Demo lecture slides

- Although I will not usually give slides for demo lectures, the first two demo lectures involve practice with things which you should really know from G51PRG
- Since I covered much of this in the G52CFJ (C and C++) module in previous years, I have included the slides from that module here, to remind people who may not attend the demo lectures, or forget what was said

Standard C Library Functions

- Part of the Standard C++ Library
- String functions : strcpy, strncpy, strcat, strcmp, …
- File access functions : fopen, fclose, fread, fwrite, …
- *Dynamic memory allocation : malloc/realloc/free memory*
- Mathematical functions : sin, cos, asin, sqrt, …
- Misc functions : e.g. random numbers

- You should **know** and use the standard library functions
- Documented in most C and C++ books
- Easy to find on the web, e.g.:
  `http://en.wikipedia.org/wiki/List_of_C_functions`
  `http://www.cplusplus.com/reference/clibrary/`
- **There is no excuse for not finding out**

strcpy

A function which uses pointers

The strcpy() function

```
char* strcpy(
        char* destination,
        char* source)
```

- Copy the characters from one string into another
  - Including the terminating zero
- ASSUMES that the destination is big enough (problem if it isn't!)

Valid destinations for strcpy?

- You can create **`char*s`** in many ways:

1) Just create a **`char*`**:

```
char* str1;
```

2) Pointing to string literals:

```
char* str2 = “Hello”;
```

3) From arrays:

```
char str3[6];
char str4[] = {‘H’,‘e’,‘l’,‘l’,‘o’,’\0’};
```

Question:

- **Which of these (str1, str2, str3, str4) are good destinations to copy a string to?**
  - **E.g.** `strcpy( str1, “Test” );`

Answer

1) Just create a char*:

```
char* str1;
```

   - Where does str1 point? So where would the string go?

2) From string literals:

```
char* str2 = “Hello”;
```

   - You should not write over a literal, see later
   - str2 is your variable, but the thing it points to is not

3) From arrays:

```
char str3[6];
char str4[] = {‘H’,‘e’,‘l’,‘l’,‘o’,’\0’};
```

   - This allocates enough memory to hold the array
   - These are the best options, as long as the array is big enough!

4) Allocate dynamic memory (see lecture 7, later)

   - Used if we don’t know the required size until runtime

Don’t get confused between these

   - Only 3 and 4 are (good) valid destinations for a strcpy

Input/output

Console/command line input

- Text input is buffered to allow editing
  - Only sent to program when you press ENTER
  - Program can then process input stream, using input functions
  - EOF means End Of File (end of input)
- Get the next character (or EOF):
  - `int getchar()`
- Get a string (up to newline character):
  - `gets( char* destination )`
  - Destination MUST be big enough! **How?**

The scanf() function : overview

- Provides the inverse (?) operation for **`printf`**
  - Reads formatted input, instead of formatting output
- Parameters have the same format but:
  - It needs to be able to set the values of the parameter variables
  - So you MUST pass pointers to the variables
- Examples:

```
printf( “%d %d %s”, int1, int2, string1 );
scanf( “%d %d %s”, &int1, &int2, string1 );
```

- Note: **`char*`** strings are already pointers
  - You don’t pass the address of a **`char*`**
  - From the **`char*`** the string itself can be changed

The control string

```
scanf( “Name: %s %d”, string1, &int1);
```

- Consists of three types of values:
  - Whitespace: Match any whitespace
  - Fields, labelled with `%` :
    - The letter which follows specifies the type, e.g.:
    - `%s` read a string ***up to whitespace character***
    - `%d` read a number
    - `%c` read a character (or multiple chars)
  - Other character: must be matched exactly, if matching fails then the read operation fails
- Returns number of fields matched

The letters for **`scanf()`**

`%d` or **`%i`**	Decimal integer
`%c`	Character
`%s`	String, to whitespace (space/return/…)
`%u`	Unsigned decimal integer
`%x`	Unsigned hexadecimal number
`%f`	Floating point number
`%e`	Scientific notation

You should experiment with these
They are VERY similar to the **`printf()`** ones
There is more to **`scanf()`** and **`printf()`**
See the docs (and **`fscanf()`**, **`sscanf()`**)

File access functions

File access

- The operating system provides methods for:
  - Opening a file: read or read&write? binary/text?
  - Reading from a file (binary or text?)
  - Writing to a file (binary or text?)
  - Moving around within a file (if appropriate)
  - Closing an open file
- Different operating systems may allow file access in different ways
- The C library functions wrap up the operating system calls

The C libraries : File access

- Hides the implementation details
- Provides a ***platform independent*** way to refer to files
- In a **`FILE`** structure
  - **`fopen()`** : Gives you a pointer to a **`FILE`** structure that it creates when file is opened
  - You do **NOT** need to know the format of **`FILE`**
  - Do NOT assume the format – it can vary
  - Just pass back the pointer it gives you
- Adds ***buffering***, so multiple writes happen at once (usually much faster)

FILE* Functions

- Open a file: (returns a **`FILE*`** pointer to use)

```
FILE* pFile = fopen("Filepath", <type>)
```

- Close an open file:

```
int fclose( FILE* pFile )
```

- Flush the write buffer of a file:

```
fflush( FILE* pFile );
```

  - Remember that files are buffered

- Read text: **`fgetc(), fgets(), fscanf()`**
- Write text: **`fputc(), fputs(), fprintf()`**
- Read binary: **`fread()`**
- Write binary: **`fwrite()`**

fopen()

- "Open a file for me and give me a way to refer to it":

  `FILE* pFile = fopen("Filepath", <type>)`

  – Example types **`"r"`** read, **`"w"`** write, **`"a"`** append

  ```
  FILE* pRead = fopen("input.txt", "r");
  FILE* pWrite = fopen("output.txt", "w");
  FILE* pAppend = fopen("append.txt", "a");
  ```

  – Other types:

  `"r+"` read/write (must exist)

  `"w+"` create empty file for read/write

  `"a+"` append to and read from existing file

  – Add '**`b`**' to type for binary (avoids some conversions)

- See documentation for details, for example:

  http://www.cplusplus.com/reference/clibrary/cstdio/fopen.html

Example : **fopen()**

```
FILE* pfileInput = fopen( pInputFileName,
            iType == 0 ? "rb" : "r" );
FILE* pfileOutput = fopen( pOutputFileName,
            iType == 0 ? "wb" : "w" );

if ( pfileInput == NULL )
{
   printf( "Unable to open input file : %s\n",
   pInputFileName );
   return 2;
}

if ( pfileOutput == NULL )
{
   printf( "Unable to open output file : %s\n",
   pOutputFileName );
   fclose(pfileInput);
   return 3;
}
```

Types for sample:
0 = binary
1 = scanf/printf
2 = fgetc/fputc
3 = fgets, fputs

Example : **`fclose()`**

Close the files at the end

```
fclose( pfileOutput );

fclose( pfileInput );
```

Example : **fread()**, **fwrite()**

Usually for reading binary data

Parameters:
Destination/source
Size of an element
Number of elements
FILE*

```
char buffer[1024];

while ( !feof(pfileInput) )
{
  int iNumberRead = fread( buffer,
     1, 1024, pfileInput );
  fwrite( buffer, iNumberRead, 1,
     pfileOutput );
}
```

Example : **fgetc()**, **fputc()**

For control, read a character at a time

```
int iChar;

while ( !feof(pfileInput) )
{
     iChar = fgetc(pfileInput);
     if ( iChar != EOF )
          fputc(iChar, pfileOutput );
}
```

Example : **fgets()**, **fputs()**

Or read an entire line (up to and including \n)

```
char buffer[1024];

while ( !feof(pfileInput) )
{
  if ( fgets( buffer, 1024,
         pfileInput ) != NULL )
    fputs( buffer, pfileOutput );
}
```

Including final '\0'

Example : **`fscanf()`**, **`fprintf()`**

Read a string, up to whitespace (not only \n!)

```
char buffer[1024];

while ( !feof(pfileInput) )
{
  if ( fscanf( pfileInput,
          "%1023s", buffer ) > 0 )
     fprintf( pfileOutput, "%s\n",
                buffer );
}
```

Note: it also adds a zero at the end! Buffer size 1024 Reads 1023 chars

Useful for reading multiple fields which are separated by whitespace (tab/space?) or for reading numbers

Moving around in a file…

- **`int fseek ( FILE* stream, long offset, int origin )`**
  - Move read/write position in file

  `origin` is a constant meaning one of:

  - ‘current position’, ‘start of file’ or ‘end of file’
- **`long ftell ( FILE* stream )`**
  - Ask where current read/write position is
- **`void rewind ( FILE* rewind )`**
  - Go to beginning of file

`stdin`, `stdout` and `stderr`

- Normal input and output can be accessed using the file functions
- **`stdin`** is standard input stream
- **`stdout`** is standard output stream
- **`stderr`** is the error output stream
- Use these as a **`FILE*`** in file functions
  - Just read/write, do not open/close them
- Defined in **`stdio.h`**

sscanf and sprintf

- More **`printf`** and **`scanf`** family members:
- **`sscanf()`** takes a string as first parameter

```
int sscanf(char* str, char* format, ...);
```

  – Read from string instead of stream (**`stdin`** or a file)

```
sscanf( "Adam 34", "%s %d", pName, &iAge );
```

- **`sprintf()`** takes a string as first parameter

```
int sprintf(char* str, char* format, ... )
```

  – Write to string instead of stream (**`stdout`** or a file)

  – **Ensure that it is big enough!**

```
sprintf( string, "%s %d", pName, iAge );
```

- **`sprintf()`** is useful for formatting text in a string