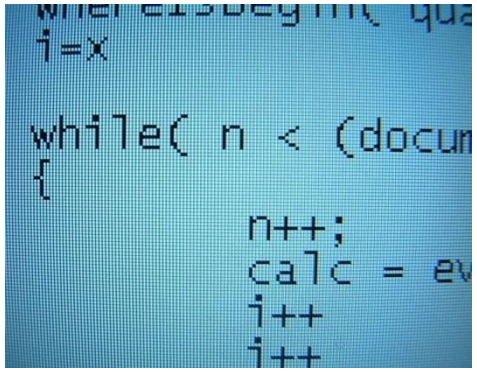
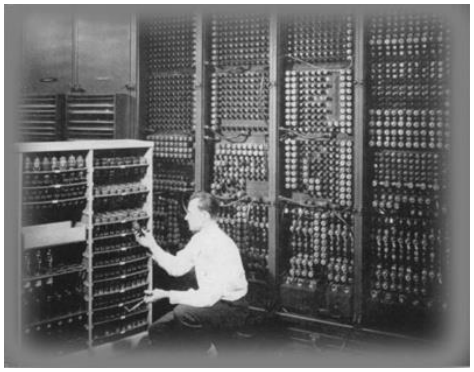
University of Reading





CS1PR16

Functions and Recursion

Learning Objectives

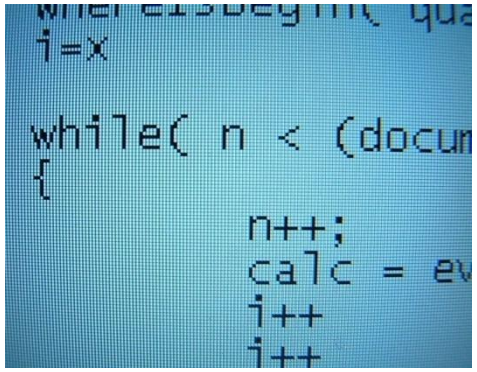
University of Reading

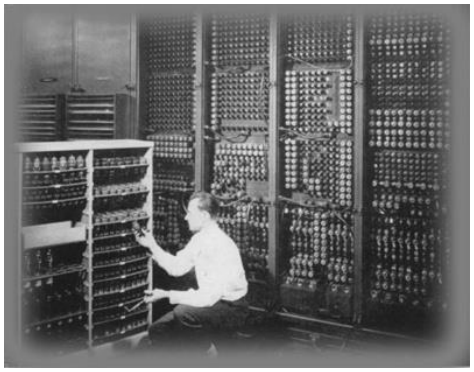
- Describing the principle of recursion and functional programming
- Define relevant terms in the concept of recursion/functions
- Contrast recursion and iteration
- Construct programs modularly from small pieces called 'functions'
- Name common functions available in the C library
- Create new functions in C

Outline

University of Reading

- Procedural programming and the principle of recursion
- Functions in C
- Libraries
- Examples from the C standard library





University of Reading

Procedural Programming

Introduction

University of Reading

- **C is a procedural programming language**
- **Procedural programming**

  *Based on the concept of the **procedure call**. Procedures, also known as routines, subroutines, or **functions**, simply contain a series of **computational steps** to be carried out. Any given procedure might **be called at any point during a program's execution**, including by other procedures or itself.*

  [Wikipedia]

- Mathematical function: Example: **f**(x, y, z) = g(x) + 2y + h(x, z)
- Functions provide us with a more manageable way to create programs
  - Divide and conquer: split the problem into smaller subproblems
  - Construct a program from smaller pieces or components (modules)
  - In a C program, modules are called '**libraries**' with '**functions**'

Functions: An Example

University of Reading

The following C-Code shows an example of a function

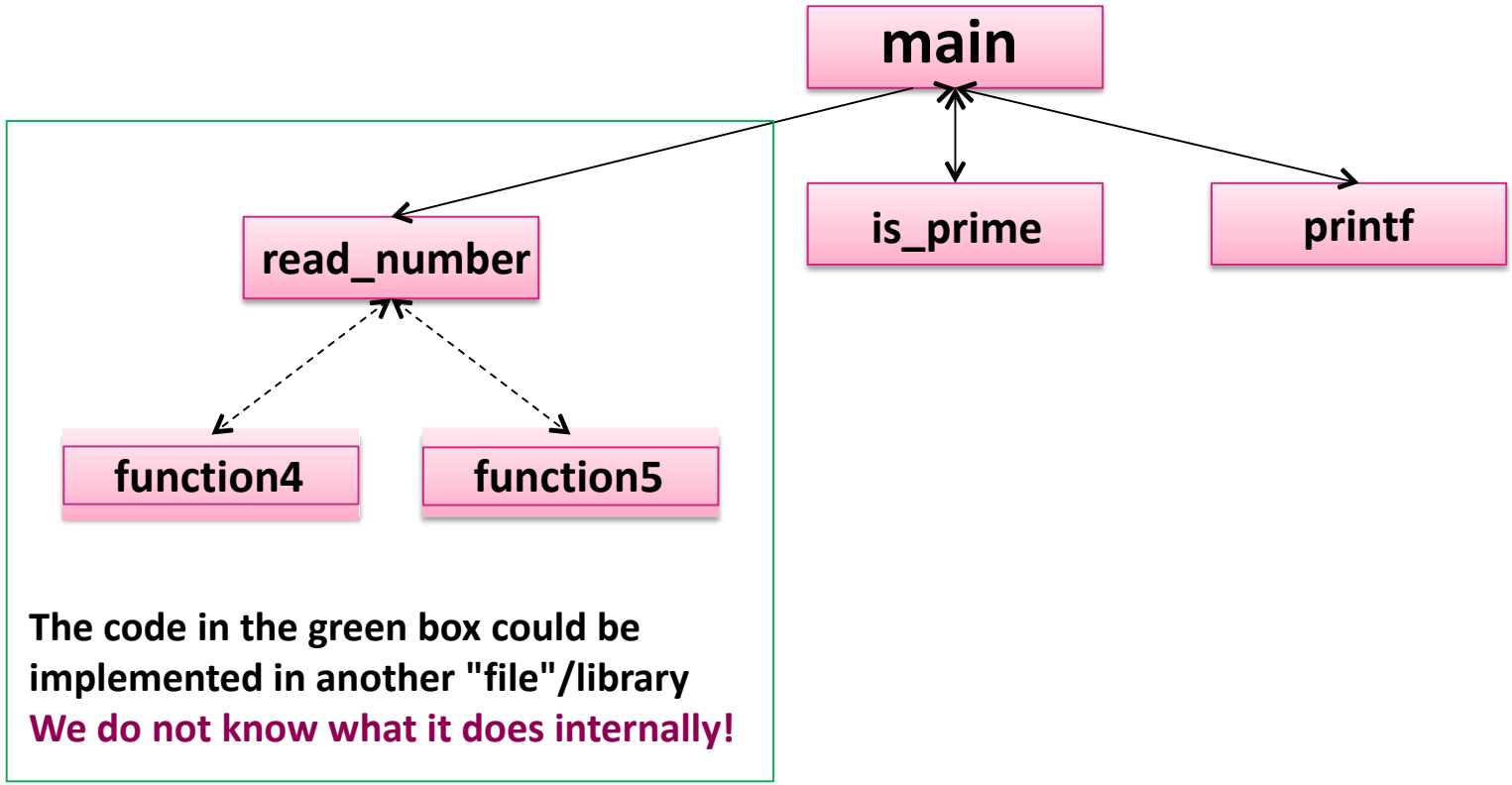
```
bool is_prime(int number){ // this is a function definition
        for(int i = 2; i < number; i++){
                if( number % i == 0) return FALSE;
        }
        return TRUE;
}

int main(){// main is also a function
        int number = read_number(); // use the function!
        if( is_prime(number) ){
                printf("%d is prime\n", number);
        }else{
                printf("%d is NOT prime\n", number);
        }
        return 0;
}
```

Remember each statement is processed in sequence

Functions

University of Reading

Function calls show an hierarchical relationship:



Recursion

University of Reading

- So far, we know **iteration**: technique to repeat a block of statements for a (defined) number of repetitions
- *Recursion is a **method** of solving a problem where the solution depends on solutions to **smaller instances of the same problem**.*

[Wikipedia]

- Recursion is one of the central ideas of computer science
- It originates from Math
  - Example of a recursive function definition in math
    - f(x) = f(x-1) * x (if x > 1)
    - f(x) = 1 (otherwise)
  - What does this function compute?

Note: The execution order is not defined

Example: The Factorial

University of Reading

- The example computes the factorial
  - f(5) = 5*f(4) = 5*4*f(3) = 5*4*3*f(2)= 5*4*3*2*f(1)=5*4*3*2*1

- How does a C program look like to compute the factorial?

```
int f(int x){
        if(x == 1) return 1;
        return x * f(x-1);
}
```

- We can also use the ternary operator:

```
int f(int x){
        return (x == 1) ? 1 : ( x * f(x-1) );
}
```

Recursion vs Iteration

University of Reading

- Recursion and iteration are equally expressive
  - Recursion can be replaced by iteration
  - Iteration can be replaced by recursion

- Sometimes one or the other expression is more appropriate
  - In terms of readability, easiness to write, or performance!
  - Benefit of recursion is that the termination can be proven

- The factorial written using iteration:

```
int f(int x){
        int result = 1;
        for(int i = x; x > 1; i--){
                result *= i;
        }
        return result;
}
```

The execution order is fixed and sequentially!

University of Reading

Functional Programming

*Functional programming languages **remove** or at least **deemphasize the imperative elements** of procedural programming.*

*Whereas procedural languages model execution of the program as a **sequence of imperative commands** that may **implicitly alter shared state**, functional programming languages model execution as the evaluation of complex expressions that only depend on each other in terms of arguments and return values.*

*For this reason, functional programs can have a **free order of code execution**, and the languages may offer little control over the order in which various parts of the program are executed.*

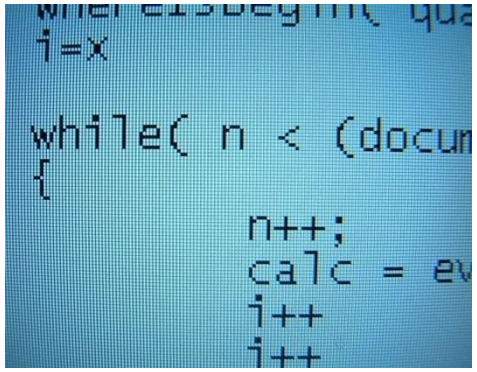
[Wikipedia]

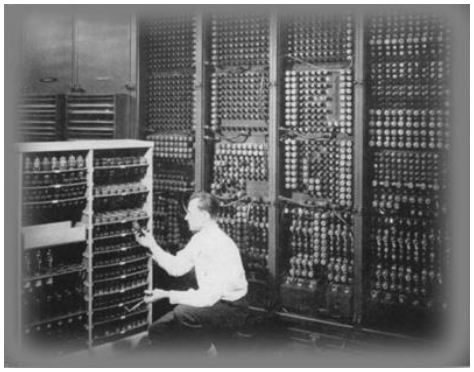
- ***More advanced features of functional programming in the Spring term!***

The Reason we use Functions

University of Reading

- Allow us to use the same code multiple times
  - Minimize replication and allow for code reuse!
    - Never reinvent the wheel, use existing functionalities
  - Save time
  - Only have to write code once (a boilerplate/"template")
- Make code simpler to understand
  - Each piece is more manageable than a complete program
  - Hide implementation details (and complexity)
  - Functions provide their own scope isolating the context (variables)
- Can share/distribute/sell code in Libraries
  - Community effort: ***Stand on the shoulders of giants***
  - Must think about the right "interface" to use the functions, though!
    - What does f(x, y, z) mean? Semantics?





Functions in C

Functions in C

University of Reading

- Two components:
  - Declaration: the function header, tells the compiler how the interface looks like
  - Definition: is a declaration that includes the implementation (function body)
- **Function declaration**
  - The interface: identifier for the function, **the arguments**, the return type

    <function-declaration> = <return-type> <identifier>({<declaration>})

    Example: int testfunc(int a, int b); // a and b are arguments, int is return type
  - Often called the "**function prototype**"
  - Tell the compiler how the function looks like as C checks the correct usage
  - Must happen before the first use
- **Function definition**: Implements the function

  <function-definition> = <function-declaration>{

  <compound-statement>

  }
- Functions cannot access data from the caller if it isn't an argument!
  - We will discuss the scope later!

Function Definition

University of Reading

- **The typical pattern for defining a function is:**

Arguments

```
<return-type> <name>({<type> <identifier>}){
    // statements, the function body
    return value;
}
```

- **Example**
  - Declaration: `double sqrt(double x);`
  - Definition (note that the name of the identifier can differ)

```
double sqrt(double value){
    double ret = compute squareroot somehow from value;
    return ret;
}
```

- **Arguments transfer data to the function when it is called**
- **Return allows data to be passed out and finishes the function**

Arguments

University of Reading

- Arguments transfer data to the function when it is called
  - A function cannot access any data of the caller, but only the arguments
  - Argument's identifiers can be accessed inside the function as a "variable"
  - **Any data passed to a function is copied into a separate area of memory for the function being called**
- An argument consists of a data type and an identifier
  - Must declare a data type for each parameter
  - The compiler checks the correct usage of the data type
  - In a declaration, the identifier is **optional (as not needed for syntax check)**
    - **But typically helpful to understand what the function does**
- Arguments are sometimes called parameters

Calling (Invoking) a Function

University of Reading

- Our example function declaration

```
double sqrt(double x);
```

- Uses either variables or a value to pass to a function using parameters

```
double result;
double X = 10.5;
result = sqrt(10.5);
result = sqrt(X); // this leads to the same result
```

- Calling a function is an expression, arguments are too, thus this works:

```
sqrt(sqrt(10.5))*5.0;
```

- The functions are evaluated from innermost to outer

- If a function uses multiple parameters, split them using ','

```
printf("Hello %s", name);
```

Return Values

University of Reading

- A function can return data by 'returning' it using the `return` statement
  - This ends the function immediately
- The value returned must be of the type as the return type of the function

```
int some_function(){
        int result = 5;
        return result;
        //The following line is never executed
        result += result + 5;
}
```

- Can use the type void when declaring a function if you are not returning any data

```
void function_name();
```

- Void functions behave like a statement and not like an expression
  - They have no return data and can't be assigned to a variable

Typical Misconceptions

University of Reading

- Consider the function:

```
int add( int num1, int num2 ){
      int result = num1 + num2;
      return result;
}
```

- That is called by main as follows:

```
int main(){
      int num1 = 4;
      printf("%d\n", add(4*4, num1));
      return 0;
}
```

- What does the function add() see as values for the arguments?
  - Remember: **isolation**: the scope of caller and callee are different!
- When calling add(), what are the actual arguments?
  - **Actual arguments**: the expressions in the function invocation
  - **Num1 = ?**
  - **Num2 = ?**
- When should the expressions be evaluated?

Typical Misconceptions

University of Reading

- Consider the function:

```
int add( int num1, int num2 );
```

- That is called by main as follows:

```
int num1 = 4;
printf("%d\n", add(4*4, num1));
```

- **The compiler creates code that is evaluated before the function is called**
- When calling add(), the actual arguments are:
  - `4*4`
  - `num1`
- These are evaluated before calling the function and the function sees:
  - num1 = 16
  - num2 = 4

Function Declarations (Prototypes)

University of Reading

- Function declarations are sometimes necessary
  - Allows the use of a function before the *definition*
  - Libraries: splitting functions into header and source files

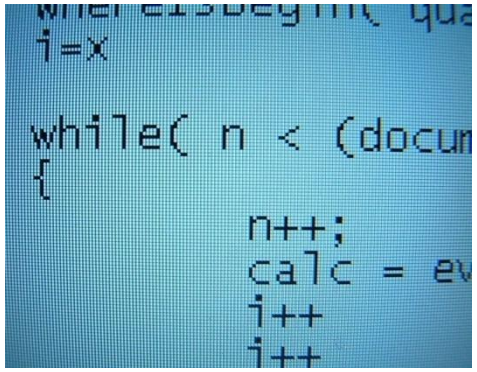
```
int add( int num1, int num2 ); // Prototype


int main(){
    int result = add( 4, 5 );
    printf("Sum of 4 + 5 = %d\n", result);
}

int add( int num1, int num2 ){ // Definition
      int result = num1 + num2;
      return result;
}
```

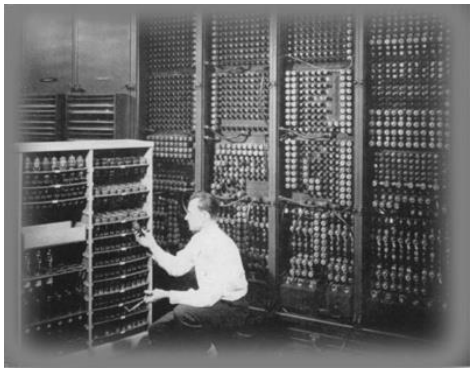
At this point the compiler must check that the function is called properly => needs declaration

Variable Number of Arguments

University of Reading

- So far, we could use a fixed number of arguments
  - How about printf()? We can call printf("X") or printf("%d", 4);
- C supports a variable number of arguments!
- However, the function must somehow infer how many!
  - The format specifiers in the first string in printf()
  - It leads to faulty code (at runtime!) missing a required argument
    - Example: printf("%d %d");
    - This example is syntactically correct but semantically wrong
- As part of this course, we will only implement functions with fixed parameters
  - Further information for implementing variable functions





Libraries

Libraries

University of Reading

- There are many libraries for any purpose
  - We'll use the C Standard Library as a part of this course
    - It is a standard across many compilers
- A library defines an API which is syntax and semantics

  *An **application program interface (API)** is a set of routines, protocols, and tools for building software applications.*

  Basically, an API specifies how software components should interact. [Wikipedia]
- The syntax of the API is defined in **header files** (typical extension ".h")
  - Contains the declarations of user-accessible functions
  - To use them in a program, use `#include <HEADER FILE NAME>`
    - That tells the compiler to read and paste the file at this location!
- The implementation of the API is typically in shared libraries
  - DLLs in Windows, Shared Objects in UNIX

To Include a File

University of Reading

- To Include another file, whether it's your own or one from other libraries, use the #include preprocessor instruction

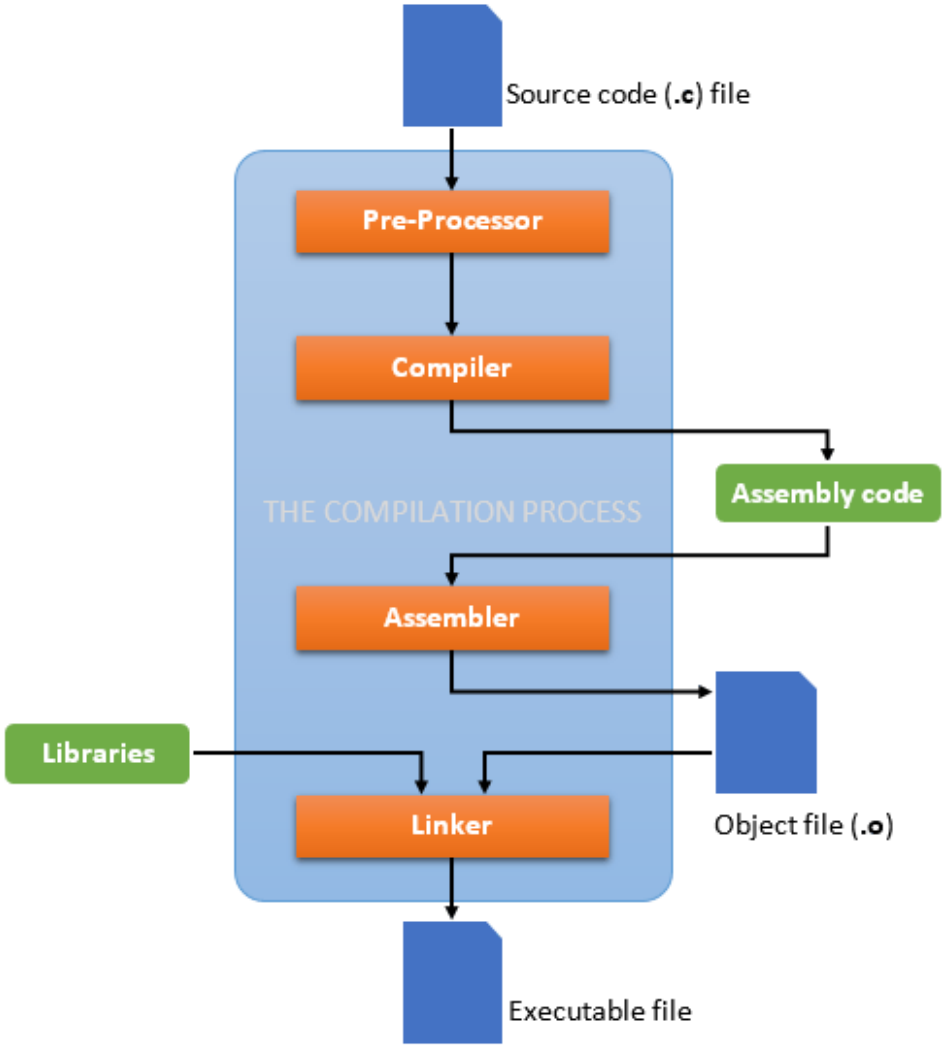
- These are used at the beginning of your file

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

- If you're using a file in the current directory, use ""

```
#include "my.h"
```

Workflow to Generate Executables



- This is the refined behavior!
- Assume we have written a program
  - stored inside a text file
- The pre-processor changes the text
  - includes text from header files as well
- The compiler reads the text
  - "parses" the syntax
  - translates the text to assembler (the lower-level language!)
    - that is **semantically identical!**
- The assembler translates the assembly
  - into binary machine code
  - stored in "object files"
- The linker combines object files
  - and links libraries to create
    - an executable program
    - or a library
- The executable can run on our system

Source: https://codeforwin.org/2017/08/c-compilation-process.html

Some of the C Standard Libraries

University of Reading

Standard Library Header	Explanation
<assert.h>	Contains macros and information for adding diagnostics
<ctype.h>	Contains functions that test characters for certain properties and convert between them e.g. lowercase to uppercase
<errno.h>	Defines macros that are useful for reporting error conditions
<float.h>	Contains the floating point size limits of the system
<limits.h>	Contains the integral size limits of the system
<locale.h>	Contains functions to help with localisation between regions
<math.h>	Contains many mathematical functions
<signal.h>	Contains functions and macros that handle various conditions that may arise during program execution
<stdarg.h>	Defines macros for dealing with a list of arguments/parameters to a function whose number and types are unknown
<stdio.h>	**Contains functions for standard input/output**
<stdlib.h>	Contains functions for many things such as conversion from text to numbers, memory allocation, random numbers.
<string.h>	Contains many string processing functions
<time.h>	Contains functions for manipulating time and date
<math.h>	**Contains math functions**

Details in https://www.gnu.org/software/libc/manual/pdf/libc.pdf

Group Work

University of Reading

Task:

Think about a library for math.

Define 2-3 function prototypes that you'd like to include

Time: 3 min

Share: 2 min

The Stdio Library

University of Reading

- Provides means to output text or binary data
  - to the screen (the console)
  - to a file
- Likewise, to read data from the user (keyboard) or a file
- Remember: a file is just a sequence of bytes
  - The input/output from the console can be considered as a file, too!
  - Unix philosophy: everything is a "file"
    - Standard input/output (console) is similar to reading from a file
- A file is usually stored on permanent storage!
  - Storage of data in variables and arrays are only temporary in memory

File Advice

University of Reading

- C imposes no file structure
  - No notion of "records" in a file
  - Programmer must design and/or provide a file structure
- Make sure not to overwrite existing files
- Designing a program with files in mind…
  - Programs may process no files, one file, or many files
  - Each file must have a unique name and should use its own variable

Views on Files: Byte Array vs Stream

University of Reading

- There are two common abstractions to I/O
- **As byte array**: a sequence of bytes (with given size)
  - Data has a **position** inside the byte array (e.g., position 50)
  - In C, a file ends with the end-of-file marker
- **As stream:** a continuous steady flow of data
  - One can read data from (or write data to) a stream
  - **The position in a "byte array" does not matter**
  - Particularly useful for input/output with the user
  - Can also provide a communication channel between programs
- Mode of accessing (which mode depends on user needs)
  - **Sequential access**: i.e., in order, the first bytes, then the next…
    - Maps well to stream concept
  - **Random access**: i.e., read byte 100, then read byte 24, …

Reinvestigate Printf()

University of Reading

- According to the man-page (run `man 3 printf` in Linux)

```
int printf(const char *format, ...);
```

- Format string

  The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments.

- We saw: %d to print integers, %f for float, %s for string …
- Some modifiers change the output
  - *Feel free to research this further…*
- Return value

  Upon successful return, these functions return the number of characters printed

Using printf()

University of Reading

- Printf() uses the directives % to identify a format operator that tells printf() how to convert the variable

Specifier	Description
%c	Character
%s	String of Characters
%d or %i	Signed Decimal Integer
%f	Floating Point Number
%E or %e	Scientific Notation e.g. 3.9265E+2
%x	Unsigned Hexadecimal Integer

- http://www.cplusplus.com/reference/cstdio/printf/ has a comprehensive collection of format operators including left or right justify and decimal precision

Streams Provided by Stdio

University of Reading

- C provides APIs for both, here we use the FILE streams
  - The FILE streams come with a new type: `FILE*`
  - Declare a variable as a FILE stream: `FILE *aFilePtr;`
- Open a file with `fopen`
  - Takes two arguments – file to open and file open mode
    - "w" for write, "r" for read, "b" for binary
  - If open fails, NULL is returned
  - `Example: aFilePtr = fopen("test.data", "bw");`
- Here are predefined streams (they don't need opening)
  - stdin - standard input (keyboard)
  - stdout - standard output (screen)
  - stderr - standard error (screen)

Various Open Modes

University of Reading

Mode	Description
`r`	Open a file for reading.
`w`	Create a file for writing. If the file already exists, discard the current contents.
`a`	Append; open or create a file for writing at end of file.
`r+`	Open a file for update (reading and writing).
`w+`	Create a file for update. If the file already exists, discard the current contents.
`a+`	Append; open or create a file for update; writing is done at the end of the file.
`rb`	Open a file for reading in binary mode.
`wb`	Create a file for writing in binary mode. If the file already exists, discard the current contents.
`ab`	Append; open or create a file for writing at end of file in binary mode.
`rb+`	Open a file for update (reading and writing) in binary mode.
`wb+`	Create a file for update in binary mode. If the file already exists, discard the current contents.
`ab+`	Append; open or create a file for update in binary mode; writing is done at the end of the file.
File open modes	

Read and Write

University of Reading

- The Standard Library includes various read/write functions to files
- The following functions can be used for characters
  - fgetc(<FILE>)
    - Reads one character from a file
    - Takes a FILE pointer as an argument
  - getchar `()` is equivalent to `fgetc(stdin)`
  - fputc(<Character>, <FILE>)
    - Writes one character to a file
    - Takes a `FILE` variable and a character to write as an argument
    - `fputc('a', stdout)` equivalent to `putchar('a')`

Read and Write

University of Reading

- Printing formatted text/parsing text
  - fprintf/fscanf: Equivalent to printf() and scanf()
  - The first parameter is a FILE variable
  - Otherwise, they act the same

```
fprintf( aFilePtr, "Hello %s", string );
fscanf( aFilePtr, "%s", string );
```

Finishing a Sequential Access File

University of Reading

- Closing the file when finished

  `fclose(<FILE>)`

  - Performed automatically when the program ends
  - Good practice to close files explicitly
- Detecting the end of a file when reading

  `feof( <FILE> )`

  - Returns true if end-of-file indicator (no more data to process) is set

Example Write

University of Reading

```
#include <stdio.h>

int main (){
   FILE *aFilePtr;
   aFilePtr = fopen("test.txt", "w");

   if (aFilePtr != NULL){
        printf("file created\n");
        fprintf(aFilePtr,"text in file");
        fclose(aFilePtr);
        return 0;
   }else{
        printf("unable to create file");
        return 1;
   }
}
```

Handling Errors

University of Reading

- Always check for errors!
- In the previous example, we checked if the result was NULL
  - That indicates an error according to the semantics of the function
- How do we identify which error it is?
- The headers errno.h and string.h help

```
#include <errno.h>
```

  - That makes the variable "`int errno`" accessible, which is the error number

```
#include <string.h>
```

  - Provides the function `char* strerror(int errno);`
  - Allows to convert the error to a human-readable string
- This style of error handling is typical for the standard library

Example Read (with Error Handling)

University of Reading

```
FILE *bFilePtr;
char text[SIZE];
int i;
bFilePtr = fopen("test.txt", "r");

if (bFilePtr != NULL){
     printf("file opened\n");
     for (i = 0; i < SIZE ; i++)
     {
            text[i] = fgetc(bFilePtr);
            if (EOF == text[i]) break;
            else printf("%c", text[i]);
     }
     fclose(bFilePtr);
     return 0;
}else{
     printf("unable to read file, error: %s\n", strerror(errno));
     return 1;
}
```

Reading and Writing Lines

University of Reading

- fgets() reads at most SIZE characters

```
while (fgets(text,SIZE,cFilePtr) != NULL)
```

- fputs() writes the string

```
FILE *dFilePtr;
char *textPtr= {"line 1\nline 2\netc"};
dFilePtr = fopen("lines.txt", "w");
if (dFilePtr !=NULL){
        fputs(textPtr,dFilePtr);
        fclose(dFilePtr);
        return 0;
}
```

Accessing Binary Data

University of Reading

- fwrite(<variable>, <elements>, <count>, <FILE>)
  - Transfer bytes from a location in memory to a file
- fread(<variable>, <elements>, <count>, <FILE>)
  - Transfer bytes from a file to a location in memory
- fseek(<position>)
  - Sets file position pointer to a specific location
- Example:

```
fwrite( &number, 4, 1, file );
```

  - `&number` – Location to transfer bytes from
  - `1` Number of bytes to transfer
  - `1` For arrays, number of elements to transfer
    - In this case, "one element" of an array is being transferred
  - `file` File to transfer to

Common <math.h> Functions

University of Reading

Function	Description	Example
sqrt(x)	Square Root	sqrt(900.0) is 30.0
exp(x)	Exponential Function	exp(1.0) is 2.718282
log (x)	Natural Logorithm	log(2.718282) is 1.0
log10(x)	Logarithm (base 10)	log10(1.0) is 0.0
fabs(x)	Absolute Value	fabs(-5.0) is 5.0
ceil(x)	Rounds Down	ceil(9.2) is 10.0
floor(x)	Rounds Up	floor(9.8) is 9
pow(x, y)	X raised to the power Y	pow(2,7) is 128.0
fmod(x, y)	Remainder of X/Y	fmod(13.657, 2.333) is 1.992
sin(x)	Trigonometric Sine	sin(0.0) is 0.0
cos(x)	Trigonometric Cosine	cos(0.0) is 1.0
tan(x)	Trigonometric Tangent	tan(0.0) is 0.0

Summary

University of Reading

- Recursion and iteration are not the same but can do the same
  - They are equivalently powerful
- Procedural programming means to structure code in "procedures"
- Functions allow to structure code, abstract and hide complexity
  - Don't confuse the scope of "variables"
- Declaration (needed for compiler) vs Definition (implementation)
- Libraries provide access to a vast amount of functions
  - Stdio for I/O (console input/output and file I/O)
  - We looked briefly at math functions