The C Standard Library

1 Logical Operators

- C has logical operators `!`, `&&` and `||`, in decreasing order of precedence
- If `a` is `0` then `!a` is `1` and `!a` is `0` otherwise
- `a&&b` is `1` if both `a` and `b` are non-zero and is `0` otherwise
- `a||b` is `0` if both `a` and `b` are `0` and is `1` otherwise
- How can you use logical operators to change an `int` to `0` if it is false and `1` if it is true?
- What is the value of `!1||0&&1`?
- The operators `&&` and `||` are short-circuiting:

```c
#include<stdio.h>
int yes(int x);
int no(int x);
int main(){
  if (yes(1)||yes(2)) printf("First thing\n");
  if (yes(3)&&yes(4)) printf("Second thing\n");
  if (no(5)||no(6)) printf("Third thing\n");
  if (no(7)&&no(8)) printf("Fourth thing\n");
  return 0;
}
int yes(int x){
  printf("%d\n",x);
  return 1;
}
int no(int x){
  printf("%d\n",x);
  return 0;
}
```

2 Classifying Characters

- C has various functions to classify different sorts of characters. These functions are declared in `ctype.h`

```c
#include<ctype.h>
int isdigit(int c);
```

- This function returns a non-zero value if the character `c` is a digit (`'0'-'9'`) and zero otherwise
- `c` must be either in the range of an `unsigned char` or equal to `EOF` (otherwise behavioour is undefined)
- Why does `c` have type `int` not `char`?
- How could we implement it?

3 Classifying Characters

- `int isalnum(int c);` alphanumeric
- `int isalpha(int c);` alphabetic character
- `int isblank(int c);` space or tab character
- `int iscntrl(int c);` control character (`0x00-0x1F` and `0x7F`)
- `int isgraph(int c);` printable character other than space

- `int islower(int c);` lowercase character
- `int isprint(int c);` printable character including space
- `int ispunct(int c);` printable character other than space or an alphanumeric character
- `int isspace(int c);` whitespace character (`' '`, `'\f'`, `'\n'`, `'\r'`, `'\t'` or `'\v'`)
- `int isupper(int c);` uppercase character
- `int isxdigit(int c);` hexadecimal digit

4 More Input/Output `stdio.h`

- We've previously seen `printf()` and `scanf()`, along with the variants `fprintf()`, `sprintf()`, `fscanf()` and `sscanf()`.
- `sprintf()` puts a formatted string into the array pointed to by `str`

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

- `snprintf()` does the same, but puts at most `size-1` characters into the array. Why might you want this?

```
int snprintf(char *str, size_t size,
    const char *format, ...);
```

- Aside: variants of these functions with bounds checking were introduced as an optional feature in C11 (see Annex K of the C standard): `scanf_s()`, `printf_s()` etc. They work in subtely different ways to the normal variants and **NO** compilers implement them entirely correctly. These functions are likely to be deprecated/removed in the future, so avoid using them.

5 More Input/Output `stdio.h`

- `feof()` returns a non-zero value if the end of file has been reached and zero if not. This only happens if you have tried to read **past** the last character in the file, not if you have only read the last character.

```
int feof(FILE *stream);
```

- `fgets()` reads in at most `size-1` characters from the file stream and stores them in the `char` array `s`, null-terminated.

```
char *fgets(char *s, int size, FILE *stream);
```

- `fgets()` keeps reading until it either reaches a newline `'\n'` (which will be included in the string `s`) or it reaches the end of the file.
- Returns `s` on success and `NULL` on failure (if end-of-file has been reached and no bytes are read, this counts as failure)

6 More Input/Output `stdio.h`

- `fputs()` writes every character from the null-terminated string `s` into the file stream `stream`

```
int fputs(const char *s, FILE *stream);
```

- `puts()` prints `s` to the screen **followed by a newline** `'\n'`

```
int puts(const char *s);
```

- `fputc()` writes the character `c` (cast to an `unsigned char`) to the file stream

```
int fputc(int c, FILE *stream);
```

- `putchar()` writes the character `c` to the screen

```
int putchar(int c);
```

- Aside: printing to the screen is often buffered. Once you are done writing, it is often good to print a newline, at which point the buffer will be flushed.

7 More Input/Output `stdio.h`

- `fgetc()` gets a single character from the input stream (i.e. file) and returns it as an `unsigned char` cast to an `int` or `EOF` on error.

```
int fgetc(FILE *stream);
```

- `getchar()` works equivalently to `fgetc(stdin)`

```
int getchar(void);
```

- `ungetc()` lets you push back one character onto the stream. This will be the first character read on future reads from the stream. You are only guaranteed to be able to push back one character, but it does **NOT** have to be the character you read from the stream.

```
int ungetc(int c, FILE *stream);
```

8 Random Numbers (`stdlib.h`)

- `rand()` generates random numbers between `0` and `RAND_MAX`
- `srand()` sets the seed of the random number generator. If `srand()` is given the same seed, it should produce the same sequence of random numbers from `rand()`
- The random-number generator is not cryptographically secure!
- Typical practice is to use the current time as the seed:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

...
srand(time(NULL));
int random_number=rand();
```