

COMP9024: Data Structures and Algorithms

Week 1: Introduction to C

Contents

- Basic Structure of a C Program
- Assignments
- Conditionals
- Loops
- Functions
- Basic Data Types
- Arrays
- Strings
- Functions
- Structures
- Basic I/O

Brief History of C

- C and UNIX operating system share a complex history
- C was originally designed for and implemented on UNIX on a PDP-11 computer
- Dennis Ritchie was the author of C (around 1971)
- In 1973, UNIX was rewritten in C by Ken Thompson and Dennis Ritchie
- B (author: Ken Thompson, 1970) was the predecessor to C, but there was no A

Basic Structure of a C Program

```
// include files
// global variable definitions
// function definitions
function_type function_name(arguments)
{
    // local variables
    // body of function
    ...
    return x; //return a value
}
...
```

```
// main function
int main(arguments)
{ // local variables
    // body of main function
    ...
    return 0;
}
```

Example 1: Insertion Sort in C

```
#include <stdio.h>
#define SIZE 6

void insertionSort(int array[], int n) {
    int i;
    for (i = 1; i < n; i++) {
        int element = array[i];          // for this element ...
        int j = i-1;
        while (j >= 0 && array[j] > element) {
            // work down the ordered list
            array[j+1] = array[j];      // move elements up
            j--;
        }
        array[j+1] = element; // and insert in correct position
    }
}
```

```
int main(void) {
    int numbers[SIZE] = { 3, 6, 5, 2, 4, 1 };
    int i;

    insertionSort(numbers, SIZE);
    for (i = 0; i < SIZE; i++)
        printf("%d\n", numbers[i]);

    return 0;
}
```

Example 2: Computing Greatest Common Divisor

```
#include <stdio.h>

int f(int m, int n) {    /* compute the greatest common divisor of m and n */
    while (m != n) {
        if (m > n)
            m = m-n;
        else
            n = n-m;
    }
    return m;
}

int main(void) {
    printf("%d\n", f(30,18));
    return 0;
}
```

Compiling with gcc

- C source code: prog.c
 ↓
 prog.exe (executable program)
- To compile a program prog.c, type the following command:
 prompt\$ gcc prog.c -o prog.exe
- To run the program, type:
 prompt\$ prog.exe

Basic Elements

- Operators
- Assignment statements
- Conditionals
- Loops
- Functions

Operators

- Arithmetic operators (+, -, *, /, %)
- Increment and decrement operators (++ , --)
- Assignment operators (=, +=, -=, *=, /=, ...)
- Relational operators (==, <, >, <=, >=, !=)
- Logical operators (&&, ||, !)
- Bitwise operators (&, |, ^, ~, <<, >>)
- Comma operator (,)
- sizeof operator
- ternary operator ?:
- Reference operator &
- Dereference operator *
- Member selection operator (->, .)

Precedence of Operators (1/2)

- If more than one operators are involved in an expression, C language has a predefined rule of priority for the operators. This rule of priority of operators is called operator precedence.
- In C, precedence of arithmetic operators (*, %, /, +, -) is higher than relational operators (==, !=, >, <, >=, <=), and precedence of relational operator is higher than logical operators (&&, || and !).
- Details can be found at

https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm

Precedence of Operators (2/2)

Examples:

$(1 > 2 + 3 \ \&\& \ 4)$ is equivalent to $1 > (2 + 3) \ \&\& \ 4$

$1 + 2 * 3 / 2$ is equivalent to $1 + (2 * 3) / 2$

Assignment Statements (1/3)

Syntax:

variable=expression;

Evaluate *expression* and assign its value to *variable*.

Examples:

```
int x, y, z, a, b;
```

```
z=x*(x+y)-a;
```

```
x=z*a-b;
```

Assignment Statements (2/3)

If *op* is one of { +, -, *, /, %, <<, >>, &, ^, |}, the assignment statement

variable op= expression

is equivalent to

variable = variable op expression

Examples:

int x, y, z, a, b;

z=x-a* is equivalent to *z=z*(x-a)*

*x+=z*b* is equivalent to *x=x+z*b*

Assignment Statements (3/3)

- The operators ++ and -- can be used to increment a variable (add 1) or decrement a variable (subtract 1)
- Post-increment and post decrement // suppose k=6 initially
k++; // increment k by 1; afterwards, k=7
n = k--; // first assign k to n, then decrement k by 1
// afterwards, k=6 but n=7
- Pre-increment and pre-decrement // again, suppose k=6 initially
++k; // increment k by 1; afterwards, k=7
n = --k; // first decrement k by 1, then assign k to n

Conditionals

```
if (expression) {  
    some statements;  
}
```

```
if (expression) {  
    some statements1;  
} else {  
    some statements2;  
}
```

- Some statements executed if and only if the evaluation of expression is non-zero
- Some statements1 executed when the evaluation of expression is non-zero
- Some statements2 executed when the evaluation of expression is zero
- Statements can be single instructions or blocks enclosed in { }

Printing Variable Values with printf()

- Formatted output written to standard output (e.g. screen)

```
printf(format-string, expr1, expr2, ...);
```

- format-string can use the following placeholders:

%d	decimal	%f	floating point number
%c	character	%s	string
\n	new line	\"	quotation mark

Examples:

```
num = 3;
```

```
printf("The cube of %d is %d.\n", num, num*num*num);
```

// Output: The cube of 3 is 27.

```
char id = 'z';
```

```
int num = 1234567;
```

```
printf("Your \"login ID\" will be in the form of %c%d.\n", id, num);
```

// Output: Your "login ID" will be in the form of z1234567.

- Can also use width and precision:

```
printf("%8.3f\n", 3.14159);
```

// Output is 3.142

Loops (1/3)

- while loop:

```
while (expression) {  
    some statements;  
}
```

Example:

```
int i = 1, sum=0;  
while (i <= 100)  
{  
    sum+=i;  
    ++i;  
}
```

Loops (2/3)

- do ... while loop

```
do {  
    statements;  
} while (expression);
```

- The do ... while loop ensures the statements will be executed at least once

Example:

```
int i = 1, sum=0;
```

```
do {  
    sum+=i;  
    ++i;  
}
```

```
while (i <= 100)
```

Loops (3/3)

- for loop:

```
for ( initialise; guard; update)
{
    statements
}
```

Example:

```
sum=0;
for(count = 1; count <= 100; ++count)
{
    sum += count;
}
```

Functions (1/4)

- Functions have the form

```
return-type function-name(parameters) {  
    declarations  
    statements  
    return;  
}
```

- if `return_type` is void, the function does not return a value
- if `parameters` are void, the function has no arguments

Functions (2/4)

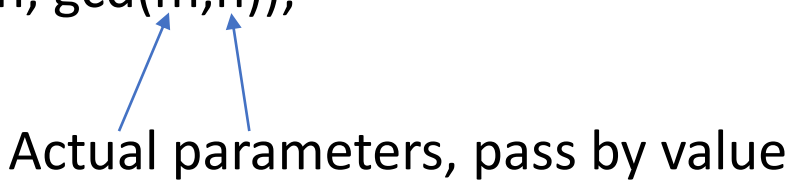
Parameter passing in C:

1. Pass by value: pass the value of an actual parameter to the formal parameter
 - Any change to the formal parameter does not have any effect on the actual parameter
 - Typically used for single-valued variables
2. Pass by reference: pass the address of an actual parameter to the formal parameter
 - Any change to the formal parameter will also be made to the actual parameter
 - Typically used for multi-valued variables such as arrays and structures

Functions (3/4)

```
#include <stdio.h>
int gcd(int m, int n);
int main()
{
    int m, n;
    printf("Enter two positive integers: ");
    scanf("%d %d", &m, &n);

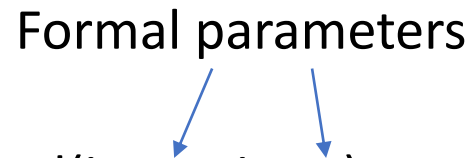
    printf("The greatest common divisor of %d
and %d is %d.", m, n, gcd(m,n));
    return 0;
}
```



Actual parameters, pass by value

Formal parameters

```
int gcd(int m, int n)
// compute the GCD of two positive
integers
{
    if (n != 0)
        return gcd(n, m%n);
    else
        return m;
}
```



Functions (4/4)

```
#include <stdio.h>
```

```
int main(void)
{
    int x = 100, y = 50;

    // pass by reference
    swap(&x, &y);

    printf("x is %d and y is %d\n", x, y);
    return 0;
}
```

```
void swap(int* i, int* j)
{
    int temp = *i;
    *i = *j;
    *j = temp;
}
```

C Style Guide

- UNSW Computing provides a style guide for C programs:
- C Coding Style Guide:
 - Brief guide: <http://www.cse.unsw.edu.au/~huiw/styleguide.pdf>
 - Detailed guide: https://cgi.cse.unsw.edu.au/~cs1511/20T1/resources/style_guide.html
- Not mandatory for COMP9024, but very useful guideline
- Use proper layout, including indentation
- Keep functions short and break into sub-functions as required
- Use meaningful names (for variables, functions etc)

Basic Data Types (1/2)

- In C each variable must have a type
- C has the following generic data types:

```
char    character    'A', 'e', '#', ... // Characters are encoded using ASCII code
```

```
int           integer    2, 17, -5, ...    // 32 bits
```

float	32 bit single precision floating-point number	3.14159, ...
-------	---	--------------

double	64 bit double precision floating-point	3.14159265358979, ...
--------	--	-----------------------

- There are other types, which are variations on these
- Variable declaration must specify a data type and a name; they can be initialised when they are declared:

```
float x;
```

```
char ch = 'A';
```

```
int j = 10;
```

Symbolic Constants

- We can define a symbolic constant at the top of the file
`#define SPEED_OF_LIGHT 299792458.0`
- Symbolic constants used to avoid burying "magic numbers" in the code
- Symbolic constants make the code easier to understand and maintain
`#define name replacement_text`
- The compiler's pre-processor will replace all occurrences of **name** with **replacement_text**. It will not make the replacement if name is inside quotes ("...") or part of another name
- Example: The constants TRUE and FALSE are often used when a condition with logical value is wanted. They can be defined by:
`#define TRUE 1`
`#define FALSE 0`

Aggregate Data Types

- Families of aggregate data types:
 - homogenous ... all elements have same base type
 - ❑ arrays (e.g. `char s[50]`, `int v[100]`)
 - heterogeneous ... elements may combine different base types
 - ❑ structures

Arrays

- An array is a collection of same-type variables arranged as a linear sequence accessed using an integer subscript
- For an array of size N, valid subscripts are 0..N-1
- Examples:

`int a[20];` // array of 20 integer values/variables

`char b[10];` // array of 10 character values/variables

Strings

- "String" is a special word for an array of characters
- end-of-string is denoted by '\0' (of type char and always implemented as 0)
- Example:

If a character array `s[11]` contains the string "hello", this is how it would look in memory:

0	1	2	3	4	5	6	7	8	9	10

h	e	l	l	o	\0					

- Characters are encoded using ASCII code

Array Initialisation

- Arrays can be initialised by code, or you can specify an initial set of values in declaration.
- Examples:

```
char s[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

```
char t[6] = "hello";
```

```
int fib[20] = {1, 1};
```

```
int vec[] = {5, 4, 3, 2, 1};
```

In the third case, `fib[0] = fib[1] = 1` while the initial values `fib[2] .. fib[19]` are undefined.

In the last case, C infers the array length (as if we declared `vec[5]`).

Arrays and Functions

- When an array is passed as a parameter to a function, the address of the start of the array is actually passed
- Example:

```
int total, vec[20];
```

```
...
```

```
total = sum(vec);
```

Within the function, the types of elements in the array are known, and the size of the array is unknown

Multi-dimensional Arrays

- Examples:

`float q[2][2];`

$$\begin{bmatrix} 0.5 & 2.7 \\ 3.1 & 0.1 \end{bmatrix}$$

`int r[3][4];`

$$\begin{bmatrix} 5 & 10 & -2 & 4 \\ 0 & 2 & 4 & 8 \\ 21 & 2 & 1 & 42 \end{bmatrix}$$

Note: `q[0][1]==2.7`, `r[1][3]==8`, `q[1]=={3.1,0.1}`.

- Multi-dimensional arrays can also be initialised:

```
float q[][] = {{0.5, 2.7}, {3.1, 0.1}};
```


Giving Data Types New Names

- C allows us to give a new name to a data type via typedef:

```
typedef ExistingDataType NewTypeName;
```

- Examples:

```
typedef char BYTE;  
BYTE x1, x2;
```

- By convention, uppercase letters are used to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase:

```
typedef char byte;  
byte x1, x2;
```

Structures

- A structure is a collection of variables, perhaps of different types, grouped together under a single name
- It helps to organise complicated data into manageable entities
- It exposes the connection between data within an entity
- It is defined using the `struct` keyword
- Example:

```
struct date {  
    int day;  
    int month;  
    int year;  
}; // don't forget the semicolon!
```

typedef and struct

- We can also define a structured data type TicketT for speeding ticket objects:

```
typedef struct {  
    int day, month, year;  
} DateT;
```

```
typedef struct {  
    int hour, minute;  
} TimeT;
```

```
typedef struct {  
    char plate[7]; // e.g. "DSA42X"  
    double speed;  
    DateT d;  
    TimeT t;  
} TicketT;
```

Basic I/O in C

- C programming treats all the I/O devices as files. So I/O devices such as the display are handled in the same way as files.
- The following three files are automatically opened when a program executes to provide access to the keyboard and screen:
 - Standard input `stdin` for keyboard
 - Standard output `stdout` for screen
 - Standard error `stderr` for screen

getchar() and putchar()

`int getchar(void)`

- Reads the next available character from keyboard and returns it as an integer

`int putchar(int c)`

- Puts the passed character on the screen and returns the same character

```
#include <stdio.h>
```

```
int main( ) {
```

```
    int c;
```

```
    printf( "Enter a value :");
```

```
    c = getchar( );
```

```
    printf( "\n You just entered: ");
```

```
    putchar( c );
```

```
    return 0;
```

```
}
```

gets() and puts()

`char *gets(char *s)`

- Reads a line from stdin into the buffer pointed to by s until either a terminating newline

`int puts(const char *s)`

- Writes the string s and a trailing newline to stdout

```
#include <stdio.h>
int main( ) {
    char str[30];
    printf( "Please enter a value :");
    gets( str );
    printf( "\nYou just entered: ");
    puts( str );
    return 0;
}
```

scanf() and printf()

`int scanf(const char *format, ...)`

- Reads the input from the standard input stream `stdin` and scans it according to the format provided

`int printf(const char *format, ...)`

- Writes the output to the standard output stream `stdout` and produces the output according to the format provided

```
#include <stdio.h>
```

```
int main( ) {
```

```
    char str[30];
```

```
    int i;
```

```
    printf( "Please enter a value :");
```

```
    scanf("%s %d", str, &i);
```

```
    printf( "\nYou just entered: %s %d ", str, i);
```

```
    return 0;
```

```
}
```

fopen()

fopen() creates a new file or to open an existing file.

```
FILE *fopen( const char * filename, const char * mode );
```

where the access mode can have one of the following values:

r: Opens an existing text file for reading.

w: Opens a text file for writing. If it does not exist, then a new file is created.

a: Opens a text file for writing in appending mode. If it does not exist, then a new file is created.

r+: Opens a text file for both reading and writing.

w+: Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.

a+: Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

fclose()

```
int fclose( FILE *fp );
```

- `fclose()` returns zero on success, or EOF if there is an error in closing the file. It flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file `stdio.h`.

fgetc() and fgets()

```
int fgetc(FILE *pointer);
```

It returns the character at the position indicated by file pointer. After reading the character, the file pointer is advanced to next character. If the pointer is at the end of file or if an error occurs, EOF is returned.

```
char *fgets( char *buf, int n, FILE *fp );
```

It reads up to n-1 characters from the input stream referenced by fp and copies the read string into the buffer buf, appending a null character to terminate the string.

fputc() and fputs()

```
int fputc( int c, FILE *fp );
```

It writes the given character at the position denoted by the file pointer and then advances the file pointer. This function returns the character written in case of successful write operation, or EOF is returned in case of error.

```
int fputs( const char *s, FILE *fp );
```

It writes the string s to the output stream referenced by fp. It returns a non-negative value on success; otherwise EOF is returned in case of any error.

Examples

```
#include <stdio.h>
```

```
main() {
```

```
    FILE *fp;
```

```
    fp = fopen("test.txt", "w+");
```

```
    fputs("This is testing for fputs\n", fp);
```

```
    fclose(fp);
```

```
}
```

```
#include <stdio.h>
```

```
main() {
```

```
    FILE *fp;
```

```
    char buff[50];
```

```
    fp = fopen("test.txt", "r");
```

```
    fgets(buff, 50, (FILE*)fp);
```

```
    printf("%s\n", buff );
```

```
    fclose(fp);
```

```
}
```

fread()

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

It reads data from the given stream into the array pointed to, by ptr.

Parameters

- ptr – The pointer to a block of memory with a minimum size of size*nmemb bytes.
- size – The size in bytes of each element to be read.
- nmemb – The number of elements, each one with a size of size bytes.
- stream – The pointer to a FILE object that specifies an input stream.

Return Value

- The total number of elements successfully read is returned as a size_t object, which is an integral data type. If this number differs from the nmemb parameter, then either an error has occurred or the End Of File was reached.

fwrite()

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

It writes data from the array pointed to, by ptr to the given stream.

Parameters

- ptr – This is the pointer to the array of elements to be written.
- size – This is the size in bytes of each element to be written.
- nmemb – This is the number of elements, each one with a size of size bytes.
- stream – This is the pointer to a FILE object that specifies an output stream.

Return Value

- On success, it returns the count of the number of elements successfully written to the file. On error, it returns a number less than nmemb.

Example 1

```
#include<stdio.h>
struct Student
{
    int roll;
    char name[30];
    float marks;
};
int main()
{
    FILE *fp;
    char ch;
    struct Student S;
    fp = fopen("Student.dat","r");
    if(fp == NULL)
    {
        printf("\nCan't open file or file doesn't exist.");
        exit(0);
    }
```

```
printf("\n\tRoll\tName\tMarks\n");
while (fread(&S, sizeof(S),1,fp)>0)
    printf("\n\t%d\t%s\t%f",S.roll,S.name,S.marks);
fclose(fp);
Return 0;
}
```

Example 2

```
#include<stdio.h>
#include<stdlib.h>
struct Student
{
    int roll, temp;
    char name[30];
    float marks;
};
int main()
{
    FILE *fp;
    char ch;
    struct Student S;
    fp = fopen("Student.dat","w");
    if (fp == NULL)
    {
        printf("\nCan't open file or file doesn't exist.");
        exit(0);
    }
```

```
do
{
    printf("\nEnter Roll : ");
    scanf("%d",&S.roll);
    printf("Enter Name : ");
    scanf("%s",S.name);
    printf("Enter Marks : ");
    scanf("%f",&S.marks);
    fwrite(&S,sizeof(S),1,fp);
    printf("\nDo you want to add another data (y/n) : ");
    while( (ch=getchar()) == EOF );
    /* this while is used to remove the new line character in the
       input buffer */
    ch = getchar();
    } while(ch=='y' || ch=='Y');
    printf("\nData have been written successfully.");
    fclose(fp);
    return 0;
}
```


Summary

- Introduction to C programming language, compiling with gcc
 - Basic data types (char, int, float)
 - Basic programming constructs (if ... else conditionals, while loops, for loops)
 - Basic data structures (atomic data types, arrays, structures)
 - Functions
 - Basic I/O
- Suggested reading (Moffat):
 - Introduction to C ... Ch.1; Ch.2.1-2.3, 2.5-2.6;
 - Conditionals and loops ... Ch.3.1-3.3; Ch.4.1-4.4
 - Functions Ch. 5
 - Arrays ... Ch.7.1,7.5-7.6
 - Structures ... Ch.8.1
 - File Operations Ch. 11

Useful Links

- GCC compiler <https://gcc.gnu.org/install/download.html>.
- Eclipse C/C++ IDE <https://www.eclipse.org/downloads/packages/>
- How to Install Eclipse C on Windows?
https://www3.ntu.edu.sg/home/ehchua/programming/howto/EclipseCpp_HowTo.html
- https://www.onlinegdb.com/online_c_compiler
This link provides an online editor and compiler for C
- C tutorials <https://www.tutorialspoint.com/cprogramming/index.htm>