WEEK 1

* **Systems Programming**
  + **Systems programming**- act of writing systems code
  + **Systems code**- constructs the environment that allows applications to execute in it.
    - Manages hardware and provides common services to applications
      * File storage, network service
  + **Systems Vs Applications Programming**

|  |  |
| --- | --- |
| Applications | Systems |
| Deals w/ abstractions/objects | Deals w/ registers, memory locations |
| Solves human problems | Controls and manages systems |
| Device/platform independence | Directly targets specific devices/systems |
| Java, C#, Python, garbage collection | C, x86 assembly, manual memory allocation |
| Big runtime systems | Resource constraint environment |

* + **Process**-instance of a program that is currently executing
    - Program → Compilation → Executable(binary) → Process
    - Importance of interaction between process and system
  + Layers
    - Layer 3: Application, User-level libraries
    - Layer 2: Device **Independent** OS code, File System, Network Stack
    - Layer 1: Device-**dependent** OS code, Device Drivers
    - Layer 0: Hardware
  + **Pros and Cons of C**

|  |  |
| --- | --- |
| Pros | Cons |
| Can operate in resource-constrained environments | No concept of object oriented programming |
| Very efficient and little runtime overhead | No language safety features |
| Allows for direct control over memory access and control flow | No automatic garbage collection of memory |
| Let’s programmer write parts directly in assembly langue |  |

* + **Data Types**

|  |  |  |
| --- | --- | --- |
| Integers | Reals | Text |
| short | float | char |
| int | double |  |
| long | long double |  |
| long long |  |  |

WEEK 2

* **Operators and Control Flow**
  + **Basic Program Structure**
    - Main should always return 0;
      * Tells the OS that the program exited normally
    - A nonzero return status indicates an error
      * The bigger the return value , the more severe the error
        + Status of 1 is used for simple errors

Missing file

Bad command line syntax

* + **Data Types**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Type | C Standard | 32 Bit | Windows 64 bit | Unix/Linux 64 bit |
| char | 8 bits | 8 bits | 8 bits | 8 bits |
| short | At least 16 bits | 16 bits | 16 bits | 16 bits |
| int | At least **16** bits | 32 bits | 32 bits | 32 bits |
| long | At least 32 bits | 32 bits | 32 bits | **64** bits |
| long long | At least 64 bits | 64 bits | 64 bits | 64 bits |

* + - sizeof(char)
      * Always = 1
    - sizeof(int)
      * depends on compiler/platform/operating system
    - sizeof(char\*) (pointer to char)
      * 4 on 32 bit systems
      * 8 on 64 bit systems
    - Range of char vs unsigned char
      * char: -128 ~ 127
      * unsigned char: 0 ~ 255
  + **Variable Declarations**
    - A variable must be defined in a declaration statement before it can be used
    - Tells complier how much memory to allocate
    - Must come before variable, usually at beginning of function
  + **Constants** 
    - Not part of the data
    - Numeric and character constants
    - Values that stay the same throughout program
  + **Operators**
    - Performs actual computation
    - Assignment
      * Stores expression to variable
    - Arithmetic
      * -, +, \*, / , %
    - Logical
      * && , || , !
    - Comparison
      * == , != , < , > , <= , >=
    - Bitwise
      * & , | , ^ , ~ , << , >>
    - Reference/ Dereference
      * & , \*
  + **Control Statements**
    - Iteration
      * Loops
        + While, do while, for
    - Selection
      * If/else, switch case
      * Selects one among many to execute
    - Jump
      * Jumps to specific location in code
      * Break, continue,
  + **Printf and Scanf**
    - Part of C library declared in stdio.h header
    - **Printf**
      * Outputs text to stdout stream connected to console
      * Printf(format, expression1, expression2);
        + Format is a string that can contain escape character and format specifiers
        + Format specifier: %[flags][width][.precision][length]specifier

Flags: 0: pads numbers with zeros instead of spaces

Width: min number of chars printed

If value is shorter will pad with space

Precision: for real #, number of digits to be printed after decimal point

Length: length of data to be printed

(none): int

hh: char

h: short

l: long

ll: long long

Specifiers

%d or %i: signed decimal int

%U: unsigned decimal int

%o: unsigned octal

%x or %X: unsigned hex

%c: char

%s: character string

%p: pointer address

%f or %F: decimal floating point

%e or %E: sci notation

%g or %G: shortest rep between f and e

* + - **Scanf**
      * Gets input from stdin steam connected to keyboard
      * int scanf(const char\* format, ….)
        + Format

Identical to printf except now it specifies the format of input stream

If input doesn’t match format, an error is returned and input not consumed

* + **Type Conversion**
    - When operation happen between two different types, the less precise type is converted to the more precise type
    - Integer division- result is truncated
  + A printf won’t print it it’s above a while that will not run
    - The stdout stream is by default buffered at line granularity or buffer size granularity
  + Know bitwise operations
* **Pointers, Arrays, and Strings**
  + **Pointers**
    - Every variable in a program has a memory location
      * Location can be accessed using & operator
    - **Pointer**- a variable whose value is the address of another variable
      * Declaration: type \*var-name;
    - Null Pointers
      * Set the initial value to null
      * Good practice
  + **Arrays**
    - Stores a sequence of variables of a given type in memory
    - Declaring: <type> <name> [ <length> ]
      * Tells compiler how much memory to reserve
    - Initializing arrays
      * Int num[3] = {1,2,3}
      * If initializer shorter than length, the rest is initialized to zero
      * Initialized arrays can still be modified
    - Accessing by index
      * value of array name iwht no index is the address of the first element
        + &nums[0]

Interchangeable with pointer

Int \*p nums;

\*p == nums[0];

* + - * Difference between array and pointer
        + Array has allocated memory statically bound to name at compile time
        + Nums cannot point to new address
    - Multidimensional Arrays
      * <type> <name> [<length1>] [<length2>]…
        + Int nums[2][3]
      * Initializing
        + Int nums[2][3] = {{0,1,2}, {3,4,5}};
      * **Conceptual layout**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Column 0 | Column 1 | Column 2 |
| Row 0 | 0 | 1 | 2 |
| Row 1 | 3 | 4 | 5 |

* + - * **Physical layout in linear memory**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| [0][0] | [0][1] | [0][2] | [1][0] | [1][1] | [1][2] |

* + - * **Accessing**
        + <type> <name> [<index1>][<index2>]…

Nums[1][2]

offset (1\*3+2) = 5 in linear memory

* + - pointer to an array
      * int(\*p)[3]
    - array of pointers
      * int \*p[3]
  + **Strings**
    - No string data type in C, must use character arrays
    - Sequences of characters
      * Consecutive in memory
      * Ends with null character (\0)
        + Character constant with ascii value 0
    - Character pointers can point to string
    - **const char** – points to an immutable string
    - String Functions

|  |  |
| --- | --- |
| Prototype | Description |
| Size\_t strlen(const char \*s); | Calculates the length of string s  Does not include terminating character |
| Int strcmp(const char \*s1, const char \*s2); | Compared string pointed to by str1 to string pointed to by str2  If return value <0 then str1 is less than str2  If return value >0 then str1 is greater than  If return value = 0 they are equal strings |
| Int strncomp(const char \*str1, const char \*str2, size\_t n) | Same as above except only compares the first (at most) n chars of str1 and str2 |
| char \*strcpy(char \*dest, const char \*src); | Copies string pointed to by src, including terminating character, to the array pointed to by destination |
| char \*strncpy(char \*dest, const char \*src, size\_t n); | Same as above except not more than n bytes of src are copied |
| char \*strcat(char \*dest, const char \*src); | Apprends src to the dest string overwriting the terminating char at the end of dest and then adds a termininating char |
| char \*strncat(char \*dest, const char \*src, size\_t n); | Same as above except it will use at most n chars from src |

* + - Try to use the “n” version to prevent buffer overruns

|  |  |
| --- | --- |
| Prototype | Description |
| char \*strchr(const char \*s, int c); | Returns a pointer to the first occurrence of the character c in the string s |
| char \*strstr(const char \*haystack, const char \*needle); | Finds first occurrence of substring needle in the string haystack |

* + - String Conversion Functions
      * String to number conversion functions declared in stdlib.h

|  |  |
| --- | --- |
| Prototype | Description |
| int atoi(const char \*nptr); | Converts string pointed to by nptr to int. |
| double atof(const char \*nptr); | Converts string pointed to by nptr to double. |

* + - * String formatting functions declared in <stdio.h>

|  |  |
| --- | --- |
| Prototype | Description |
| int sprintf(char \*str, const char \*format, ...); | Same as printf except instead of writing to stdout formatted string written to str |
| int snprintf(char \*str, size\_t size, const char \*format, ...); | Same as above except no more than size bytes are written to str |

WEEK 3

* **Functions**
  + **Procedural Languages/Procedural Programming**
    - Uses a **list of instructions** to tell the computer what to do step-by-step
    - Relies on **procedures** – routines or subroutines
    - Also referred to as **imperative programming**
    - Known as **top down** languages
    - It is intuitive
      * Step by step instructions on how you want the computer to do something
    - Functions are what makes C a procedural language
  + **Function**- name for a self-contained group of statements that performs a task
    - Can be executed by invoking or calling it
    - Code modularization(better readability)
    - Reusability
    - Implementing recursive algorithms
  + **Function Declaration**
    - <return type> <name? ( <parameter list> );
    - Declares function prototype
      * Type of the function
      * Consists of function name, return type, and params
      * Crucial for type checking
      * Crucial for generating correct memory allocations during function call
    - Must come before call if the definition doesn’t
    - Can be outside functions in a global scope or in another function in a local scope
    - Param names are optional and ignored by compiler
  + **Function Definition**
    - Consists of:
      * Function prototype
      * Local variable declarations
      * Statements
    - Must match function prototype in declaration exactly
    - Must return a value of the return type
    - Function can’t be defined inside another function
  + **Function Call**
    - <name> ( <argument list> );
    - Consists of:
      * Function name
      * Arguments
        + Expressions that evaluate to each respective type in param list
    - Compile error if number of arguments differ for number of params
    - If argument types differ from params, arguments are coerced into param types
    - All arguments are passed by value
      * Arguments are **copied** to parameters
      * Argument and parameter refer to different locations
  + **Passing Arguments by Value**
    - Function defines parameter
    - The calling code passes an argument
    - Function has access to the following locations:
      * Global variables
      * Local variables
      * Parameters
      * NOT the local variables or params of caller function
    - Arguments allow local variable and param values to be passed from **caller** function to **callee** function