# Week/Lecture 1,2: Intro

C in general:

* medium-level, structured, procedural, imperative, programming language
  + Medium-level:
    - Compared to high-level languages (Bash, Python, Fortran, Java Haskell, etc), statements in C are “closer” to the native instructions of the computer
    - Compared to low-level (assembly) languages, there is not a precise one-to-one mapping to those instructions
  + Procedural:
    - Functions let you encapsulate common solution into self-contained units
    - The function can then be used at multiple locations in the program, or by other people with access to the code
  + Imperative:
    - Statements are commands that the computer is expected to obey
    - They are obeyed in the order they are written
* Relatively small – does basic things
* History
  + Developed in early 1970s at Bell Labs
  + Intended as a “systems programming language” for convenient low-access ???
  + First standard – ANSI C
  + Second standard (used in course) – C99 (in 1999)

Program Structure

* Programs contain data and instructions
* Start – 1+ standard lines which tell the linker where to find info
  + ex: #include <stdio.h>
* Consists of 1+ functions
* Function composed in terms of DECLARATIONS and STATEMENTS (which end with a semicolon)
* Expression statements (variables, operators, literals, …)
* Return code is either 0 (success) or non-zero (error-code – failure), can be used by calling script

Compiling

* Under Linux command-line compilers are most common
* gcc -o test myProg.c
  + Compiles a program contained in myProg.c
  + Tells the compiler to read, compile, and link myProg.c and write the executable code into the output file “test”
* ???

Literal values

* A literal – an explicit value written into the C source code itself
  + Ex: 99 (integer type), ‘a’ (character type), 5.67
* All values in C, including literals, have a **type** that defines what kind of value they are

Types

* The type of a value determines how the computer interprets the pattern of bits that represent it in memory
* ???
* Integer Types
  + Representations of integer values
  + Several different types for different memory use (and consequent max and min representable values)
  + Each type has a signed (default) and an unsigned (positive only) subtype
* Floating-Point types
  + Limited precision representations of real numbers
  + float (standard), double, long double
* ASCII Character Types
  + American Standard Code for Information Interexchange
  + Encodes 128 characters into 7-bit ints
  + 95 are printable (digits, lower and upper letters, special characters)
  + Non-printable:
    - ‘\’ is the escape character
    - \t – tab
    - \n – new line
    - \\ - \
* Other types
  + chars are single characters of text. Internally, they are represented as integers (indices into a table of symbols)
    - Ability to ‘add’ values to characters to get the next (‘a’+1=’b’)
  + void – no type

Computer memory

* The smallest bit of info stored by a computer is 1 or 0 – a bit. The natural number system is binary
* Byte – 8 bits
* Word – a complete unit of info (1+ bytes)
* Memory can be visualised as an array of memory cells with 8 columns
* Address- location of data in memory

Operators

* Built-in methods for manipulating values
* C provides a small set of ops for arithmetic, logical and utility purposes, most of which translate directly to ops that the CPU itself has built
* Have precedence rules, but parentheses override it
* Arithmetic
  + Work on numeric data types
  + Work within the type that is given (division of ints returns nearest int)
  + Mixing 2 types performs op as if both have the ‘widest’ type
* Relational
  + Compare 2 values and return true (1) or false (0)
  + Non-zero is taken to mean “true”
* Logical
  + Combine 2 true or false values using Boolean logic
  + As all non-zero integer values are considered true, these ops can be used on ints, too
    - && - and
    - || - or
    - ! – not
* Misc.
  + Number of bytes to represent value – sizeof()

Converting types

* Some operators automatically change the type of their operands so that they all match in type
* Cast – explicitly changing type
* (type)
* ???

# Lecture 2: Variables, Branching

Variables

* Provide a way to store values of a given type, with an associated identifier (variable name) to refer to the storage location
* Variable names may be of any length and can be made up of any combination of letter, digit and underscore characters which do not look like a literal value or a C keyword
* Because the identifier refers to the storage location,
* Declaring:
  + Before use
  + type and name
  + End with semicolon
  + Possible to declare multiple ones in-line
* Assign:
  + Possible in-line with declaration
* Operators:
  + assignment (=)
  + add to stored value (+=)
    - subtract, multiply, divide
  + increase (++), decrease (--) by 1
    - a++ == 3
      * compares, then increments
    - ++a ==3
      * increments, then compares
  + equality (==)

printf()

* Standard function included in <stdio.h>
* Form: printf(format, expression1, expression2, …)
* Format:
  + %d
    - Conversion with d – integer conversion (decimal)
  + %f
    - float numbers
  + %X (hex integer)
  + %c (char)
  + %s (char \* (string))
  + %e (float in exp. form)
  + Precision:
    - %6d (at least 6 chars wide)
    - %8.2f (float, 2 decimal digits)
    - %.10s (first 10 chars of a string)
  + Special chars:
    - \b backspace
    - \f form feed
    - \n newline
    - \t tab
    - \a audible alert

Conditional Branching

* Decision-making
* if:
  + if, else if, else
* switch, case
  + Tests a variable
  + **break** all cases
  + default (optional) – like “else”
    - **no break**

Loops

* while (test) {}
  + test performed first
* do {} while ()
  + executed first
* for(init ; test ; iter ) {}
  + init:
    - Counter variable
    - C99: Declare it there
  + test:
    - performed every time before execution (including after init)
  + iter:
    - Always after statements in the body
* Altering flow:
  + break
    - stop looping
  + continue
    - quit iteration, but continue to next iteration

# Lecture 3: Scope, Arrays, Strings

Scope

* A variable name is not always defined over the entire length of a program
* Scope – part of the program in which a variable name is defined
* File scope
  + defined outside of blocks
* Block scope
  + Defined inside a block
  + Can *shadow* variables of the same name in a wider scope

Allocation of memory

* The memory used to store values has a limited duration – lifetime
* Block scope variables are backed by automatic allocation (and deallocation). The storage used for the variable is allocated to it when the block it is declared in starts, and then automatically given back when we encounter the end of the block
* File scope variables declared with *static* keyword have *static* allocation. While the name associated with the storage may only have limited scope, ???

Arrays

* Storage in memory is all the values in arrays, one immediately after the other
* A list of values
* Declaring:
  + Type specifier
  + Name
  + Suffix number of elements (in [])
  + ex: int b[4]
  + Initialisation:
    - With curly brackets and comma-separated values
    - If initialised while declaring, no need for number of elements
      * double q[] = {0.1, 4.0}
* Using
  + Indexing
    - a[4] specifies 5th element
    - C does not check for array length (no “index out of bounds” error)
* Length – sizeof(array) / sizeof(element)
  + sizeof(foobar) returns size in bytes
* Comparing
  + compare each element using a loop
* Copying
  + copy in loop
* Multiple dimensions
  + type name[size1][size2]…

Strings

* Text is a sequence of characters
* But C has features
* Declaring
  + String literal – sequence of characters, enclosed in double-quotes
    - Equivalent to an initialiser list containing the same sequence of characters with an additional ‘\0’ char at the end
    - \0 used to mark the end of the string for string processing functions
  + char s[] = “A string”
    - equivalent to char s[] = {‘A’, ‘ ‘, ‘s’, ‘t’, ‘r’, ‘i’,
* Using
  + Cannot be operated on as a whole
    - NOT char str2[] = str1
  + Functions in #include <string.h>
    - strcmp(str1, str2)
      * Compares
      * Positive if str1>str2
      * 0 if equal
      * <0 if str1<str2
    - strlen(str)
      * Length
    - strcpy(str1, str2)
      * Copies str2 into str1
      * str1 must be large enough
  + All functions rely on strings ending in \0
* Reading with fgets()
  + fgets(name, size, stdin)
  + name
    - Name of a character array (string). The line, including end-of-line character, is read into this array
  + size
    - Reads until it gets a line complete with ending \n or reads size-1 chars. Convenient to use sizeof(name) as size.
  + stdin
    - File to read

# Lecture 4: Reading Strings, Structs, Functions

fgets() reads the whole line as a string. To read a line with a number:

* Use scanf() instead, which is analogous to printf() and allows to specify 1+ format conversions
* Continue to use fget() to read in line and then use sscanf() to re-read the string (internally) and break it up into parts

scanf()

* Direct equivalent to printf()
* Format: scanf(format, &variable1, &variable2, …)
  + & - address (memory)
  + ex: scanf(“%d%d%f”, &a, &b, &x) reads 2 integers and a floating point number
* %s conversion in scanf ignores leading blanks and reads until either \0 or first blank after non-blank charaacters, e.g., for “ ABCD EFG” %s will read “ABCD”

Reading numbers with fgets() and sscanf():

* Historically, scanf() was unreliable, so this was often used
* Format: sscanf(line, format, &var1, &var2, …)

Structured data types

* Declare a name for a struct
  + struct particle {  
     double x, y, z;  
     double vx, vy, vz;  
     char ptype;  
    };
  + Can use that name to make variables with requested internal structure
    - ex: struct particle electron;
    - array of particles:
      * struct particle p\_array[5];
* typedef:
  + Simplifies declaring structured types
  + Allows to specify a synonym for an existing type (including a struct type)
  + ex: typedef struct particle particle\_t  
    particle\_t electron;
    - Convention: end new typenames with a \_t
* Initialising:
  + Using curly brackets {}
  + If only values listed, order maintained, remaining get set to 0
  + Explicit assignment with a ‘.’
    - struct particle p = {3, 4, 5, .pytpe=’e’ };
    - equivalent to:
      * struct particle p = {3, 4, 5, 0, 0, 0, ‘e’};
* Accessing:
  + Attach element name to the variable with a joining ‘.’
  + p.x = 3.0;

Functions

* Encapsulate a chunk of code and give it a name
* Declaration
  + Function prototype
    - type signature, along with its name
  + Function body
    - Statements executes each time function is called
  + Forward declarations:
    - Function body does not have to follow prototype
    - Prototype has to appear in file scope, before the function is used
    - Later declaration of body must be provided
    - Forward declaration – “early” function prototype
* Definition
  + Only file scope variables and variables defined in the body are in scope
  + Parameters count as variables declared as block scope in the body
* Parameters, Return types
* Side effects
  + printf() returns number of characters it printed

pointers???

# Lecture 5: Pointers, CLI

Pointer – a type that points to a variable’s storage location

* Has a type which tells the compiler how to interpret
* Declaration
  + \* prefix
  + Can point to NULL – useful
* Pointing at a variable
  + & operator before variable gives its location
* Getting a value from a pointer
  + \* provides the value located at the memory address
    - **dereferencing**
* Passing to functions
  + pass with &
  + refer in function with \*
* Dangers
  + Can accidentally change other addresses
  + C provides no guarantees about the type
* Segmentation faults
  + When you attempt to reads/write a part of memory you are not allowed to access
  + What to check:
    - in bounds array access
* Arrays
  + The bare name of an array can be treated as a pointer
  + Possible to pass an array to a function with a pointer
    - Should pass the size because the function doesn’t know it
* Structs
  + “->” syntax

Command Line Arguments

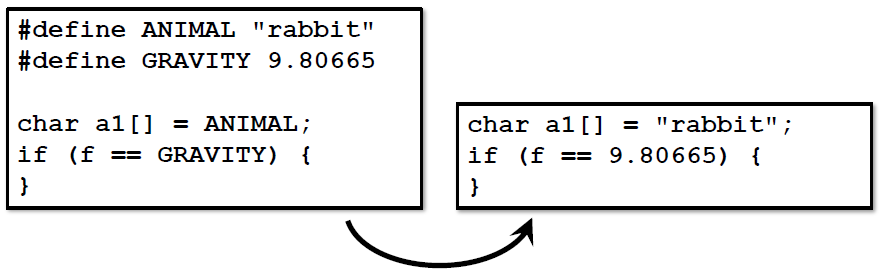
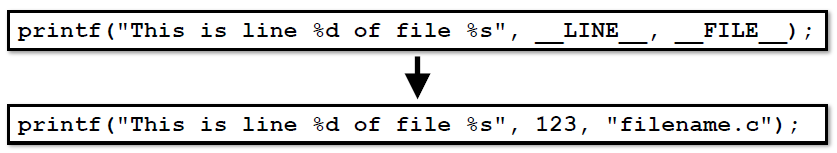
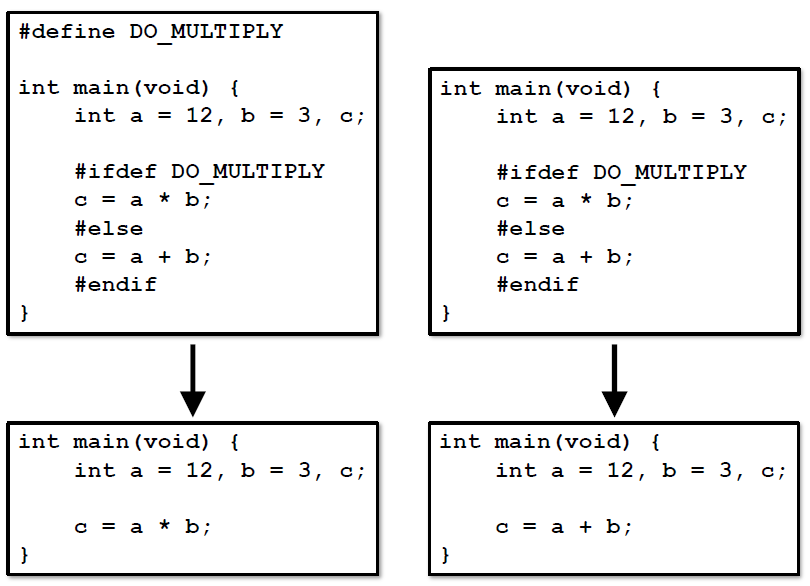
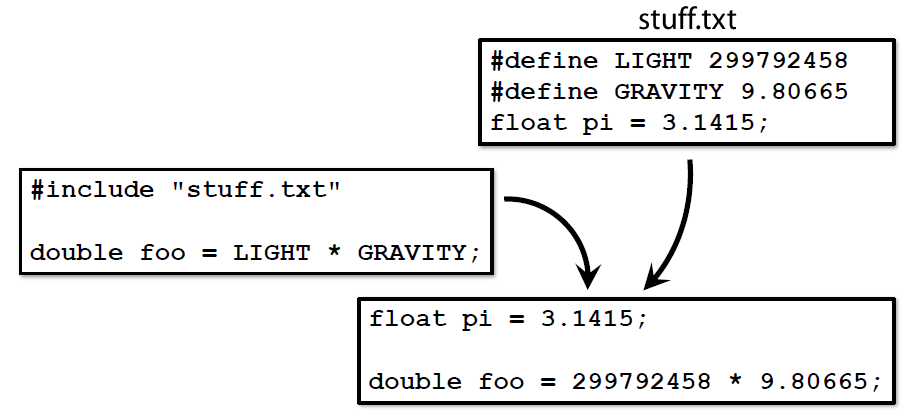
* main function ways to write:
  + int main(void)
  + int main(int argc, char \* argv[])
    - argc – argument count (number of)
    - argv – an array of strings of each argument

strcmp()

* compares 2 strings
* 0 if equal
* negative if first is smaller
* positive if first is bigger

# Lecture 6: Build Process

Pre-processing

* Prepares source code for compiler
* Performs text manipulation on source code
  + Replaces some text with other text
  + Insert/remove text based on the result of a conditional test
  + Include test from another file
* Output: C source code
* Commands (**directives**) start with **#**
  + #define
    - replaces text with something else
      * similar to “find and replace”
    - Replaces a single word **macro** with the provided text throughout the rest of the file (text within “ “ is not replaces)
    - 
    - \_\_LINE\_\_ will be replaces with number of the line on which it appears
    - \_\_FILE\_\_ - name of file
    - \_\_DATE\_\_ and \_\_TIME\_\_
    - 
  + #ifdef
    - User-defined macros to control whether a piece of text is included in the source code
    - #ifdef … #else … #endif
    - 
    - Negation - #ifndef
  + #include
    - Copies all text from the named file and inserts it in place of the directive
    - Directives in the included text will also be processed
    - 
    - To search for a file in standard system locations, enclose in angled brackets <>
    - To first search for a file locally, enclose filename in inverted commas (can include path)

Header files (.h)

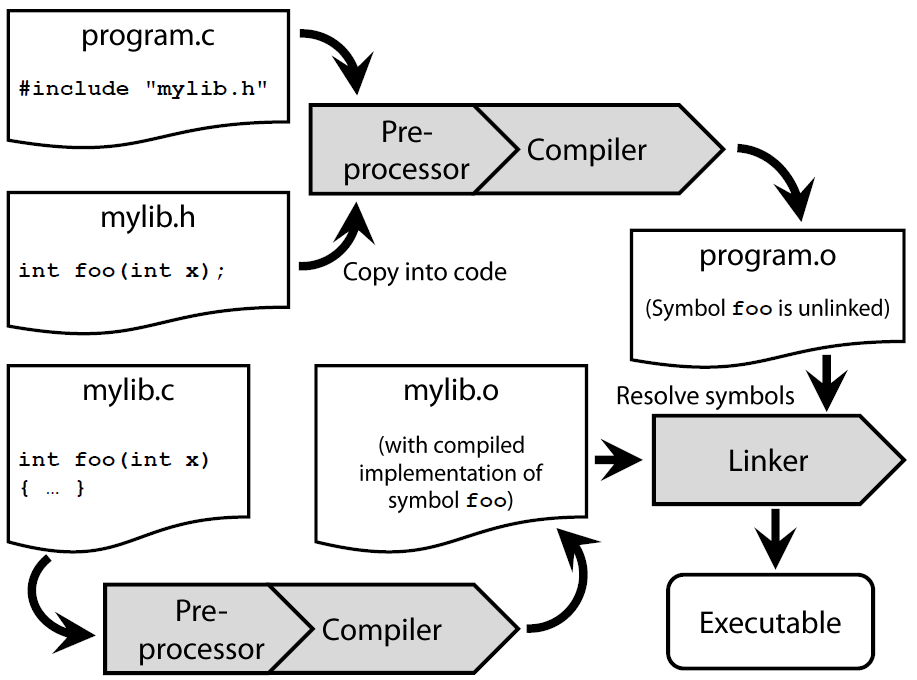
* To allow code in 1 source file to use functions and values from another source file, add
  + function prototypes for functions defined in other files
  + common constants
  + etc
* Linker joins up all code at the end
* Contains common declarations and exists purely to be used in #include directives
* Contain normal C source code and have .h file extension by convention
* For each .c file which contains code to be shared, write a .h file with the function prototypes
* #include guards
  + Can’t declare function (or name) in C more than once in same file
  + During first time the header is included, define it; on subsequent occasions, skip header

Compilation

* Takes C source code and translates it into machine code
* Source code is broken down into syntactic units called **tokens**
* Tokens are then parsed to construct an **abstract syntax tree** (an abstract representation of the program)
* This representation is converted (via assembly code) to **machine code**
* Items like file-scope variables and functions are labelled with a name (**symbol**)
* This machine code, along with a table of all symbols used in the file, is called the **object code** (with file extension .o)
* These symbols are used later to join up bits of code from different sources
* Compile only object code with **-c** argument with **gcc**
* Optimisation
  + Can tell compiler to optimise the code, which might include reordering instructions
  + **-O**
  + Can slow down compilation, use **-O2** for a good balance
  + Logic is preserved, but underlying implementation may change
  + Different flags
    - -O0 – turn off optimisation
    - -O1 – run faster, don’t think too hard
    - -O2 (recommended) –run even faster, reorder bits if you like but don’t make exe bigger
    - -O3 – run really fast, make exe bigger (if necessary), takes very long
    - -Os – as small as possible
    - -Ofast – fastest, doesn’t comply with standards, assumes things about floating-point numbers which aren’t necessarily true proceed with caution

Linker

* Joins up all object code from the compiler, mapping symbols to their corresponding representation
* How:
  + First, all object code is merged into a single file with a consolidated **symbol index** at the top
  + Before object code is linked, function calls are simply a note of the symbol corresponding to the function to be called
  + The linker looks up the symbol in the symbol index, and uses this info to replace it with a call to the matching function
  + A similar process happens for variables with file scope
* **-o**
* **gcc** uses **ld**
* Libraries
  + Provide a set of useful functions to others
  + #include 1+ header (.h) files to use function prototypes within them
  + Names ending in .a or .so (difference relates to how and when the code is linked)
  + Link library with NAME - **-lNAME** (at the end for gcc)
  + libc
    - C standard library
    - Split into multiple header files, including
      * stdio.h – I/O functions
      * string.h – string-handling
      * math.h – floating-point maths
    - gcc links with libc by default (except math.h - **-lm**)
  + Linker will search in standard locations for libraries
    - Add another location by **-L**



Useful gcc commands:

* Compile and link in one step:
  + **gcc -Wall -std=c99 -O2 -o OUTPUT\_NAME source1.c source2.c**
    - **-Wall** – turn on compiler warnings
    - **-std=c99** – enable C99 compliance (needed for certain language features)
    - **-O2** – turn on compiler optimisations
* Compile to object code only (create .o files but don’t link):
  + **gcc -Wall -O2 -c source1.c source2.c**
* Link object code:
  + **gcc -o OUTPUT\_NAME file1.o file2.o file3.o**
* Link with libraries
  + **gcc -o OUTPUT\_NAME file1.o file2.o -lLIBRARY\_NAME**

Summary

* Build process converts source code to machine code, and comprises 3 stages: pre-processor, compiler, linker
* The **pre-processor** prepares source code for compilation
  + It processes directives such as #include and #define
* The **compiler** converts prepared source code to object code
* The **linker** combines the program’s object code with object code from supporting libraries, producing the final exe
* **gcc** can do all this in 1 command, but it’s useful to break it into stages when working on large projects to avoid the need to rebuild everything following minor changes

# Lecture 7: File I/O

Tool – decdump

* **decdump -n 100 Figure\_1.png**
  + 100 – number of bytes to read
  + Figure\_1.png: file to read bytes from

Streaming IO

* Stream of data (characters/bytes) flowing between the program and the file itself
* Functions can take some data out of a stream (reading)
* Functions can also put some data into a stream (writing)
* A stream can connect to any file (or file-like things – stdio, stdin, stderr files for handling I/O)
* Text
  + fgets() takes characters from the stream until it reaches a newline
  + After fgets() returns, the stream no longer contains the characters it took
  + (If the stream represents a file, the file isn’t changed)
* Files
  + Open - fopen
    - **FILE \* fopen(char[] filename, char[] mode)**
      * **FILE \* fp = fopen(“file”, “r”);**
    - FILE – special struct which represents the (state of the) stream of data flowing to or from a file
    - fopen will try to open a file with the path and na,e, and return a pointer to a FILE representing (the stream connected to) it
    - Need to specify *how* to access file
    - **File Access Modes**
      * Second string passed to fopen
      * **r** – just read from start
        + r+ - read and write from start
      * **w** – overwrite the file, truncating first
        + w+ - read and write (but truncate first)
      * **a** – append from the end
        + a+ - read and append from the end
  + Buffers
    - Strictly, writes to a stream are not immediately reflected in the file
    - Instead, the writes are “queued up” in a buffer of things to be written
    - Periodically, the buffer is emptied and its contents committed
    - **int fflush(FILE \* fp);**
      * fflush makes sure that all the data written has been committed to file (flushes buffer immediately)
  + Closing
    - **int fclose(FILE \* fp);**
    - fclose does a flush, then removes connection between fp and file
      * If successful, returns 0
    - After closing, file pointer cannot be used for I/O until assigned a new file with an fopen
      * Sometimes good to explicitly set pointer to NULL after closing
  + Reading (text)
    - **char \* fgets(char \* string, int len, FILE \* file);**
      * fgets works as before, but file pointer replaces stdin
      * String pointed at must hold at least len chars
      * If fgets succeeds, it will return a pointer to string (as well as filling string with values up to a \n, \0 or end of file before len)
      * If it fails, returns NULL. If fgets returns NULL, the string’s contents may not be safe to use as a C string, so – check
      * fscanf(stdin, “%d is an integer\n”, &myint)
      * ==
      * scanf(“%d is an integer\n”, &myint)
    - **int fscanf(FILE \* file, char [] format, …)**
      * fscanf returns an int, which is the no. of variables it successfully put values into
      * fscanf returns EOF (a special constant) if it reaches the end of a file (and there’s nothing more to read), or there are other problems
      * Stream is left at point of first “non-matching” characters
  + Writing (text representation)
    - **int fputs(char [] string, FILE \* file);**
    - **int fprintf(FILE \* file, char [] format, …);**
    - Both return an int
      * fprintf – no. of characters it wrote
    - If failed, return EOF
    - Unlike puts, fputs does not add \n
    - fprintf(stdout, “%d is an integer\n”, myint)
    - ==
    - fputs(“%d is an integer\n”, myint)
  + Text Stream
    - Pro – human readable
    - Con – precision loss (floats), often uses more space than necessary
* Binary Stream
  + Accessing bytes in a file directly
    - Can be more efficient
    - Doesn’t lose precision
  + Pros – Exact representation of data
  + Con – Opaque, Less portable to different architectures
  + Opening, Closing
    - Still use fopen, fclose
    - Add “b” to file mode (e.g., “r” to “rb”)
  + Sometimes, conversion happens, so always explicitly turn on binary mode for binary I/O
  + Reading and Writing
    - **long fread(void \* start, long size, long num, FILE \* file);**
      * reads size\*num bytes from file, inserts them into memory starting at start
      * return how many it wrote
      * void \* - consider tart as a memory location, any pointer can be used here, regardless of type
      * **int a[5];  
        fread(a, sizeof(int),5,fp);**
    - **long fwrite(void \* start, long size, long num, FILE \* file)**
      * does the same thing, but takes from **start** and writes into **file**

# Lecture 8: Pseudo-Random Number Generators

Pseudo-Random numbers

* Sequences which have statistical properties of a random sequence
  + each value is hard to predict from the ones before it (**uncorrelated**)
  + the probability of any value is the same (**uniform**)
* PRNGs produces such sequences, but deterministically, from some kind of internal state

PRNGs

* Bad example:
  + Made by John Von Neumann in 1949
  + Each value in the sequence is the square of the middle digits of the value before it
* Flaws
  + Badly designed PRNGs can have unexpected behaviour for some seed values
    - Can generate a predictable (non-random) sequence
  + Even much more sophisticated PRNGs, which don’t have obvious flaws, can have subtle issues
    - Ex: IBM’s RANDU PRNG, which was widely used in the 1960s
    - Although the output looked random, it actually had some extremely strong correlations, which made the output untrustworthy
    - Many millions of hours of simulations had to be redone once this was discovered
* Advantages:
  + Seeds
    - Because a PRNG generates a sequence from an internal state, which is set by a seed value,
    - the same seed will always give the same sequence
    - Seeds allow comparison of the result of changes to code by running the same seed
    - (Mechanism for “seeded”)
  + PRNGs > HRNGs
    - Hardware Random Number Generators exist, but must gather truly random sources – radioactive decay, or shot noise in electronic circuits – which limits the rate at which they can work
    - PRNGs are usually faster than HRNGs, and reliability because they’re always available
    - HRNGs don’t have repeatable output
* In C
  + Functions for generation of pseudorandom sequences prototyped in stdlib.h (standard library)
    - Also defines constants, like RAND\_MAX (largest integer that the RNGs will return
  + srand()
    - sets the seed to the value passed
      * if not used, seed = 0
  + rand()
    - returns the next value in the pseudorandom sequence with the current seed
    - value between 0 and RAND\_MAX
  + Examples:
  + Continuous variable:
    - **return rand() / (double) RAND\_MAX;**
  + Discrete (6-sided die)
    - **return 1 + floor(rand() \* 6.0 / RAND\_MAX) ;**
  + Can seed from system time
    - **#include <time.h>**
    - **srand(time(NULL));**
      * time(NULL) returns current time in seconds since midnight 1 Jan 1970
  + rand() limitations
    - As srand() takes an unsigned int as a seed, there are only UINT\_MAX possible sequences that rand can produce (1 per seed)
    - The C Standard doesn’t require rand to have a particularly long sequence length (for portability). In some versions of the library, rand can have quite short repeat lengths, or fail some statistical tests
* Beyond rand()
  + Mersenne Twister and WELL
    - <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>
    - <http://www.iro.umontreal.ca/~panneton/WELLRNG.html>
  + For cryptography, even better ones exist (including HRNG)