COMP2303/COMP7306 Network and OS Principles

Week 3 - Tuesday

More C

School of Information Technology and Electrical Engineering
The University of Queensland

This Week

- Lectures
  - More C
  - Linked Lists
  - Debugging
- Pracs
  - Assignment One – due Wednesday 19th March
  - More C programming exercises
    - System calls and libraries

Outline

- Admin
- C Programming Language
  - Variants of main()
  - Dynamic memory allocation
  - I/O and files.
  - **switch** statements
  - **break**, **continue**
  - **unions**
  - Function pointers
  - Type casting
- Useful UNIX – indent, dos2unix

Admin

- Course newsgroup (uq.itee.comp2303) has quite a few posts – and plenty of people to answer any questions (yourselves, tutors, lecturers)
- Running UNIX at home... options:
  - Remote access to agave
  - Installing Linux
  - Using a "live cd" (eg Ubuntu).
  - Wubi – ubuntu on Windows.
  - Install the dev tools on a Mac.
  - …
  - Profit

Variants of main()

- **main()** is the entry point of the program
- Standard C permits
  - `int main(void) { … }`
  - `int main (int argc, char* argv[]) { … }`
  - `int main() { … }     /* OK, but not ** recommended */`
- **argc** is the number of command line arguments (including the program name)
- **argv[0]** is the program name
- **argv[1]** to **argv[argc-1]** are the command line arguments (if argc ≥ 2)
- **argv[argc]** is a null (0) pointer
- Illustration and example code to be presented in class

Variants of main() cont.

- Some implementations support third argument:
  - `int main(int argc, char* argv[], char* envp[]) {…}`
  - **envp** is a null terminated array of environment variables, each of the form "name=value"
- Can always use standard C **getenv()**

sizeof operator

- sizeof() returns number of “storage units” (usually bytes) occupied by a given type
  - `sizeof(`*type*`)`
  - `sizeof` *expression*
- Result is machine dependent
- Result includes any padding necessary because of alignment restrictions
  - Structures and unions padded out to size type would occupy as an element of an array
  - Structure elements stored in order of declaration

Dynamic Memory Allocation

- Sometimes we don’t know how much storage we need at compile time, e.g.:

```
int main(int argc, char* argv[]) {
    char cmdLine[MAX_SIZE];
    int i;
    cmdLine[0] = '\0';
    for (i=0; i<argc; i++)
    {   /* Construct orig. cmd. line */
        if(i != 0)
        {  /* If not 1st arg,add space */
           strcat(cmdLine, " ");
        }
        strcat(cmdLine, argv[i]);
    }
```

- How big should MAX_SIZE be?
  - No way to tell!

Dynamic Memory Allocation

- Can ask system to dynamically set aside memory using malloc()

```
#include <stdlib.h>
void *malloc(size_t size);
```

  - Returns pointer to first element of region
    - void* is a generic pointer type
      - Cast to the type you want
    - Allocated memory is not initialised
- When finished with memory, call free():

```
void free(void *ptr);
```

- See **calloc()** and **realloc()** also

Example

```
int main(int argc, char* argv[]) {
    char* cmdLine;
    int i;
    int size = argc;/* space between args, \0 at end */
    /* Work out size of command line */
    for (i=0; i<argc; i++) {
        size+= strlen(argv[i]);
    }
    cmdLine = (char*)malloc(sizeof(char) * size);
    cmdLine[0] = '\0';    /* empty string */
    for (i=0; i<argc; i++)
    {   /* Construct orig. cmd. line */
        if(i != 0)
        {  /* If not 1st arg,add space */
           strcat(cmdLine, " ");
        }
        strcat(cmdLine, argv[i]);
    }
```


(>=2)D arrays

- `int a[10][20]`
  is legal but you can't declare functions like this:
  `void f(int[][] p)`
  You can only leave the rightmost dimension empty.

Option 1 (RxC)

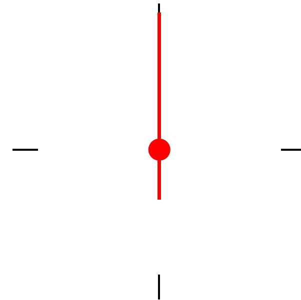
- `int* v=malloc(sizeof(int)*R*C);`
- To access [x][y] we would write:
  `v[x*R+y]`
- `free(v);`

Option2

- ```
  int** v=malloc(sizeof(int*)*R);
  for (i=0;i<R;++i) {
       v[i]=malloc(sizeof(int)*C);
  }
  ```
- To access [x][y] use `v[x][y]`.
- ```
  for (i=0;i<R;++i) {
       free(v[i]);
  }
  free(v);
  ```


Short Break

- Stand up and stretch



Output

- printf(*format_str*, *v1*, …)
- Eg: `printf("Hello\n");`
  ```
  printf("x=%d",ans);
  printf("%d squared=%d",ans,ans*ans);
  ```
- Place holders:
  ```
  %d - int
  %u - unsigned int
  %c - char
  %f - float or double (or %e or %g)
  %s - string
  ```


Output

```
char* name="Pi"
double v=3.14159;
printf("%s=%f",name,v);
```

- Apart from `float/double`, it is a good idea to cast values to the correct type.
- For other placeholders, see documentation.

Input

- int scanf(*control_str*, *vars*)
- Eg: `r=scanf("%d %c %f %lf", &i, &ch, &fl, &db);`
  Reads an int into `i`, a char into `ch`, a float into `fl` and a double into `db`.
- Returns EOF or the number of vars successfully read.

Input

- You can put other chars into the control string.
  ```
  char name[11];
  r=scanf("my name is %10s",name); /* no & because name is already a pointer*/
  ```

Files

- **`FILE* f=fopen(name, mode)`** Returns NULL(0) on error. Mode="r" or "w".
- **`fprintf(FILE*, char*, …)`** and **`int fscanf(FILE*, char*, …)`** – same as before but with files.
- Predefined files: **`stdin`**, **`stdout`**, **`stderr`** (standard error)

Files

- **`int fclose(FILE*)`** be sure to close files when you are finished with them. Unix can be a lot more picky about this than windows.
- **`int fflush(FILE* )`** used to ensure output is actually written.

Switch statements

- Switch = multi-way decision
- Example:

```
switch (argc)
{
    case 1:     /* No argument supplied */
        debug=0;
        break;
    case 2:     /* One argument supplied */
        if(strcmp(argv[1], "-debug") == 0)
        {   /* First argument was -debug */
            debug=1;
            break;
        }
        /* else drops through */
    default: /* All other cases */
        printf("Usage: %s [-debug]\n", argv[0]);
        exit(1);
}
```


Switch statements (cont.)

- Statement syntax:

```
switch (expression)
{
    case const-expr:  statements
    case const-expr:  statements
    default:          statements
}
```

- **`default`** is optional
  - If omitted and no matching pattern, nothing executed
- Execution starts at matching expression
- Continues until break statement or end of switch
- Usually faster than **`if-else-if-else-if…`**
  - May be implemented with table lookup
- Expression must be integral type, can't compare strings!

break and continue statements

- Can be used inside loops to alter flow of control
- **break**
  - terminates execution of innermost **`while`**, **`do`**, **`for`** or **`switch`** statement
- **continue**
  - terminates execution of **body** of innermost **`while`**, **`do`** or **`for`** statement and transfers control to end of body
    - i.e. will perform loop again if conditional allows it
- Illustration in class

Unions

- Like a structure (struct), but can only contain *one* of its elements at a time
- Example:

```
union U {
    double d;
    char c[2];
    int i;
};
```

- Illustration in class
- Member access is as for structures
  - selection (.)
  - indirection (->) for pointers to unions
- Programmer has to keep track of which type is stored

Break

Useful UNIX

- indent
- dos2unix, unix2dos

Function Pointers

- Often useful to be able to dynamically choose the function to be called
  - e.g. instead of

```
if(i==1) {
    fnOne(…);
} else if (i==2) {
    fnTwo(…);
} else if (i==3) {
    fnThree(…);
} else …
```

  - use

```
void (*fnArray[NUM])();
/* Declares fnArray to be
** an array of pointers to
** functions which return
** void. */
fnArray[1] = fnOne;
fnArray[2] = fnTwo;
…
```

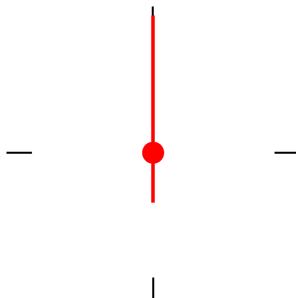
  - Then call with `fnArray[i](…);`
  - or equivalently `(*fnArray[i])(…);`
- Note: can combine declaration and initialisation

Some examples

- `int (*fp)(int,char*);`
  - Declares fp to be a pointer to a function which takes int and char* arguments and returns int
- `void (*fp2[10])(double);`
  - Declares fp2 to be an array (of size 10) of pointers to functions taking a double parameter and returning nothing
- `int (*fp3)();`
  - Declares fp3 to be a pointer to a function returning int.
  - Argument types unknown and won't be checked by the compiler.
    - Up to programmer to use this correctly.

Short Break

- Stand up and stretch



Type casting

- Often necessary to convert from one type to another
  - Some conversions happen automatically
    - e.g. function arguments, assignment operations, arithmetic expressions
    - Note: doesn't happen for functions like printf which support variable argument types
      - Up to programmer to get it right!
  - Other conversions require a **cast**
    - e.g. `dest = (type-name)source;`
  - Good to use an explicit cast anyway

Automatic conversions

To	From
Any real type	Any integer type
(void *)	(a) The constant 0 (b) Pointer to object (c) (void*)
Pointer to object	(a) The constant 0 (b) Pointer to compatible object (c) (void*)
Pointer to function	(a) The constant 0 (b) Pointer to compatible function

Things To Do This Week

- More C exercises
  - The more you do, the better you'll become
- Assignment One