TYPES: Typed programming language is a language that use types of data to define storing values in memory, which operations are admissible on values, defines the # of bytes available for storing and ranges of possible values. 4 arithmetic types are *char* (1byte)*, int* (short 2byte /long 4byte /long-long 8byte)*, float* (4byte)*, double* (long 8byte). Floating-point data store value into 3 distinct components (a sign, an exponent and a significant). Value ranges: range of *char, int* has limit by a #s, floating- point *(float, double)* limit by their component. integral types: **char** **int** floating-pointtypes: **float** **double** EXPRESSIONS: Evaluate exprs: arithmetic, relational, logical Arithmetic exprs consist of binary operands: integral (+,-,\*,/,%) and floating-point (all except %). Binary refer to (+,-,\*,/,%) and unary uses to assign a sign to # (positive / negative). Relational exprs: to evaluate a cond (1 true,0 false) They are ==, >, >=, <, <=, != (both integral and floating -point). Logical exprs: && (and), | | (or), ! (not true) ex. !(a && b) Shorthand assign’: same on integral and floating-point types. Binary operands are +=, -=, \*=, /=, %= (integral only). Unary operands are (++i, i++, --i, i--). Casting: to convert one data type to another (long double, double, float, long long, long, int, short, char). Ex (float) minutes. Mixed-type exprs: C ranks long double> double> float> long long>long> int>short>char. For operands with different types, assignment exprs: C will promote or narrow the right to the type on the left (ie, (float) cash = (int) change => *float* (promotion), if (int) change => (float) change => *int* (narrowing)). Arithmetic and relational exprs: C will promote the operand of lower type to a higher type before evaluating expr (ie 10 \* 1.0=10.0 ; 1.00 \* 1 = 1.00 ). Compound exprs: is an expr that contains an expr as one of its operands. C evaluates according to the *rules of precedence.* C operates on the one with highest rank first ( ++ -- (post fix) > ++ -- (prefix) > unary (+,-,&,!) > . LOGIC. Structured programming: sets of simple construct that has one entry point and one exit point. So that anyone can replace, modify or upgrade without affecting other parts in the program or causing bugs. A sequence is either a simple statement or a code block that enclosed in brackets { } to be run sequentially. Use pseudo-coding and flow-charts to achieve a clear, concise and better designed codes. Pseudo-code: is a set of shorthand notes in human language that shows in sequence of instructions to the solution. Flow-charts: is a set of conventional symbols connected by arrows to show the flow of control throughout the program (diamond for selection, rectangle for action, pentagon for iteration, oolong for start and exit). Selection constructs: 1.Optional path (do sth only when cond is true) 2. Alternative paths: has binary selections *(if-else*) and multiple selections *(if – else if – else*): {compound conds (*if (cond1 && cond2) – else*): use logical to form more than one conds. Case-by-case *(switch*): use when we have a constant value for each case} 3.Conditional expr: short way to do alternative path (if-else, 2 cases only). It has the form ( *condition ? operand1 : operand2 )*. If the cond is true, the expr evaluates operand1. If the cond is false, operand2 is run. Interation constructs: While: runs its sequence as long as the cond is true. Do-while: runs the codes at least once and will continue as long as cond is true. For: groups initialization, test cond and change together. Flags: flagging is a method of coding iteration within the single entry-and-exit rule of structured programming. Flags are variables that determine whether an iteration continues or stops. Nested Constructs: one logic construct within another. 2 types: Nested selections (if-else within if-else). Dangling else: will be associate with the innermost if (ie. If if else, the else will go with the 2nd else). Nested loops: a loop in another loop. An inner loop will run full, then the outer loop run and form a full cycle of the inner loop again. STYLES: good program is easy to read and to debug. Identifiers should be self-descriptive, meaningful, short, concise identifiers (variables, functions, naming in general!). Practice good layout (spacing, indentation, line length, braces, comments). Magic numbers: prefer to values that is not generated from the program (could be math constants, default values), name them (ex const pi 3.14 OR #define pi = 3.14). TESTING. 2 types of Errors: Syntactic errors are ones that breaking the rules of the language. Semantic errors: ones that fail to implement the intent and meaning (use = instead of ==, dangling else, mismatch data types). Testing techniques: black box tests: are data driven, internal logics are hidden, output alone decide the success or failure of tests. Large possible values to test require narrow down into equivalent classes with boundary values. An equivalent class is a set where test wise any member is as good as another other. Errors occurs often at boundaries. White box testing: a complimentary test to black box test, it is logic driving, all internal logics are visible, it is path- oriented. We run each possible path once using a flow graphs. WB test criteria: run every statement, edge coverage, (compound) conds check, path coverage, iteration coverage. Debugging techniques: use IPE or command-line debugger. Walkthrough tables: consist of 2 parts: a record of every change in the value of every program variable, a listing of the output. Sufficient form: tables that show data type, variables, address, initial values and values that changing while on. ARRAYS: list of variables of the same type, we use arrays to perform same instructions on multiples variables. A list is called *arrays* and the variables in array are elements, we refer to any element by its index starts from 0. Arrays consist of ordered set of elements of common type that are stored contiguously in memory (without gap). C compliers can’t check if an index is within array, programmers need to ensure that codes call or include with size of arrays. Define array != initialize array. Parallel arrays is a convenient way to store tabular info. One array holds the key, while the other hold values. The arrays are parallel because the elements at the same index hold data that are related (ie student – marks, product – price). STRUCTS. A structure type is a collection of data that are not necessarily same types. We use struct to define a group of variables as a single object. A type directs how C interpret stored data in memory: 2 types (char, int, float, double) types and derived type (everything else). Struct declaration: We like to declare a struct globally and store in a header file for easy modification, correction or changes. NOTE: there are no allocation of memory when we declare a struct! When we define an object, allocation of memory will take place. Initialization of struct is similar to a primitive type separate using { {a,b}, {b,c} } by groups. FUNCTIONS. Modular design is an approach in programming involves separating code into self-contained components that can be accessed multiple times from diff locations in a complete program. Modular design identifies which components can be developed separately. Each module consists a set of instructions that are related together. Design principles (in module): easy to upgrade, not too long, may be utilize more than once. For structure design (as a whole): each module has one entry and one exit point, highly cohesive, low coupling. Cohesion: describes the focus: highly cohesive module performs a single task and only that task (unrelated tasks should be left out). Coupling describes the degree of interrelatedness of a module with other modules. The less info passes between modules the better. A procedural programming language supports modular design through function syntax. Functions: transfer control between one another (or *call* other functions). Once it completes its task, it transfer control back to its *caller*. A function consists of a header and a body [ *type identifier (type parameter, …., type parameter) { //function body return x: }* ]. Type specifies the type of the return value or the function’s parameter. Identifier specifies the name of function. Parameter is a variable that holds data received from its caller. Special cases: 1.Void functions: a function that doesn’t return any values (return empty and can be omitted). 2.No parameters: a function that doesn’t receive data passes from caller. Parameters are null (ex. void alphabet (void) { // } ). 3.Main main() function is a to which the operating system transfers control after loading the program into RAM. Main() returns a value of *int* tyoe (value 0) to the operating system once it has completed execution. A caller function will transfer control to the called function. Once its done, it returns control to caller. To call a function, we use [ *identifier (argument, … argument)* ] in which identifier is the name of the function being called, argument specifies values to be pass for it to utilize. An argument may be a constant, a variable, or an expression. Numbers of argument should match the number of parameters in function header. Pass by value: a function will pass a copy of arguments to called function for them to work on. Mixing types: if arguments types don’t match the parameter when we declare, the complier will (promote or narrow) argument types to the ones match in declaration. Use walkthrough tables for a modular program is an extension where we include each function and its handled variables. POINTERS.C makes sure values passed to function cannot be change outside of the function itself, but in cases when we need to change the values, we can do this through pass by address. Every variable occupies a unique address in memory when the program is run (this change when program shuts off and reruns). A variable that holds an address is called a pointer. To store an address, we define a pointer type and assign address to that pointer (variable’s address holder). It takes form: [ *type \*identifier;* ] ie int \*apple (an *int* that hold address of variable named apple). The \* operator is called *dereferencing* or *indirection* operator. Pointer types (char\*, short\*, int\*, long\*, float\*, etc) allows C to know what type of data is stored at that address and how to evaluate them. C store address in 4 bytes of memory. NULL address: each pointer has a special value called its null value. We can initialize a pointer to NULL before address is known [ int \*p = NULL; ]. Parameters: a function can receive addresses in its parameters this is called *pass by address.* This will allows us to access the original values and change them from the function. Multiple Return Values: C function syntax only allows them to return of a single value. If program design requires a function to return more than one value, we do so through parameter pointers that hold the addresses of the variables. TEXT FILES. Files: are named area of secondary storage. They may be fragmented meaning stored at different non-contiguous location in memory. They do not necessarily occupy contiguous space on storage device. EOF marks end-of-file and typically has value of -1. Text format: a file holds info in 2 formats: text – readable and editable data and binary. Text chars are almost universal except for $ and ` chars. Sequential access: access a text file byte-by-byte. Connection: C connects a file through an object of FILE type.. The objects holds info about the file and keep tract of the nect position to be accessed. We use a F/AR/STRU: Scope: determines a variable visibility. Global scope: located out of all fn, avoid due to high coupling. Fn scope: located within fn header (parameters). Local scope: declare within fn body. Block scope: located within a block of code (if statement). Passing arrays: we on need the address of the start of the array. The name of an array without the brackets holds address of array (1st element). Use const to block changes. Passing structs: 2 ways: pass-by-value or pass-by-address. Arrow notation: arrow operator takes a pointer to an object on its left and a member identifier on its right (\*st.grade eqv st->grade). INPUT Fn: A *buffer* is a small region of memory that holds data temporarily and provides intermediate storage between a device and prog. Formatted input: scanf() exacts chars from input buffer until it has either interpreted and processed data to match all conversion specifiers in format string, leave the offended chars in buffer, in case of empty buffer, it will wait for more data from users. Conversion control: % (\*) width size conversion\_char. Ex: “%\*c%c”, will discard 1 char. “ %c” skip all whitespace first, “%d%\*c” swallow newline OUTPUT Fn: Buffering Output buffering lets a program continue executing without having to wait for the output device to finish displaying the characters it has received. The output buffer flushes if: it is full, it receives a newline (**\n**) character, the program terminates. Conversion control: % (\*) flags width .precision size conversion\_char. Flags (- left justifivation, 0-add leading 0). Special chars: printf("\\ \' \" %%\n"); TEXT FILES. Files: are named area of secondary storage. They may be fragmented meaning stored at different non-contiguous location in memory. They do not necessarily occupy contiguous space on storage device. EOF marks end-of-file and typically has value of -1. Text format: a file holds info in 2 formats: text – readable and editable data and binary. Text chars are almost universal except for $ and ` chars. Sequential access: access a text file byte-by byte. Fgets(fp)/fputc(ch, fp) put or print a single character to file, no format. Open a File: The most common connection modes are **"r"** - read from the file **"w"** - write to the file: if the file exists, truncate its contents and then write; if the file does not exist, create a new file and then write to that file **"a"** - write to the end of the file: if the file exists, append to the end of the file; if the file does not exist, create it and then write to it. The less common connection modes for text files are **"r+"** - opens the file for reading and possibly writing **"w+"** - opens the file for writing and possibly reading; if the file exists, truncates its contents and then writes to the file; if the file does not exist, creates a new file and then writes to that file **"a+"** - opens the file for writing to the end of the file and possibly reading; if the file exists, appends to the end of the file; if the file does not exist, creates it and then writes to the file.**Rewind rewind()**resets the record pointer in the **FILE** object the first byte in a file.  The next byte to be accessed by the object will be the first byte in the file. In other words, to jump to the beginning of a file, instead of disconnecting and re-connecting it, we simply rewind the file. **End of File feof()** indicates whether or not the caller attempted to read the end-of-file mark; that is, read beyond the last character in the file.  The prototype for this library function is **int feof(FILE \**fp*) feof()** returns false (0) if the caller has not attempted to read the end-of-file mark; true if the caller attempted to read the end-of-file mark. If the next byte to be read is the end-of-file mark, but the caller has not yet read the mark (that is, has only read the last character in the file), **feof()** returns false.  In other words, to receive true, the caller must have attempted to read the end-of-file mark at least once. **RECORDS** A record occupies a single line in a text file and holds all of the data associated with one chunk of information.  The record is a sequence of characters that ends with a record delimiter.  The typical record delimiter is the newline character (**\n**). **FIELDS** A field holds one element of information within a single record.  We separate adjacent fields within a record by a field delimiter. **TABLES** A table is a set of records in which each record contains the same number of fields. **DEFINITION (REVIEW)** A string is a **char** array with a special property: a terminator element follows the last meaningful character in the string.  We refer to this terminator as the *null* terminator and identify it by the escape sequence **'\0'**. The null terminator has the value 0 on any host platform. **FORMATTED STRING INPUT** The **scanf()** and **fscanf()** library functions support conversion specifiers particularly designed for character string input. These specifiers are: **%s** - whitespace delimited set. The **%s** conversion specifier: reads all characters *until the first whitespace character*, stores the characters read in the **char** array identified by the corresponding argument, stores the null terminator in the **char** array after accepting the last character, leaves the delimiting whitespace character and any subsequent characters in the input buffer; **%[ ]** - rule delimited set. The **%[ ]** conversion specifier accepts input consisting only of a set of pre-selected characters. The brackets contain the admissible and/or inadmissible characters. The symbol **^** prefaces the list of inadmissible characters. The symbol **-** identifies a range of characters in an inclusive set. For example, the **%[^\n]** conversion specifier reads all characters until the newline (**'\n'**) stores the characters read in the **char** array identified by the corresponding argument stores the null terminator in the **char** array after accepting the last character leaves the delimiting character (here, **'\n'**) in the input buffer. **Caution** Because **%[ ]** ignores leading whitespace, it cannot accept an empty string; that is, **%[^\n]** does not treat a **'\n'** in an otherwise empty input buffer as an empty string.  If the input buffer only contains**'\n'**, **scanf("%[^\n]", name)**, unlike **%s**, returns 0 and leaves **name** unchanged. **Qualifiers** Qualifiers on the **%s** specifier add detail control: **%20s** displays a string right-justified in a field of 20 **%-20s** displays a string left-justified in a field of 20 **%20.10s** displays the first 10 characters of a string right-justified in a field of 20 **%-20.10s** displays the first 10 characters of a string left-justified in a field of 20. The **puts()** and **fputs()** library functions output a character string to the standard or specified output device respectively. **fputs()** writes a null-terminated string to a file.  The prototype for **fputs()** is:**int fputs(const char \**str*, FILE \**fp*);** ***str*** receives the address of the string to be written and ***fp*** receives the address of the **FILE** object.  **fputs()** returns a non-negative value if successful; **EOF** in the event of an error. POINTERS/ARRAYS/STRUCTS: **Arrays:** The name of the array holds the address of the start of the array. We can store the address of an array in a separate pointer. **double a[10]; int i = 3; double \*p = a; p[i] = 5.4;** Note that **a** and **p** are interchangeable. **Passing a Part of an Array: display (&sku[1], n - 1); TWO DIMENSIONAL ARRAYS: *array*[ *row* ][ *column* ] Order**

The C language stores the elements of a two-dimensional array in *row-major order*: the first row, column-element by column-element, then the second row, column-element by column-element, then the third row, etc..

**Passing to a Function** We pass a two-dimensional array to a function in the same way that we pass a one-dimensional array.  We specify the name of the array as an argument in the function call.  The corresponding function parameter receives th evalue of this argument as the address of the array.  The parameter declaration identifies the array as two-dimensional by two pairs of brackets.  The parameter declaration includes the array's column dimension - the column dimension must be included. **Passing a Specific Row of an Array: *array*[*row*] SEARCHIN** Search algorithms finds the index of one or more array elements that satisfy a specified condition or set of conditions.  These algorithms work with key-value pairs.  Each key is unique while the values are not necessarily unique. **Two Algorithms Unsorted Key Array** Given an unsorted key array, we start our search at the first element and progress through the array element by element until we find a match. **SORTING** Sorting algorithms rearrange the elements of an array according to a pre-defined rule.  Typically, this rule is ascending or descending order.  The sorting criterion may be numeric or based upon a collating sequence such as ASCII or EBCDIC. The two simplest algorithms are S**ection Sort** A selection sort selects a reference element and steps through the rest of the elements looking for any one with a value that does not meet the test condition.  If found, the algorithm swaps that element with the reference element. The following program sorts the array in ascending order.  Starting with the first element in the array, it picks the first unsorted element as the reference element, swaps it with the largest element in the unsorted part of the array, and iterates until it reaches the last element in the array. **Bubble Sort** A bubble sort moves through the array element by element swapping elements if the next one does not satisfy the sort condition.  The algorithm repeats this process for each unsorted subset of the array starting with the first element.  The algorithm moves elements to their terminal positions just like bubbles rising through a liquid - hence the name bubble sort. The **stdio** library functions for processing input are: **scanf()** - input from standard input under format control **fscanf()** - input from file under format control **getchar()** - character by character input from standard input **fgetc()** - character by character input from file **gets\_s()** - line by line input from standard input (not universally implemented) **fgets()** - line by line input from file. **OUTPUT** The **stdio** library functions for processing output are: **printf()** - output to standard output under format control **fprintf()** - output to a file under format control **putchar()** - character by character output to standard output **fputc()** - character by character output to a file **puts()** - character string output to standard output **fputs()** - character string output to a file. **Member Access** Note that a structure type may NOT contain an object of its own type. Consider a section of a course that contains a list of enrolled students: struct Student { int no; float grade [4]; char name [31]; }; Struct Section { int no\_of\_students; struct Student s[40] ; }; Dot, arrow, and subscript syntax extends to the members of structure types that are themselves structure types. For example, let us define a **Section** object: **struct Section abc123a;** To set the number of students in **abc123a** to **23**, we write: **abc123a.no\_of\_students = 23;**To set the student number of the sixth **student** to **123-456-789**, we write: **abc123a.s[5].no = 123456789;** To set the third grade of the sixth student to an **67.8**, we write:**abc123a.s[5].grade[2] = 67.8f;** We say that **Section abc123a** has a **Student** with student number **123456789**, whose third grade has been set to **67.8**.