# Week 0

## Lecture 0: Intro to Course

Assessment:

* 2 courseworks, 10% each
  + CW1 due 04/11 at 16:30 (already released)
  + CW2 due 25/11 at 16:30 (released mid-Nov)
  + Each assessment 8%, peer-review is 2%

Compilation of C:

* Do it on uni servers because of inconsistent compilers
* Connect with ssh/scp/PuTTY

# Week 1: Intro to Systems Programming

Basics:

* SP – activity of writing computer *system software*
  + Not application software
  + Examples of System Software:
    - OS, web browser, video games, scientific computing apps and libraries
  + System Software has (often) particular performance constraints (e.g., fast execution time, low memory consumption, low energy usage)
    - To achieve these performance constraints, SP languages allow for a more fine-grained control over the execution of programs
* C
  + Teaching of fundamentals:
    - **Memory** and **Computation** (fundamental resources)
    - Representation of **data structures** in memory and the role of **data types**
    - Techniques for **management** of computational resources
    - Reasoning about **concurrent** systems

History of SP languages:

* 1950s:
  + Software wasn’t distinguished between system and app
  + 1 app always used the entire machine
  + Fortran, LISP, COBOL
  + Grace Hopper wrote one of the first compilers
* until 1970s:
  + Sys software written in processor-specific assembly languages
* 1970s:
  + Dennis Ritchie and Ken Thompson wanted to port UNIX from the PDP-7 to the PDP-11
    - Wanted portable language, tried B, invented C as an **imperative** language supporting **structured programming**
* 1980s:
  + Bjarne Stroustrup wants new abstraction mechanisms, creates C++
    - Major influence is the first OOP language Simula
* 2010s:
  + New SP languages appear. Rust (2010) and Swift (2014) include many functional programming language features

Programming Paradigms

* Imperative: computations are sequences of statements that change the program’s state
  + x = 1; x = x + 1
* Structured: organisation of programs with subroutines and structured control flow constructs
  + sum = 0; for x in array{ sum += x }
* OOP: organises programs into objects containing data and encapsulate behaviour
  + animal = Dog(); animal.makeNoise()
* Functional: programs are mathematical functions and avoid explicit change of state
  + fib = lambda x, x\_1=1, x\_2=0: x\_2 if x == 0 else fib(x-1, x\_1 + x\_2, x\_1)

Compilation:

1. **Preprocessor** expands macros
   1. e.g., #include <stdio.h>
2. In the **compiler** stage, the source code is:
   1. parsed and turned into an intermediate representation
   2. used to generate machine-specific assembly code
   3. used to generate machine code in an object file
3. The **linker** combines multiple object files into an executable

# Week 2: Intro to C

Useful function:

* in math.h, fabs() calculates absolute floating point value

Compiler warnings:

* Errors and warnings are feedback from the compiler that there is something wrong with the program
* Error – impossible to compile
* Warning – unusual condition that may (and often does) indicate a problem
  + No good reason to ignore warnings
* -Werror – turns all warnings into errors making it impossible to compile a program with warnings
* -Wall – enables most compiler warnings

Makefile

* Most popular build system for C programs. Automates the process of compiling programs
  + Alternatives: Maven, Bazel, Ninja
* A **rule** explains how a **target** is built
* The lines below the first line are the **commands** of the rule and define how the target is built
* After the colon **dependencies** (files/other targets which are required to build target) are listed
* ‘make’ will execute the first rule of Makefile in the current directory, ‘make [target]’ will execute the rule to make the named target

Syntax of C

* Every program has exactly 1 **main** function, which is the entry point into the program
  + main function is special – if the terminal } is reached, 0 is automatically returned
  + 0 in return – successful execution to the environment executing the program
  + non-negative return value – unsuccessful execution
  + 2 versions:
    - main()
    - main(int argc, char\* argv[])
      * Allows processing of command-line arguments
* printf()
  + Defined in <stdio.h> and allows a formatted printing of values
  + 1st argument – **format string** containing special characters indicating the formatting of values
  + Starting from the 2nd argument, values to be printed
  + The number and order of special characters has to match
  + Special charactes:
    - %c – character (char)
    - %s – string (char\*)
    - %d – integer (int)
    - %f – floating-point number (float, double)
* Variables
  + Definition – **data type** (gives meaning), then **name** (**identifier**) and an **initialisation expression**
    - Declaring data type because C is **statically** typed
  + Every variable is stored at a constant location in memory that does not change over its lifetime
    - char/unsigned char – 1 byte
    - short/unsigned short – 2 bytes
    - int/unsigned int – 4 bytes
    - long/unsigned long – 4/8 bytes
    - long long/unsigned long long – 8 bytes
  + Boolean
    - 0 – false
    - any other value – true
    - Since C99, #include <stdbool.h>
      * bool a = true
      * %d
      * True and false are declared as macros with values 1 and 0
* Lexical Scoping
  + Each pair of curly braces {} is called a **block** and introduces a **lexical scope**
  + Variable names must be unique in the same lexical scope
  + For multiple variables with the same name, the variable declared in the **innermost** scope is used (if the usage is innermost)
  + Pre-processor macros
    - Dangerous as they don’t respect lexical scoping
    - #define ADD\_A(x) x + a
      * Should not define a function like this because a is not in the lexical scope of the definition
      * **Dynamic scoping** – should be avoided
* Variable lifetime
  + Automatic – variables declared locally in a block. Their lifetime ends at the end of the block
  + Static – variables declared with **static** or defined at file-level. Lifetime – entire execution of the program
  + Allocated – variables for which there needs to be an explicit request for memory to use dynamic memory allocation functions (e.g., malloc). Lifetime – managed by programmer

Stack-based Memory management

* When the lifetime of an **automatically** managed variable ends, its memory location is freed and can be reused
  + Every time a block is entered, put aside a location in memory for every variable declared in the block
  + Every time a block is exited, free the locations in memory for every variable declared in the block
* LIFO => **stack-based** memory management, area of memory – **stack**
* The other area of memory is the **heap** (manually managed)

Types combining multiple elements:

* Struct
  + A sequence of members of (potentially) different types
  + struct point {int x; int y;};
    - struct point p = {1, 2};
    - p.x., p.y
  + typedef struct { int x; int y; } point; (at top of file)
    - point p = {1, 2};
* Array
  + Multiple elements of the same type
  + int array[] = {1,2};
  + int array[2];
  + Must have fixed size (for automatic memory management)
* String
  + Array of characters
  + char greeting[] = “Hello world”;
  + char greeting[] = {“H”, “e”, “l”, “l”, “o”, “\0”}
  + Double quotes “” – (ASCII) string literal
  + Single quotes ‘’ – character literal
  + Strings are terminated by **\0** automatically added for string literals

Functions

* Definition
  + Function definition:
    - int max(int lhs, int rhs) {  
      if (lhs > rhs) { return lhs; } else { return rhs; }  
      }
  + Return type – data type of value that will be returned. If nothing returned, return type – **void**
  + Name – describes behaviour of function
  + Parameter list – data type and name of each parameter expected
  + Function body – block containing code executed when called
* Declaration vs Definition
  + Definition fully specifies the behaviour of the function
  + Declaration only specifies the interface describing how a function can be used:
    - int max(int lhs, int rhs);
  + Declarations are important for writing modular software as it allows to separate the interface (declaration) from the implementation (definition)
  + For calling a function, the compiler checks that the data types of the call expression and the function declaration match
  + The linker searches for the definition which might be in a different file/library (like printf)
* Call-by-value
  + Copies values into functions, not the pointers to them
  + Structs and Arrays
    - Structs are passed-by-value (changes are not visible outside of function)
    - For arrays, the **address of the first element** is passed, so changes to the array work globally

# Week 3: Memory & Pointers

Byte-Addressable Memory

* Every byte (8 bits) has its own address
* On a 64-bit architecture these addresses are 64-bit (8-byte) long
  + 64-bit architecture can address up to 16 exabytes
  + In practice, x86-64 only uses the lower 48 bits of an address, supporting up to 256 TB
* To modify a single bit, must load an entire byte, then (1 << 3)

Variables in Memory

* Every variable in C is stored at a constant location in memory (address) that does not change over its lifetime
* Depending on the size of the data type, the value of the variables will span multiple bytes in memory
* C address of variable with **address-of operator &**
* Size of variable: sizeof(x)

Pointers

* A variable storing the address of another variable
  + => Has its own address
  + Can change where a pointer points to
* **Dereference operator \*** allows accessing the value of the variable pointed to
* A pointer to a variable of data type t has data type t \*
* Every pointer has the same size
  + 64-bit arch – 8 bytes
* NULL
  + Use 0 or NULL to represent a pointer that points to nothing
  + NULL often represents an erroneous state, e.g., that an element wasn’t found in an array
  + Dereferencing a NULL pointer will crash the program
    - Tony Hoare – billion-dollar mistake
* const
  + Type qualifier
  + Enforced by compiler
  + Pointers can be const (unmodifiable) in 3 ways:
    - The pointer itself (address) cannot be changed (float \* const ptr)
    - The value pointed at cannot be changed (const float \* ptr)
    - Both value and pointer cannot be changed (const float \* const ptr)
* Arrays in Memory
  + 1D and 2D array values are stored the same way in memory
  + The name of the array refers to the **address of the first element** when assigned to a pointer variable
  + Can use array indexing notation on pointers (ptr[4])
  + Difference:
    - sizeof returns size of array and size of pointer
  + Cannot change a vector, only its elements
* Arithmetic
  + Possibilities
    - Add and subtract integer values to/from a pointer
    - Subtract 2 pointers from each other
    - Compare pointers
  + Take into account the size of the type the pointer is pointing to
  + ptr[i] == \*(ptr + i)
* Linked List with pointers and structs
  + struct node { char value, struct node \* next; }
  + Last node in the list has a next-pointer to NULL
  + Iteration:
    - while(ptr){  
       ptr = \*(ptr).value;  
      }
* BST
  + struct node { char value, struct node \* left, struct node \* right; }
  + s\_ptr->m == (\*s\_ptr).m
* Call-by-value
  + True for pointers that the value of the argument is copied into the parameter variable
  + For arrays, a pointer to the first element is copied instead of the entire array
  + The array is treated like a pointer => int param[] == int \* param
* main with command line arguments
  + int main(int argc, char\* argv[]){}
  + argc – number of command-line arguments
  + argv – an array of the command-line arguments as strings
  + A single string is represented as an array of characters: char \*
  + argv char \* [] can also be written char \* \*
* void \*
  + Generic pointer. Every pointer is automatically convertible to it
  + Only serves as an address pointing to **something**
  + Cannot access value as it’s not known
  + **Dereferencing a void pointer is forbidden**

Stack vs Heap as memory regions

* Automatic lifetime variables are stored in a stack
  + Size of variables needs to be known
* Heap – part of memory for dynamic memory allocation
* Shape

  Description automatically generated with medium confidence

Dynamic memory management

* Allocation
  + malloc() (in <stdlib.h>)
    - void\* malloc ( size\_t size );
    - Specify number of bytes to allocate
    - If malloc succeeds, a void-pointer to the first byte of the uninitialized memory is returned
    - If malloc fails, a NULL pointer is returned
* Deallocation
  + Memory allocated with malloc must be manually deallocated by calling free (exactly once) (in <stdlib.h>)
    - void free( void\* ptr );
  + If free is not called, allocated memory is **leaked**
* [examples of dynamic array and linked list in lecture]
* Returning a pointer to a local variable
  + Easy mistake
  + The pointer has a longer lifetime than the variable it’s pointing to

Function Pointers

* Memory also stores program code
* return type (\*)(argument\_types)
* [quick sort in lecture]

# Week 4: Memory Management & Ownership

Dynamic Memory management challenges

* Generic Binary Tree
  + Use generic pointer void \* to store an address in the node
  + No need to know what data is stored in the address
  + Use Function Pointers
* Responsibility to free up memory (free())
  + Call free exactly once for each address obtained with malloc()
  + Good practice: assign NULL to freed pointers
    - Does not prevent all double free errors
* Dangling pointers (point to freed locations)
* Memory leak
  + If free() is never called for a heap-allocated pointer

Ownership

* To organise memory management
* \* - identify a single entity which is responsible for managing a location in memory
* C does not enforce this
* C++ does enforce - RAII
  + Resource acquisition is initialisation
  + Idea – Tie management of resource to the lifetime of a variable on the stack
    - Allocation of resource (malloc) is done when variable is created
    - Deallocation (free) is done when variable is destroyed
  + Implemented by struct data types with 2 member functions (methods in Java):
    - Constructor
    - Destructor
  + Applicable to any resource management
    - Files, Network and Database connections, etc
* In Rust, ownership is enforced by the compiler, which makes **double free** and **dangling** **pointers** impossible and prevents most common cases of memory leaks

Modern Memory Management in C++

* Common data types for easy and non-leaking memory management
* Multiple values of same type on the heap: std::vector
* Single value with smart pointers: std::unique\_ptr (unique ownership) or std::share\_ptr (shared ownership)
  + Unique – default where a single variable is owning the value on the heap
    - [example of binary tree in lecture]
  + Shared – where not possible to identify a single owner
    - Common in multi-threaded apps
    - [example of DAG in lecture]
* Ownership transfer
  + std::move
  + Forbidden to access the value of a variable for which ownership has been transferred

# Week 5: Debugging, Development Tools, Peer Code Review

Bugs and finding them

* Software dev is not free of errors
* **Debuggers** – run a program in a controlled environment in which it’s possible to investigate its execution
  + For C/C++: GDB and LLDB

1. Compile program with -g
2. Start debugger and load program
   1. gdb –arg ./program
   2. lldb -- ./program
3. Inside debugger, start execution with **run**
4. Commands: <https://lldb.llvm.org/lldb-gdb.html>
   1. **bt** – backtrace
5. To exit, **quit**

* **Static analysis** tools – reason about a program’s behaviour without running it
* **Dynamic analysis** tools add instructions to a program to detect bugs at runtime

Common bugs:

* Segmentation fault
  + Raised by the hardware notifying OS that program has attempted to access a restricted area of memory
  + The OS will then immediately terminate the program
  + Most common causes:
    - Dereferencing a NULL pointer
    - Writing to ROM
    - Buffer overflow, i.e., accessing memory outside of an allocated buffer
    - Stack overflow, often triggered by a recursion without a base case

Breakpoints and GUI for debugging

* Breakpoints – points at which the execution is stopped and it’s possible to investigate the state of the execution (variables)
* Visual Studio Code/Atom have GUI for GDB/LLDB

Static Analysis

* Reasons about the code without executing it
* Compiler performs some static analysis in every compilation, e.g., type checking
* Good practice to enable all warnings (-Wall) and make warnings errors (-Werror)
* Simple:
  + Some static analysis is too expensive to perform in every build
    - Other static analysis enforces a particular coding guideline
  + Available with special flags/separate tools
* Clang
  + clang –analyze –analyzer-output html program.c
  + Generates report
  + clang-tidy
    - linter
    - Invoked like a compiler, accepting the same flags as clang specified after 2 dashes --
    - Checks can be enabled/disabled
    - Flexible
    - Allows enforcing coding guidelines and modernising code
    - Possible to extend with own checks

Dynamic Analysis Tools

* Need program to run and can only detect bugs which are encountered during the execution of a particular test input
* Clang calls them **sanitisers** 
  + Address Sanitiser – memory error detector
    - Detects:
      * out of bounds accesses
      * use after free
      * double free
    - By enabling this tool, clang will insert instructions in the program to monitor every single memory access
    - Slows down execution by about 2x. valgrind often by 20-100x
    - clang -fsanitize=address -fno-omit-frame-pointer -O1 -g -Wall -Werror program.c -o program
      * -fno-omit-frame-pointer – produces a readable stack
      * -O1 enables basic optimisations
    - Run program like normal
  + Memory Sanitiser – detector of uninitialized reads
    - Under active development and currently only available for Linux
  + Leak Sanitiser – memory leak detector
    - Detects memory which hasn’t been freed at the end of the program
    - Under active development and marked as experimental
  + Undefined Behaviour Sanitiser – detector of undefined behaviour
    - What is undefined behaviour?
      * Dereferencing a null pointer
      * Not guaranteed the program will crash
  + Thread Sanitiser – data race detector

# Week 7: Introduction to Concurrent Systems Programming

Concurrency

* \* - ability of different program parts (usually functions) to be executed simultaneously
* Methods:
  + **Shared memory** locations are read and modified to communicate between concurrent components. Requires **synchronisation** to ensure that communication happens safely
  + **Message passing** easier to understand, considered more robust
    - Ex: actor model in Erlang, CSP-style communication in Go
* vs Parallelism
  + Concurrency – programming paradigm, wherein threads are used typically for dealing with multiple asynchronous events from the environment, or for structuring program as a collection of interacting agents
    - Handling multiple different things simultaneously
  + Parallelism – about making programs go faster, no asynchronous stimuli to respond to
    - Doing multiple things at once

Processes vs Threads

* Processes:
  + Program executions
  + Multiple processes can be executed simultaneously
  + Each process has its own memory address space
* Threads
  + A thread of execution is an independent sequence of program instructions
  + Multiple threads can be executed simultaneously
* A process can have multiple threads sharing the same address space of the process
* Threads used to implement concurrent programs

Thread implementation

* In almost all programming languages
  + C: pthread
* Lifecycle:

1. Created
   1. starts executing a specified function
   2. with arguments
   3. given an identifier
2. Waits for another thread to terminate
3. Interrupt/kill another thread
4. Terminates by explicitly calling exit() or when its function terminates

* Communication between threads is by modifying the state of **shared variables**

POSIX Threads

* Most commonly used threading implementation for C
* <pthread.h> with compiler flag -lpthread
* Creating
  + int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(start\_routine)(void\*), void \*arg)
  + Arguments:
    - thread identifier (pointer to a memory location of type pthread\_t
    - Thread attributes, which set properties, such as scheduling policies/stack size. Passing NULL – default
    - A function pointer to the start routine. Takes a single argument of type void\* and returns a value of type void\*
    - Argument for start\_routine()
  + Returns 0 if successful, non-zero for fail
  + Passing pointers to and from start\_routine() allows the passing of arbitrary data
  + Requires care to ensure that the memory locations pointed to have appropriate lifetimes
* Waiting for another thread to Terminate
  + int pthread\_join(pthread\_t thread, void \*\*value\_ptr)
  + Arguments:
    - A thread identifier
    - A pointer to a memory location of type void\*
      * The return value of start\_routine passed to pthread\_create will be copied to this location
  + Returns 0 on success and non-zero otherwise

Mutual Exclusion

* Problem first identified by Dijsktra in 1965
  + Beginning of CS of concurrency
* Ex:
  + 2 threads using same variable and not seeing what other is doing with it
  + Singly linked list, removal of an element by one may be inconsistent with another thread
* **Critical region** – part of the code that updates some shared variable(s)/data structure
* Mutual exclusion executes **only one thread over a critical section**
* **Race condition** – result of a program execution depends on the order in which threads are executed
* One solution – Locks
  + Associate a lock (mutex) with each critical section
  + Before executing a critical section, a thread must **acquire** the lock
    - If another thread owns the lock, requesting thread is **blocked** until the owning thread releases the lock
    - If the lock is not owned, the requesting thread is granted ownership
  + On leaving the critical section, the thread must **release** the lock
    - On release, one of the threads (if any) blocked waiting for the lock will be granted ownership and continue execution
  + Ex: Bounded Buffer
  + Deadlock
    - Avoid deadlock if, when a thread becomes blocked, it releases the lock it owns
      * **Busy Waiting** (**Polling**)
      * Wastes CPU cycles and energy
    - Condition Variable
      * Better to block the thread seeking to acquire a lock, and wake it when it has a chance to proceed
      * Achieving using a condition variable cv
        + pthread\_cond\_wait(&cv, &m)

must be called with a locked mutex m

Releases the mutex and blocks the calling thread on cv

* + - * + pthread\_cond\_signal(&cv)

assigns mutex ownership to **one** of the threads blocked on cv, and wakes it

* + - * Condition variables are analogous to hardware signals/interrupts
        + pthread\_mutex\_lock(&m);  
          while (!cond) {  
           pthread\_cond\_wait(&cv, &m); }
        + Important to check condition again because it may have changed before the wakened thread is scheduled

Concurrency Reflection

* Concurrent Programming is Hard
  + Computation & Coordination (how threads should operate)
  + Coordination:
    - Types:
      * Partitioning
        + determining what parts of the computation should be separately evaluated, e.g., a thread to serve each request, to render each frame of a film
      * Placement
        + determining where threads should be executed
      * Communication
        + when to communicate and what data to send
      * Synchronisation
        + ensuring threads can cooperate without interference
    - Abstraction Levels
      * Low-level
        + locks, semaphores (higher)
      * Mid-level
        + Go, Monitors (Java threads), C++ Threads
      * High-level
        + OpenMP, Erlang
    - Options for Languages:
      * C++ has:
        + Thread libraries, e.g., POSIX
        + std threads
      * Java has:
        + Thread libraries, e.g., POSIX
        + Java threads
        + Executors

# Week 8: C++ Threads

#include <thread>

Compiling with -std=c++11 (or greater)

auto: Local Type Inference

* Compiler infers type based on initialisation expression
* **auto** i = 42;

Lambda expressions

* Functions without names
* “[](){}”
  + [] – captures
    - Variables from the surrounding scope that are passed to the lambda when it is created
    - Used for pointers to variables
  + () – parameters
    - When it’s called
    - By-value (copy of variables)
  + {} – body
* & Threads
  + Construct an std::thread object with a function pointer/lambda
  + Capturing allows using pointers

Mutual exclusion

* Commonly managed with a mutex and call lock() before entering and unlock() when leaving critical section
* In C++
  + Avoids forgetting by viewing locking mutex as **owning a resource** and applying the RAII (Resource Acquisition Is Initialisation) technique
  + #include <mutex>  
    std::mutex m;  
    void foo() {  
     std::unique\_lock<std::mutex> lock(m); //acquire mutex  
     …  
    } // releases mutex automatically (calls m.unlock(); )
  + [**EXAMPLE OF THREAD-SAFE LINKED LIST IN LECTURE**]

Condition Variables

* cv.wait(mutex, lambda expression)
* In C, waiting is while not true
* In C++, waiting for the condition to become true

[Bounded Buffer example in lecture]

Asynchronous Tasks

* std::async
  + Returns std::future
  + Future – handle representing a value not yet computed
  + future.get() retrieves value, but not before it has been computed
  + [example in lecture]
* Communication – std::promise
  + Promise – the channel that the task should write to
    - Reading – **future**, writing – **promise**
    - Possible to get std::future from std::promise (get\_future())
    - [example in lecture]
  + Waiting until all worker threads are ready before sending work – with std::promise<void> and std::future<T>::wait()
    - [example in lecture]

Tasks as First-Class Objects

* Important for thread scheduling, VMs, etc
* std::packaged\_task
  + task.get\_future()
  + [example in lecture]