CS1010 Finals Cheatsheet

Unit 1 - Computational Thinking

4 types of Computational Thinking

Decomposition

- Breaking a problem into smaller, more manageable parts

Pattern Recognition

- Recognizing which parts are the same and the attributes that define them

Abstraction

- Filtering out information not needed and generalizing information that is needed

Algorithms

- Creating solutions using a series of ordered steps.

Unit 2 - Computing Environment

- High-level language programs (eg: C) cannot be executed directly by the computer
- Require a translation process called **compilation**
- A special program called **compiler** is used
- The original C program is called the **source code**
- The compiled program is the **executable code** or **machine code**
- In general, executable codes generated on a certain machine cannot be executed on another machine with a different architecture
- The source code needs to be compiled on the new machine

Unit 3 - Algorithmic Problem Solving

Euclid's Algorithm

1. Let A and B be integers with A > B >= 0
2. If B = 0, then GCD is A and algorithm ends.
3. Otherwise, find q and r such that A = q*B + r where 0 <= r < B
4. Replace A by B, and B by r, then go to step 2.

Example

```
// Uses Euclid's Algorithm to find GCD(A,B)
// Pre-Condition: A and B are non-negative integers != 0
int GCD(A, B){
  int r;
  while (B > 0){
    r = A % B;
    A = B;
    B = r;
  }
  return A;
}
```

Pseudocode guidelines

- Every step must be unambiguous, so that anybody is able to hand trace the pseudocode and follow the logic flow
- Use a combination of English (but keep it succinct) and commonly understood notations (such as for assignment in our previous example)
- Use indentation to show the control structures

Control structures (explained in pseudocode)

Sequence

```
To compute the average of three integers:

    enter values for num1, num2, num3
    total <- ( num1 + num2 + num3 )
    ave <- total  / 3
    print ave
```

Selection (also called branching)

Usage of if/else if/else statements

```
To arrange two integers in ascending order (sort):
enter values for num1, num2
// Assign smaller number into final1,
// and larger number into final2
if (num1 < num2)
  then
    final1 <- num1
    final2 <- num2
else
    final1 <- num2
    final2 <- num1
    // Transfer values in final1, final2 back to num1, num2
num1 <- final1
num2 <- final2
// Display sorted integers
print num1, num2
```

Repetition (also called loop)

```
To find the sum of positive integers up to n:

    enter value for n
    // Initialise a counter count to 1, and ans to 0
    count <- 1
    ans <- 0
    while (count ≤ n) do
         ans <- ans + count  // add count to ans
        count <- count + 1  // increase count by 1
    // Display answer
    print ans
```

Unit 4 - Overview of C Programming

Variables

- Data used in a program are stored in **variables**
- Every variable is identified by a name (identifier), has a **data type**, and contains a **value**

which could be modified
- A variable is *declared* with a data type

```
int count; // variable ‘count’ of type ‘int’
```

- Variables may be **initialized** during declaration

```
int count = 3; // count is initialized to 3
```

- Without initialization, the variable contains an unknown value (Cannot assume that it is zero!)

Types of Errors

- Syntax errors (and warnings)
  - Program violates syntax rules
  - Incomparable use of types of output
- Run-time Errors
  - Program terminates unexpectedly due to illegal operations when running (eg. division by zero)
- Logic Errors (or incorrect algorithm)
  - Program produces incorrect result
- Undetected Errors
  - Do not show up if we do not test the program thoroughly enough

Program Structure

A basic C program has 4 main parts:

- **Preprocessor Directives**
- **Input**
- **Compute**
- **Output**

Preprocessor directives

- Inclusion of header files eg. `#include <stdio.h>`
  - Enables program to use functions in header files

    - `#include <math.h>` to use mathematical functions
- Macro expansions eg. `#define GRAVITY -9.81`
- Conditional compilation

Input/Output

- Input/output statements:
    - `printf(format string, print list)`
    - `printf(format string)`
    - `printf(format string, input list)`

Example of using `printf` and `scanf` functions

```
int age;
double cap; // cumulative average point
printf("What is your age? ");
scanf("%d", &age);
printf("What is your CAP? ");
scanf("%lf", &cap);
printf("You are %d years old, and your CAP is %f\n", age, cap);
```

Format Specifiers

- `%d` and `%lf` are examples of **format specifiers**; they are placeholders for values to be displayed or read

Placeholder	Variable Type	Function Use
%c	`char`	printf / scanf
%d	`int`	printf/scanf
%f	`float` or `double`	printf
%f	`float`	scanf
%lf	`double`	scanf
%e	`float` or `double`	printf (for scientific notation)

Escape Sequence

- Escape sequences are used in printf() function for certain special effects or to display

certain characters properly

Escape Sequence	Meaning	Result
\n	New line	Subsequent output will appear on the next line
\t	Horizontal tab	Move to the next tab position on the current line
\"	Double quote	Display a double quote "
%%	Percent	Display a percent character %

Compute

- Computation is through **functions**
- A function body has two parts
  - Declarations statements: tell compiler what type of memory cells needed
  - Executable statements: describe the processing on the memory cells

Basic structure of a function

```
int main(void) {
    /* declaration statements */
    /* executable statements */
    return 0;
}
```

User-defined Identifiers (naming variables/functions)

- May consist of letters (a-z, A-Z), digits (0-9) and underscores (_), but ***MUST NOT*** begin with a digit
- case-sensitive i.e. count and Count are different identifiers
- Guideline: Usually should begin with lowercase letter
- Must not be **reserved words**
- Should avoid standard identifiers
- Valid identifiers : maxEntries, _X123, this_IS_a_long_name
- Invalid : 1Letter, double, return, joe's, ice cream, T*S

Reserved words (keywords)

- Have special meaning in C
- Eg: `int`, `void`, `double`, `return`

Standard identifiers

- Names of common functions, such as `printf`, `scanf`
- Avoid naming your variables/functions with the same name of built-in functions you intend to use

Executable statements

- I/O statements (eg: printf, scanf)
- Computational and assignment statements

Assignment statements

- Store a value or a computational result in a variable
- (Note: '=' means 'assign value on its right to the variable on its left'; it does NOT mean equality)
- Left side of '=' is called lvalue (Note: lvalue must be assignable, like a variable)

Side Effect

- An assignment statement does not just assigns, it also has the **side effect** of returning the value of its right-hand side expression
- Hence `a = 12`; has the side effect of returning the value of `12`, besides assigning `12` to `a`
- Usually we don't make use of its side effect, but sometimes we do, eg:

```
z = a = 12; // or z = (a = 12);
```

- The above makes use of the side effect of the assignment statement a = 12; (which returns 12) and assigns it to z
- Side effects have their use, but **avoid convoluted codes**

Arithmetic operations

- Binary Operators: `+`, `-`, `*`, `/`, `%`
  - Left Associative (from left to right)

    - 46 / 15 / 2 -> 3 / 2 -> 1
    - 19 % 7 % 3 -> 5 % 3 -> 2
- Unary operators: `+`, `-`
    - Right Associative
    - x = -23
    - p = +4 * 10
- Execution from left to right, respecting parentheses rule, and then precedence rule, and then associative rule
    - remainder/modulo `%` -> division `/` -> multiplication `*` -> subtraction `-` and addition `+`
- Truncated result if result can't be stored (the page after next)
    - `int n; n = 9 * 0.5;` results in 4 being stored in n.

Operator Type	Operator	Associativity
Primary expression operators	`( )` `expr++` `expr--`	Left to Right
Unary Operators	`*` `&` `+` `-` `++expr` `--expr` `(typecast)`	Right to Left
Binary Operators	`*` `/` `%` `+` `-`	Left to Right
Assignment Operators	`=` `+=` `-=` `*=` `/=` `%=`	Right to Left

Mixed-Type Arithmetic Operations

```
int m = 10/4; // m = 2;
float p = 10/4; // p = 2.0;
int n = 10/4.0; // n = 2;
float q = 10/4.0; // q = 2.5;
int r = -10/4.0; // r = -2;
```

Type Casting

Use of a **cast operator** to change the type of an expression (syntax: (type)expression)

```
int aa = 6; float ff = 15.8;
float pp = (float) aa / 4; // pp = 1.5;
int nn = (int) ff / aa; // nn = 2;
float qq = (float) (aa / 4) // qq = 1.0;
```

Unit 5 - Top Down Design and Functions

Components of function definition

- Header (or signature): consists of **return type**, function name, and a list of **parameters** (with their types) separated by commas
- Function names follow identifier rules (just like variable names)
  - May consist of letters, digit characters, or underscore, but cannot begin with a digit character
- Return type is `void` if function does not need to return any value
- Function body: code to perform the task; contains a return statement if return type is not void

Function prototypes

- It is a good practice to put **function prototypes** at the top of the program, before the main() function, to inform the compiler of the functions that your program may use and their return types and parameter types.
- **Function definitions** to follow *after* the main() function.
- Without function prototypes, you will get error/warning messages from the compiler.

Using function prototyes:

```
#include <stdio.h>

int f(int, int);

int main(void) {
  printf("%d\n", f(100, 7));
  return 0;
}

int f(int a, int b) {
  return a*b*b;
}
```

Pass-By-Value and Scope rules

- **Formal parameters** are local to the function they are declared in.
- Variables declared within the function are also **local** to the function.
- Local parameters and variables are only accessible in the function they are declared – scope rule.
- When a function is called, an activation record is created in the call stack, and memory is allocated for the local parameters and variables of the function.
- Once the function is done, the activation record is removed, and memory allocated for the local parameters and variables is released.
- Hence, local parameters and variables of a function exist in memory only during the execution of the function. They are called **automatic variables**.
- In contrast, static variables exist in the memory even after the function is executed.
- **Actual parameters** (also arguments) are values passed to function for computation

Global Variables

- **Global variables** are those that are declared *outside* all functions.
- Global variables can be accessed and modified by any function!
- Because of this, it is hard to trace when and where the global variables are modified.
- Hence, we will NOT allow the use of global variables

Unit 6 - Problem Solving with Selection and Repetition

Conditions

- A condition is an expression evaluated to true or false.
- It is composed of expressions combined with **relational operators**.
  - Examples: (a <= 10),="" (count="">> max), (value != -9)

Relational Operator	Interpretation
`<`	is less than
`<=`	is less than or equal to
`>`	is greater than
`>=`	is greater than or equal to

`==`	is equal to
`!=`	is not equal to

Truth Values

- Boolean values: `true` or `false`.
- There is no boolean type in ANSI C. Instead, we use integers:
  - 0 to represent false
  - Any other value to represent `true` (1 is used as the representative value for true in output)

Example:

```
int a = (2 > 3);
int b = (3 > 2);

printf("a = %d; b = %d\n", a, b);
//prints out a = 0; b = 1
```

Logical Operators

- **Complex condition**: combining two or more boolean expressions.
- **Logical operators** are needed: `&&` (and), `||` (or), `!` (not).

Evaluation of Boolean Expressions

The evaluation of a boolean expression is done according to the precedence and associativity of the operators.

Operator Type	Operator	Associativity
Primary expression operators	`( ) [ ] . -> expr++ expr--`	Left to Right
Unary Operators	`* & + - ++expr --expr (typecast) sizeof`	Right to Left
Binary Operators	`* / % + - > < <= >= == != && &#124;&#124;`	Left to Right

Ternary Operators	`?:`	Right to Left
Assignment Operators	`=` `+=` `-=` `*=` `/=` `%=`	Right to Left

Short-Circuit Evaluation

- expr1 || expr2: If expr1 is true, skip evaluating expr2 and return true immediately, as the result will always be true.
- expr1 && expr2: If expr1 is false, skip evaluating expr2 and return false immediately, as the result will always be false.

if and if-else statements

```
// if statement
int a, b, t;

if (a > b) {
    // Swap a with b
    t = a; a = b; b = t;
}
// After above, a is the smaller
```

```
// if-else statement
int a;

if (a % 2 == 0) {
    printf("%d is even\n", a);
}
else {
    printf("%d is odd\n", a);
}
```

switch statement

- An alternative to if-else-if is to use the switch statement.
- Restriction: Value must be of discrete type (eg: `int`, `char`)

```
switch ( <variable or expression> ) {
    case value1:
        Code to execute if <variable or expr> == value1
        break;
```

```
    case value2:
        Code to execute if <variable or expr> == value2
        break;
    ...

    default:
        Code to execute if <variable or expr> does not
        equal to the value of any of the cases above
        break;
}
```

`while` loop

```
// If condition is true, execute loop body; otherwise, terminate loop
while ( condition )
{
        // loop body
}
```

Each round of the loop is called an iteration.

`do` - `while` loop

```
do
{
    // loop body
}  while ( condition );
```

Executes the loop body at least once before checking for condition

`for` loop

```
for ( initialization; condition; update )
{
    // loop body
}
```

- Initialization: initialize the loop variable (variable MUST be declared beforehand for ASCI C)
- Condition: repeat loop while condition on loop variable is true

- Update: change value of loop variable after an iteration

using `break`

- `break` is used in switch statement
- Can also be used in a loop
- In a nested loop, `break` only breaks out of the inner-most loop that contains the `break` statement.

using `continue`

- Skips to the next iteration of the loop the `continue` is in.
- In a nested loop, `continue` only skips to the next iteration of the inner-most loop that contains the `continue` statement.

Unit 8 - Arrays

Array Declaration: Syntax

> T `arrname` [E]

- `arrname` is the name/identifier of array (like naming a variable)
- T is a data type (eg. int, double, char, ...)
- E is an integer constant expression with a positive value, which states the size of the array
- NO VARIABLE-LENGTH ARRAYS FOR ANSI C
  - Array size is determined at compile time

Array Declarations with Initializers

- Array Variables can be initialized at the time of declaration.

```
// a[0]=54, a[1]=9, a[2]=10
int a[3] = {54, 9, 10};

// size of b is 3 with b[0]=1, b[1]=2, b[2]=3
int b[] = {1, 2, 3};

// c[0]=17, c[1]=3, c[2]=10, c[3]=0, c[4]=0
int c[5] = {17, 3, 10};
```

```
// c[0] = 0, c[1] = 0, ... c[98] = 0, c[99] = 0...
int d[100] = { 0 };
```

Array Assignment (Copying array)

- Must be done using a loop

```
#define N 10
int source[N] = { 10, 20, 30, 40, 50 };
int dest[N];
int i;
for (i = 0; i < N; i++) {
    dest[i] = source[i];
}
```

Array Parameters in Functions

```
//Function prototypes can be written in the following ways:
int sumArray(int arr[], int size); // with parameter names
int sumArray(int [], int); // Without parameter names

//Function header can be written in the following ways:
int sumArray(int arr[], int size) { return 0; } // Without array size
int sumArray(int arr[8], int size) { return 0; } // with array size

//Passing array arguments
// ENSURE that SIZE is not larger than the actual array
//size otherwise core dump will happen
printf("sum is %d\n", sumArray(foo, 8)); // use the NAME of the array, no need []
```

- array name is the address of its first element. Hence foo means &foo[0].

Standard I/O functions for Arrays

- It might be advisable to write a function to read values into an array, and a function to print values in an array.
- Especially so for the latter, as you probably want to use it to check the values of your array elements at different stages of your program.

Input and Output

```c
void scanArray(float arr[], int size) {
    int i;
    // You may add a prompt for user here
    for (i=0; i<size; i++) {
        scanf("%f", &arr[i]);
    }
}

void printArray(float arr[], int size) {
    int i;
    // To print each value on one line
    for (i=0; i<size; i++)
        printf("%f\n", arr[i]);
}
```

Modifying Array arguments

- As one would with regular variables

```c
int main(void) {
    int foo[8] = {44, 9, 17, 1, -4, 22};
    doubleArray(foo, 4);
    . . .
}
// To double the values of array elements
void doubleArray(int arr[], int size) {
    int i;
    for (i=0; i<size; i++)
        arr[i] *= 2;
}
```

Unit 9 Multidimensional Arrays

Common array methods

```c
void printArray(int arr[], int size) {
    int i;

    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

```
int findMax(int arr[], int size) {
    int i, max;

    max = arr[0];
    for (i = 1; i < size; i++)
        if (arr[i] > max)
            max = arr[i];

    return max;
}

int sum(int arr[], int size) {
    int i, sum = 0;

    for (i = 0; i < size; i++)
        sum += arr[i];

    return sum;
}

int sumAlt(int arr[], int size) {
    int i, sum = 0;

    for (i = 0; i < size; i+=2)
        sum += arr[i];

    return sum;
}

int sumOdd(int arr[], int size) {
    int i, sum = 0;

    for (i = 0; i < size; i++)
        if (arr[i]%2 == 1)
            sum += arr[i];

    return sum;
}
/// NOT SO COMMON, BUT EXAMPLES OF ALGORITHMS
// SUM OF LAST 3 ELEMENTS. TAKE NOTE OF THE CONDITIONAL STATEMENT
int sumLast3(int arr[], int size) {
    int i, count = 0, sum = 0;

    for (i = size - 1; (i>=0) && (count<3); i--) {
        sum += arr[i];
        count++;
    }

    return sum;
}
```

Accessing 1D Array Elements in Function

A function header with array parameter,
`int sum(int a[ ], int size)`

- A value is not necessary (and is ignored by compiler if provided) as accessing a particular array element requires only the following information
  - The address of the first element of the array
  - The size of each element
- Therefore, both information are known by using `int a[]`, since that already refers to the address of the first element in the array `a`.
- Also, the size of each element is determined by the element type `int`

Multidimensional Arrays

- In general, an array can have any number of dimensions

Example of a 2-dimensional (2D) array:

```
// array with 3 rows, 5 columns
int a[3][5];
a[0][0] = 2;
a[2][4] = 9;
a[1][0] = a[2][4] + 7;
```

- Arrays are stored in row-major order
  - That is, elements in row 0 comes before row 1, etc.

Multidimensional Array Initializers

Examples:

```
// nesting one-dimensional initializers
int a[3][5] = { {4, 2, 1, 0, 0},
                {8, 3, 3, 1, 6},
                {0, 0 ,0, 0, 0} };

// the first dimension can be unspecified
int b[][5] = { {4, 2, 1, 0, 0},
               {8, 3, 3, 1, 6},
               {0, 0, 0, 0, 0} };

// initializer with implicit zero values
int d[3][5] = { {4, 2, 1},
                {8, 3, 3, 1, 6} };
```

- Uninitialized Elements are given zero value if not stated during initialization

Accessing 2D Array Elements in Function

A function header with 2D array parameter,
`function(int a[][5], ...)`

- To access an element in a 2D array, it must know the number of columns. It needs not know the number of rows.
- For multi-dimensional arrays, all but the first dimension must be specified in the array parameter.

Broad Example of Multidimensional Array Usage:

```
// Find students who are enrolled in all classes
void busiestStudents(int enrol[][MAX_STUDENTS],
            int numClasses, int numStudents) {
    int sum;
    int r, c;

    printf("Students who take all classes: ");
    for (c = 0; c < numStudents; c++) {
        sum = 0;
        for (r = 0; r < numClasses; r++) {
            sum += enrol[r][c];
        }
        if (sum == numClasses)
            printf("%d ", c);
    }
    printf("\n");
}
```

To note:

- Number of columns of a 2D array ***MUST*** be stated, number of rows not necessary.

- As elements are stored linearly in memory in row-major order, element `a[1][0]` would be the 4th element in the 3-column array, whereas it would be the 6th element in the 5-column array.

- In a function definition for multi-dimensional arrays, all but the first dimension must be specified in the array parameter.

Unit 10 - Random Numbers

rand()

- In sunfire, `rand()` generates an integer in the range [0, 32676]. (Note: [a, b] indicates a closed range, i.e. the range is inclusive of both a and b.)
- The same set of numbers are printed every time the program is run because the numbers are picked from a pre-determined sequence based on some **seed**.

srand()

- Generates a different set of random numbers each time `srand()` is called.
- Calling rand() after srand() will pick up the next number from the newly pre-determined sequence of pseudo-numbers
- Hence we only need to call `srand()` once before calling `rand()` function

Example

```
#include <stdio.h>
#include <stdlib.h> // required for rand() and srand()

int main(void) {
    int seed, i;

    printf("Enter seed: ");
    scanf("%d", &seed);
    srand(seed); // feed srand() with a new seed

    for (i = 1; i <= 10); i++)
        printf("%d\n", rand()%400 + 101); // gives a

    return 0;
}
```

Randomizing the Seed

- A seed is required in the `srand()` function
- To automate this step, we use `time(NULL)` function to be used as the seed for the `srand()` function

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    int i;

    srand(time(NULL));

    for (i = 1; i <= 10); i++)
        printf("%d\n", rand()%400 + 101);

    return 0;
}
```

Unit 20 - Searching and Sorting

Linear Search (or Sequential Search)

- Idea: Search the list from one end to the other end in linear progression.

Example of Linear Search Implementation (Returns the first element found)

```c
// To search for key in arr using linear search
// Return index if found; otherwise return -1
int linearSearch(int arr[], int size, int key) {
    int i;

    for (i=0; i<size; i++)
        if (key == arr[i])
            return i;
    return -1;
}
```

Binary Search

- Pre-condition: List must be sorted beforehand.
- How the data is organized (in this case, sorted) usually affects how we choose/design an algorithm to access them.

The Binary Search algorithm :

- Look for the key in the middle position of the list. Either of the following 2 cases happens:
    - If the key is smaller than the middle element, “discard” the right half of the list and repeat the process.
    - If the key is greater than the middle element, “discard” the left half of the list and repeat the process
- Terminating condition: when the key is found, or when all elements have been “discarded”.
- In binary search, each step eliminates the problem size (array size) by half!

Example:

```
int find(int data) {
   int lowerBound = 0;
   int upperBound = MAX -1;
   int midPoint = -1;
   int index = -1;

   while(lowerBound <= upperBound) {
      // compute the mid point
      // midPoint = (lowerBound + upperBound) / 2;
      midPoint = lowerBound + (upperBound - lowerBound) / 2;

      // data found
      if(intArray[midPoint] == data) {
         index = midPoint;
         break;
      } else {
         // if data is larger
         if(intArray[midPoint] < data) {
            // data is in upper half
            lowerBound = midPoint + 1;
         }
         // data is smaller
         else {
            // data is in lower half
            upperBound = midPoint -1;
         }
      }
   }
   return index;
}
```

Sorting

Sorting is any process of arranging items in some sequence and/or in different sets

Selection Sort

Algorithm

1. Find the smallest element in the list (find_min)
2. Swap this smallest element with the element in the first position. (Now, the smallest element is in the right place.)
3. Repeat steps 1 and 2 with the list having one fewer element (i.e. the smallest element just found and its place is “discarded” from further processing).

Example:

```
void selectionSort(int arr[], int size) {
    int i, start, min_index, temp;

    for (start = 0; start < size-1; start++) {
      // each iteration of the for loop is one pass
      // find the index of minimum element
      min_index = start;
      for (i = start+1; i < size; i++){
        if (arr[i] < arr[min_index]){
          min_index = i;
        }
      }
      // swap minimum element with element at start index
      temp = arr[start];
      arr[start] = arr[min_index];
      arr[min_index] = temp;
    }
}
```

- **Selection sort** is classified under exchange sort, where elements are exchanged in the process.

Bubble Sort

- The key idea Bubble sort is to make **pairwise comparisons** and exchange the positions of the pair if they are in the wrong order.

Example:

```c
// To sort arr in increasing order
void bubbleSort(int arr[], int size) {
    int i, limit, temp;

    for (limit = size-2; limit >= 0; limit--) {
        // limit is where the inner loop variable i should end

        for (i=0; i<=limit; i++) {
            if (arr[i] > arr[i+1]) { // swap arr[i] with arr[i+1]
                temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
            }
        }
    }
}
```

Unit 12: Testing and Debugging

Unit 13: Separate Compilation

Unit 7: Pointers

Variables and Addresses

- A **variable** has a unique **name** (identifier) in the function it is declared in, it belongs to some data **type**, and it contains a value of that type
- A variable occupies some space in the memory, and hence it has an address

Example of usage:

```c
int a = 123;
printf("a = %d\n", a);
printf("&a = %p\n", &a);
```

```
// prints out
// a = 123
// &a = ffbff7dc (or any address that it's at)
```

- `%p` is used as the format specifier for addresses
- Addresses are printed out in hexadecimal (base 16) format
- The address of a variable varies from run to run, as the system allocates any free memory to the variable

Pointers (or pointer variables)

A variable that contains the address of another variable is called a pointer variable, or simply, a pointer.

Example/Usage:

```
// Declaraiton/Initialization of Variables
int numberA = 10;
float floatingB = 20;
char[8] word = "hunter2"; // a string
// Declaration of Pointers
int* a_pointer; // OR int *a_pointer
// a_pointer refers to the address, while *a_pointer refers to its value
float* b_pointer;
char* character_pointer;

a_pointer = &numberA; // valid, &numberA refers to the address
b_pointer = &numberA; // THIS IS INVALID, different type

b_pointer = &floatingB // this is valid

character_pointer = word // Valid! Since word is a char array.
character_pointer = &word[0] // analogous to the statement above !

printf("%c", *character_pointer); // remember to dereference with * !
printf("%d", *numberA);
printf("%p", character_pointer); // prints out an address
```

Unit 14: Functions with Pointer Parameters

Main Reason: To get around not being able to return more than 1 value in a function (without using structs)
by modifying values of variables **outside** of the function

Example:

```c
#include <stdio.h>

void swap(int *, int *); // USE * to indicate the use of pointers

int main(void) {
    int var1, var2;

    printf("Enter two integers: ");
    scanf("%d %d", &var1, &var2);

    swap(&var1, &var2);
    // swap() call uses the addresses since function requires pointers

    printf("var1 = %d; var2 = %d\n", var1, var2);
    return 0;
}

void swap(int *ptr1, int *ptr2) {
    int temp;
    temp = *ptr1; *ptr1 = *ptr2; *ptr2 = temp;
}
```

Unit 16: Characters and Strings

Introduction

- In C, single characters are represented using the data type char
- Character constants are written as symbols enclosed in single quotes
  - Examples: 'g', '8', '*', ' ', '\n', '\0'
- Recall: Practice S02P03 - NRIC Check Code
- Characters are stored in one byte, and are encoded as numbers using the ASCII scheme

```c
// Unit16_CharacterDemo1.c
#include <stdio.h>

int main(void) {
```

```
    char grade = 'A', newgrade, ch;
    int value;

    printf("grade = %c\n", grade); // prints 'A'
    newgrade = grade + 2;
    printf("newgrade = %c\n", newgrade); // prints 'C'
    printf("newgrade = %d\n", newgrade); // prints '67'

    value = 65;
    printf("value = %d\n", value); // prints '65'
    printf("value = %c\n", value); // prints 'A'

    if ('A' < 'c') // 'A' is 65, 'c' is 99
        printf("'A' is less than 'c'\n");
    else
        printf("'A' is not less than 'c'\n");

    for (ch = 'p'; ch <= 't'; ch++) // using character variable as a loop var
        printf("ch = %c\n", ch);

    return 0;
}
```

Useful char functions

- `getchar(char)` --> Reads a character from stdin
- `putchar(char)` --> Prints a character to stdout

*** functions ***

- `isalpha(char)` - returns 1 if character is alphabet
- `isalnum(char)` - returns 1 if character is alphanumeric
- `isupper(char)` - returns 1 if character is uppercase
- `islower(char)` - returns 1 if character is lowercase
- `isdigit(char)` - returns 1 if character is a digit
- `isspace(char)` - returns 1 if character is a whitespace character
- `ispunct(char)` - returns 1 if character is a punctuation.

Strings

A string is an array of characters, **terminated by a null character '\0'** (which has ASCII value of zero)

Usage:

```c
// The Basics
/// Declaration
char[4] str;
/// Assigning individual characters to an element in char[]
str[0] = 'e';
str[1] = 'g';
str[2] = 'g';
str[3] = '\0';
/// Initializing a string (two ways)
char[6] fruit = "apple"; // the '/0' is added automatically
char[7] anotherFruit = {'b','a','n','a','n','a','\0'}; // like a normal array
// Reading from stdin
int size = 4;
fgets(str, size, stdin); //reads size - 1 char or until newline.
// also reads in the newline character
scanf("%s", str); // reads until white space
// Output to stdout
puts(str); // terminates with newline
printf("%s", str); // until '\0' character
```

functions

- `strcmp(s1, s2)` - Compare the ASCII values of both strings.
  Returns negative int if s1 < s2, 0 if s1 == s2, positive if s1 > s2
- `strncmp(s1, s2, n)` - Compare first n characters of s1 and s2
- `strcpy(dest, src)` - copies string pointed to by `src` to `dest`
- `strncpy(dest, src, n)` - copies first n characters from src to dest
- `strstr(s1, s2)` - Returns a pointer to the first instance of s2 in s1
  Returns a NULL pointer if s2 is not found in s1

Array of strings

Usage/Example:

```c
// Declaration
char fruits[5][25]; // array of 5 strings of max size 25
//Initialization
char fruits[][6] = {"apple", "mango", "pear"};
// or fruits[3][6]
//Output
printf("%s %s\n", fruits[0], fruits[1])
printf("%c", fruits[2][1]) // prints the char 'e'
```

- a string is physically an array of characters, the name of a string is also a pointer (that points to the first character of the string)

```
char str[] = "apple";

printf("1st character: %c\n", str[0]); // prints 'a'
printf("1st character: %c\n", *str); // prints 'a'

printf("5th character: %c\n", str[4]); // prints 'e'
printf("5th character: %c\n", *(str+4)); // prints 'e'
```

interpreting while(*p++)

`while (*p++)`

1. Check whether *p is 0 (that is, whether *p is the null character '\0')
2. Then, increment p by 1 (so that p points to the next character).
   Not increment *p by 1!

- (*p++) is not the same as (*p)++
  (*p)++ is to increment *p (the character that p points to) by 1. (Hence, if p is pointing to character 'a', that character becomes 'b'.)

Unit 17: Recursion

To write a recursive function:

- Identify the base case(s) of the relation
- Identify the recurrence relation
- Always check for base case(s) first
- Do not write redundant base cases

How recursive functions work

- When a function is called, an activation record (or frame) is created by the system.
- Each activation record stores the local parameters and variables of the function and its return address.
- Such records reside in the memory called stack.
  - Stack is also known as LIFO (last-in-first-out) structure

- A recursive function can potentially create many activation records
  - Winding: each recursive call creates a separate record
  - Unwinding: each return to the caller erases its associated record

Examples

Fibonacci

```
// Pre-cond: n >= 0
int fib(int n) {
    if (n < 2)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

Sum of Squares (x to y)

```
int sumSq1(int x, int y) {
    if (x == y) return x * x;
    else return x * x + sumSq1(x+1, y);
}
```

Unit 15 & 18: Structures

The Structure Type

- A type needs to be defined before we can declare variable of that type
- No memory is allocated to a type

```
// defining new struct/type
typedef struct {
    int length, width, height;
    float density;
    char[30] name;
} box_t;

// declaring a new variable of the type
box_t box1, box2;
```

```c
// Initializing variable
box_t box3 = {20, 40, 10, 0.53, "Bobby"};

// Accessing members of a structure variables
box3.length = 40;
strcpy(box3.name, "Timmy");
scanf("%d %f %s", &box3.width, &box3.density, box3.name);

// Assignment of Structures
box1 = box3;
```

Array of Structures (example/usage)

```c
typedef struct {
    int id;
    char name[MAX_LENGTH+1];
} customer_t;

typedef struct {
    int cusID;
    char category[MAX_LENGTH+1];
    int spending;
} record_t;

void readInputs(customer_t customerArr[], record_t recordsArr[],
    int* customersSize, int* recordsSize){

    int i;

    printf("Enter number of customers: \n");
    scanf("%d", customersSize);

    printf("Enter customers: \n");
    for (i = 0; i < *customersSize; i++)
        scanf("%d %s", &customerArr[i].id, customerArr[i].name);

    printf("Enter number of records: \n");
    scanf("%d", recordsSize);

    printf("Enter records: \n");
    for (i = 0; i < *recordsSize; i++)
        scanf("%d %s %d", &recordsArr[i].cusID,
                recordsArr[i].category, &recordsArr[i].spending);
}

//SEARCHING
int getIDByName(customer_t customerArr[], char name[], int size){
    int i;
    for (i = 0; i < size; i++){
        if (strcmp(customerArr[i].name, name) == 0){
            return customerArr[i].id;
        }
```

```
    }
    return -1;
}

// SEARCHING
int findRecords(record_t recordsArr[], int id, int size, record_t result[]){

    int i, runningCount = 0;

    for (i = 0; i < size; i++){
        if (recordsArr[i].cusID == id){
            result[runningCount] = recordsArr[i];
            runningCount++;
        }
    }
    return runningCount;
}

// (SELECTION) sorting
void sortRecords(record_t arr[], int size){
    int i, j;
    record_t tmp;
    int maxPrice, maxIndex;

    for (i = 0; i < size - 1; i++){
        maxPrice = arr[i].spending;
        maxIndex = i;
        for (j = i + 1; j < size; j++){
            if (arr[j].spending > maxPrice){
                maxPrice = arr[j].spending;
                maxIndex = j;
            }
        }
        if (maxIndex != i){
            tmp = arr[i];
            arr[i] = arr[maxIndex];
            arr[maxIndex] = tmp;
        }
    }
}
```

The arrow operator (`->`)

- dereferences the struct (given the address) to access the instance's variables

```
void change_name_and_age(player_t *player_ptr) {
    strcpy((*player_ptr).name, "Alexandra");
    (*player_ptr).age = 25;
}

// OR
```

```c
void change_name_and_age(player_t *player_ptr) {
    strcpy(player_ptr->name, "Alexandra");
    player_ptr->age = 25;
}
```

Unit 19 : File processing

- In C, input/output is done based on the concept of a stream
- A stream can be a file or a consumer/producer of data
- A stream is accessed using file pointer variable of type `FILE *`
- The I/O functions/macros are defined in `stdio.h`

Main Functions:

`fopen()`
`fclose()`
`fscanf()`
`fprintf()`

Two useful constants:

- `NULL` : null pointer constant
- `EOF` : used to represent end of file or error condition

Example :

```c
// Read number of prices and prices into array arr.
// Return number of prices read.
int scanPrices(float arr[]) {
    FILE *infile;
    int size, i;

    infile = fopen("prices.in", "r"); // open file for reading
    fscanf(infile, "%d", &size);

    for (i=0; i<size; i++)
        fscanf(infile, "%f", &arr[i]);
```

```
    fclose(infile);
    return size;
}

// Print the total price
void printResult(float total_price) {
    FILE *outfile;
    outfile = fopen("prices.out", "w"); // open file for writing
    fprintf(outfile, "Total price = $%.2f\n", total_price);
    fclose(outfile);
}
```

Opening File and File Modes

Prototype:
`FILE *fopen(const char *filename, const char *mode)`

'r' for read, 'w' for write

- Returns NULL if error; otherwise, returns a pointer of FILE type
- Possible errors: non-existent file (for input), or no permission to open the file

Example :

```
int scanPrices(float arr[]) {
    FILE *infile;
    int size, i;
    if ((infile = fopen("prices.in", "r")) == NULL) {
        printf("Cannot open file \"prices.in\"\n");
        exit(1);
    }
```

- Function exit(n) terminates the program immediately, passing the value n to the operating system. Putting different values for n at different exit() statements allows us to trace where the program terminates. n is typically a positive integer (as 0 means good run)
- To use the exit() function, need to include `<stdlib.h>`.

Closing File

Prototype:
`int *fclose(FILE *fp)`

Allows a file that is no longer used to be closed

- Returns `EOF` if error is detected; otherwise, returns 0
- It is good practice to close a file after use

Formatted I/O

Uses format strings to control conversion between character and numeric data

- `fprintf` : converts numeric data to character form and writes to an output stream
- `fscanf` : reads and converts character data from an input stream to numeric form

Both `fprintf` and `fscanf` functions can have variable numbers of arguments

```
float weight, height;
FILE  *fp1, *fp2;
. . .
fscanf(fp1, "%f %f", &weight, &height);
fprintf(fp2, "Wt: %f, Ht: %f\n", weight, height);
```

- `fprintf` returns a negative value if an error occurs; otherwise, returns the number of characters written

- `fscanf` returns EOF if an input failure occurs before any data items can be read; otherwise, returns the number of data items that were read and stored

Detecting EOF & Errors

- Each stream is associated with two indicators: error indicator & end-of-file (EOF) indicator

    - Both indicators are cleared when the stream is opened
    - Encountering end-of-file sets end-of-file indicator
    - Encountering read/write error sets error indicator
    - An indicator once set remains set until it is explicitly cleared by calling clearerr or some other library function

- `feof()` returns a non-zero value if the end-of-file indicator is set; otherwise returns 0

- `ferror()` returns a non-zero value if the error indicator is set; otherwise returns 0

- Need to include

Line I/O: Output

- `fputs()` and `puts()` return EOF if a write error occurs; otherwise, they return a non-negative number (input is a `FILE *`)

```
FILE *fp;

// writes to stdout with newline character appended
puts("Hello world!");

fp = fopen( ... );
// writes to fp without newline character appended
fputs("Hello world!", fp);
```

Line I/O: Input

- `fgets()` and `gets()` store a null character at the end of the string
- `fgets()` and `gets()` return a null pointer if a read error occurs or end-of-file is encountered before storing any character; otherwise, return first argument

```
char s[100];
FILE *fp;

gets(s);  // reads a line from stdin

fp = fopen( ... );
fgets(s, 100, fp); // reads a line from fp
```

Appendix

printf formatting options

```
int main()
{
    std::printf("Strings:\n");

    const char* s = "Hello";
    std::printf("\t[%10s]\n\t[%-10s]\n\t[%*s]\n\t[%-10.*s]\n\t[%-*.*s]\n",
        s, s, 10, s, 4, s, 10, 4, s);

    std::printf("Characters:\t%c %%\n", 65);
```

```
    std::printf("Integers\n");
    std::printf("Decimal:\t%i %d %.6i %i %.0i %+i %i\n", 1, 2, 3, 0, 0, 4, -4);
    std::printf("Hexadecimal:\t%x %x %X %#x\n", 5, 10, 10, 6);
    std::printf("Octal:\t%o %#o %#o\n", 10, 10, 4);

    std::printf("Floating point\n");
    std::printf("Rounding:\t%f %.0f %.32f\n", 1.5, 1.5, 1.3);
    std::printf("Padding:\t%05.2f %.2f %5.2f\n", 1.5, 1.5, 1.5);
    std::printf("Scientific:\t%E %e\n", 1.5, 1.5);
    std::printf("Hexadecimal:\t%a %A\n", 1.5, 1.5);
    std::printf("Special values:\t0/0=%g 1/0=%g\n", 0.0/0.0, 1.0/0.0);

    std::printf("Variable width control:\n");
    std::printf("right-justified variable width: '%*c'\n", 5, 'x');
    int r = std::printf("left-justified variable width : '%*c'\n", -5, 'x');
    printf("(the last printf printed %d characters)\n", r);

    // fixed-width types
    uint32_t val = std::numeric_limits<std::uint32_t>::max();
    printf("Largest 32-bit value is %" PRIu32 " or %#" PRIx32 "\n", val, val);
}
```

OUTPUT:

```
Strings:
        [     Hello]
        [Hello     ]
        [     Hello]
        [Hell      ]
        [Hell      ]
Characters:     A %
Integers
Decimal:        1 2 000003 0  +4 -4
Hexadecimal:    5 a A 0x6
Octal:  12 012 04
Floating point
Rounding:       1.500000 2 1.30000000000000004440892098500626
Padding:        01.50 1.50  1.50
Scientific:     1.500000E+00 1.500000e+00
Hexadecimal:    0x1.8p+0 0X1.8P+0
Special values: 0/0=nan 1/0=inf
Variable width control:
right-justified variable width: '    x'
left-justified variable width : 'x    '
(the last printf printed 40 characters)
Largest 32-bit value is 4294967295 or 0xffffffff
```