — ones become zeros, zeros become ones.

7	6	5	4	3	2	1	bit 0	
0	1	0	1	1	0	1	0	`char ch = 'Z';` (90)
1	0	1	0	0	1	0	1	`~ch` (165)

Bitwise AND (&)

- Same symbol as the address-of operator, but different operation.
- "`a & b`" calculates the bitwise AND of `a` and `b`.
- Each resulting bit is 1 if *both* operand bits are 1.

7	6	5	4	3	2	1	bit 0	
0	1	0	1	1	0	1	0	`char z = 'Z';` (90)
			bitwise-AND					
0	1	0	0	0	1	1	0	`char f = 'F';` (70)
			=					
0	1	0	0	0	0	1	0	`z & f` (66, `'B'`)

Bitwise OR (|)

- "`a | b`" calculates the bitwise OR of `a` and `b`.
- Each resulting bit is 1 if *either* operand bit is 1.

7	6	5	4	3	2	1	bit 0	
0	1	0	1	1	0	1	0	`char z = 'Z';` (90)
			bitwise-OR					
0	1	0	0	0	1	1	0	`char f = 'F';` (70)
			=					
0	1	0	1	1	1	1	0	`z	f` (94)

Bitwise XOR (^)

- XOR means "exclusive OR".
- "`a ^ b`" calculates the bitwise XOR of `a` and `b`.
- Each resulting bit is 1 if the two operand bits are *different* (i.e. only one of the operands is 1).
- Used in encryption and hash functions.

7	6	5	4	3	2	1	bit 0	
0	1	0	1	1	0	1	0	`char z = 'Z';` (90)

bitwise-XOR

0	1	0	0	0	1	1	0	`char f = 'F';` (70)

=

0	0	0	1	1	1	0	0	`z ^ f` (28)

Bit Fields

- You can place "bit fields" inside structs (and unions).
- These are integers that occupy a specified number of bits.
- They can be signed (allowing for negative numbers) or unsigned.
- The number of bits determines how large their values can be:
  - 1 bit allows only zero and one.
  - 2 bits allows values 0 to 3 (unsigned) or -2 to 1 (signed).
  - 3 bits allows values 0 to 7 (unsigned) or -4 to 3 (signed).
  - 4 bits allows values 0 to 15 (unsigned) or -8 to 7 (signed).
  - etc.
- The syntax is:

```
unsigned int <name> : <bits>;
```

OR

```
signed int <name> : <bits>;
```

Bit Fields — Example

```
typedef struct {
    unsigned int flag : 1;        /* 0 or 1 */
    unsigned int quadrant : 2;    /* 0 to 3 */
    signed int rating : 4;        /* -8 to 7 */
    int normalInt;                /* much larger */
} SmallThings;
```

Here, the `SmallThings` struct has 3 bit fields and one normal `int` field.

Bit Fields — Limitations

- Unfortunately, pointers can only point to bytes, not bits.
- Therefore, you can't have a pointer to a bit field.
- The address-of operator won't work on bit fields.
- You can't pass a bit field by reference.
- You can't use it with `scanf()` (but `printf()` will work).

Coming Up

- That's the last of the C lectures!
- In the final lecture next week, you'll get a taste of C++.