Summary of C language

The following C notions are assumed to be known by all students who are enrolled in 159.234.

General structure of a C program:

```
/* include statements */
#include <stdio.h>
#include <string.h>

/* define statements */
#define MAX 2000

/* declarations of functions and global variables */
void subfunc(void);
int i,j;
char c;
float x,y;
char s[80];

/* main program */
int main() {
   statements...
}

/* functions used by main */
void subfunc(void) {
   statements...
}
```

Comments:

Can occur anywhere that whitespace is allowed. Start with **`/*`** end with **`*/.`**

Indentation rule: Indent one further tab stop when a **`{`** is encountered

Remove one tab stop from the indentation when a **`}`** is encountered

Functions:

Functions are used to structure programs. They must be declared before being used.

Variables: Are used to store data.

Accessing a variable does not affect its contents. Assigning to a variable overwrites any previous values.

Variables must be declared before they are used. Names must start with a letter, subsequent characters can be letters, numbers or the underscore. They can be any length.

Those declared at the top of the file are GLOBAL - all functions can use them.
Those declared inside functions are LOCAL to a function and can only be used by it.

There are three simple types of variables:

```
char
int
float
```

Strings are created by using **`arrays`** of **`chars`**, individual characters within strings can be accessed by using the notation **`s[i]`** where **`i`** is zero for the first element of the string.
Text strings are enclosed in double quotes. Characters are enclosed in single quotes.
Special characters can be entered using a **`\ - \n`** gives a new line **`\\`** a backslash.
The last element in a valid string is the **`NUL`** character - **`\0`**

Assignment statements:

Are used to assign values to variables.

```
i = 3;
x = 4.5;
c = 'a';
strcpy(s,"a string"); /*must use functions for strings*/
```

Expressions:

Assignments can also include expressions involving the operators **`+ - * / %`**.
Where the **`%`** operator forms the remainder after division. Integer division always truncates.
Use round brackets to force the order of evaluation.

```
i = (i*j) + 3;
c = s[3];
i = c - 48; /*converts a char digit to a number*/
strcpy(s,t);
```

Shortcut operators are **`++, += -=`** etc

```
i++;  ++i;  i--;  --i;
j += k;  j *= 4
```

Output:

The standard library routine **`printf`** is used for output. **`printf`** has at least one argument.
The first argument is a text string, which may contain format specifers.
When **`printf`** encounters a format specifier it prints the next argument in that form.

`%d`	for integers
`%c`	for characters
`%f`	for real numbers
`%s`	for strings
`%%`	to output one % sign

Format specifers can contain an optional field-width. This specifies the minimum number of characters to be used on output.
An optional precision can be used to specify either the maximum field-width (for strings), whether leading zeroes should be printed (for integers) or the number of decimal places (for reals).

```
printf("%12.4f %8s %c %d\n",x,buffer,c,fw,i);
```

Input:

`gets` for complete lines of text.	**`gets(s1);`**
`getchar/getch` for characters.	**`a = getchar();`**
`scanf` for numbers and words	**`scanf("%d %s",&i,s2);`**

`gets` discards the newline character. **`getchar/getch/scanf`** do not.
`scanf` uses the same format specifiers as **`printf`**, use **`&`** for simple types.

Loops (iterations):

There are two different loop structures: pre-test and post-test.
Pre-test loops are implemented in C using the **`while`** and **`for`** loops.
Post-test loops (which are rarely used) use the **`do while`** loop.

```
   while (test) {
      statements...
   }

   for (initialisation;test;end of loop increment) {
      statements...
   }

   do {
      statements...
   } while (test);
```

`test` uses **logical operators**:**`<,   <=,  ==`**(equals), **`!=`**(not equals), **`>=, >,`**
`&&`(and), **`||`**(or), **`!`**(not)

```
   i = 1;
   while (i < 10) {
      printf("the value of i is %d\n",i);
      i++;
   }

   for (i=0;i<10;i++) {
      printf("the value of i is %d\n",i);
   }
```

Conditional statements:

The **`if`** and **`switch`** statements allow the program to chose between several options.

```
   if (test) {
      statements...
   }


   if (test) {
      statements...
   } else {
      statements...
   }

   if (test) {
      statements...
   } else if (test) {
      statements...
   } else {
      statements...
   }

   switch (int or char) {
   case number or character:            /* eg case 1:  for ints*/
      statements...
      break;
   case number or character:            /* or case 'A': for chars*/
      statements...
      break;
   .
   .
   default:
      statements...
   }
```

String functions:

Assignment statements and tests only work for simple types.
Functions available in the standard library must be used for strings:

`strcpy(s1,s2);`
copies string **`s2`** into string **`s1`**

`strlen(s1);`
returns the length of string **`s1`**

`strcmp(s1,s2);`
compares **`s1`**with **`s2`** alphabetically.

`s1`< **`s2`**	returns a negative value
`s1`== **`s2`**	returns zero
`s1`> **`s2`**	returns a positive value

`strcat(s1,s2);`
adds string **`s2`** onto the end of **`s1`**. **`s2`** is not affected

`strncpy`, **`strncmp`** and **`strncat`** are similar to above but with a third argument - the maximum number of characters in the string that the function should handle.

Functions: can **return** values, and have **arguments** or **parameters**:

eg:

```
int fct1(int i,int j);   /*returns an int, two int parameters*/
char fct2(char c);       /*returns a char, 1 char parameter */
void fct3(char *s);      /*no return value, 1 char string */
```

There must be the same number of arguments in the function call as there are parameters in the function prototype.

Parameters must also be of compatible type.

Arguments which are simple types, have their values copied into the corresponding formal parameter. For arrays, a pointer to the first element of the array is copied instead.

The **return** statement can be used anywhere within a function.
Its value (optionally enclosed in ()'s) is the returned value of the function.

Global variables are declared at the beginning of the program. If they are not initialised, they default to the value 0 - however it is good practice to explicitly initialise them.

Local variables are declared within a function. They are not automatically initialised.

A program may force its own termination inside any function by use of the **exit** function.

Arrays: can be of any type eg:

```
int m[100];
```

```
    float f[10];
    char s[80];
```

To access any particular element of an array we use eg:

```
        m[4] = 3;
        f[i] = f[i+1];
```

Individual elements of an array, which are simple types, can be acted on using the standard C operators. Actions on whole arrays may only be performed via function calls.

Only a **pointer** is copied to a function when an array is used as an argument.

Arrays have many uses including: storing many similar data items, counting up categories and looking up items (table look-up).

Loops and arrays are used extensively. Typical actions on arrays are:
finding maximum/minimum values, averaging, summing values and sorting.

Arrays may have more than one dimension:

```
        int mm[10][50];
        char sarray[25][80];
        float x[4][5][6];
```

declares:

- a two-dimensional array of ints.
- an array of 25 strings.
- a three dimensional array of reals.

Individual elements can be accessed:

```
        sarray[10][4]
```

accesses the 5'th character of the 11'th string in the array.

`sarray`	is an array of strings
`sarray[i]`	is a string
`sarray[i][j]`	is a character within a string

Initialising variables: variables may be initialised when they are declared:

```
    char c = ‘A’;
    int i = 10;
    char s[80] = “hello world”;
    int m[5] = {4,7,5,9,1};
    float x[2][3] = {
       {1.2,4.0,5.6},
       {2.2,0.0,1.3}
    };
    char s[3][80] = {
       “string one”,
       “string two”,
       “string three”
    };
```

Structures: Allow the programmer to group similar data items together.

```
struct person {
   char surname[80];
   char forename[80];
   int sex;
   int age;
};
```

declares a structure called person. Variables may now be declared with this structure:

```
struct person x;
```

The individual elements within a structure are called **fields** and can be accessed using the dot operator

```
x.age = 26;
strcpy(x.surname,”Kay”);
```

New name for existing types: can be defined by the programmer:

```
typedef struct person person_type;
typedef unsigned char byte;

person_type x;
byte a;
```

Pointers: are variables that hold the address of other variables.

```
To declare a pointer:

int *p;          /* p is a pointer to an int */
char *s;         /* s is a pointer to a character */
stuct person *q; /* q is a pointer to a person structure */
```

Addresses of variables can be found with the **&** operator:

```
   p = &i;       /* p contains the address of i, p 'points' to i */
   s = &filename[0];   /* or s = filename */
```

The item that the pointer points to can be accessed using the ***** operator (or **->** for structs)

```
   (*p) = 6;     /* this alters the value of i */
   (*s) = ‘A’;   /* this alters the value of filename[0] */
   q->age = 26;
```

If a function wants to alter the value of its parameters, then **pointers** must be used.

```
   fct1(&i);     /* pass across a pointer, the address of i */
   ...
   void fct1(int *ip) {
      (*ip) = 6;  /* alters the value of i to 6 */
   }
```

Arithmetical operators can be used on pointers

```
        p++;
```

makes **p** point to the next item. This is particularly useful when writing efficient code that handles arrays of items. If we add an integer to a pointer we get a new pointer. If we subtract two pointers we get an integer.

Pointers can also be assigned the address of unused memory - dynamic memory.
malloc is a function that obtains unused memory from the operating system.

```
        p = malloc(128);
```

This memory can be returned to the system using

```
        free(p);
```

Files: Can be accessed by using library routines:

All file I/O routines use a **FILE** pointer variable.

```
FILE *f;

     f = fopen(filename,mode);
     /* opens a file for either reading, writing or appending (r,w,a) */

     fprintf(f,”%s”,s);
     /* writes to a file - similar to printf*/

     fgets(string,size,f);
      /* the new line is NOT discarded */
     fscanf(f,”%d”,&i);
     /* read from a file*/

     fclose(f);
     /* closes the file*/
```

- **fopen** may fail if you try to open a non-existent file for reading.
  In this case it returns the value **NULL**.

- **fscanf** and **fgets** fail if they try to read past the end of a file.
  fscanf returns the value **EOF**, **fgets** returns the **NULL** pointer.

- **sprintf** and **sscanf** work like **fprintf** and **fscanf**, except the first argument is a character string, where the data is either read from or written to.