CAB202 Exam Cheat Sheet

Downsides of C

- Compilation adds complexity
- Few data structures, small standard library
- Typically more code required for a given application
- Not memory-safe and only weakly-typed; easy to write buggy code (eg. buffer overflow)

Rational Operators

Operator	Description	Example
==	Equal	1 == 1
!=	Not Equal	1 != 0
>	Greater	3 > 2
<	Less	3 < 4

Common Problems

- Unlike other languages, C doesn't have Boolean data type.
- Any non-zero value is true, 0 is false
- Be sure to use == to test equality, = for assignment.
- Identation

Logical Expressions

Operator	Description
&&	AND
||	OR
&	BITWISE AND
|	BITWISE OR
^	BITWISE XOR
&=	AND EQUAL
|=	OR EQUAL
^=	XOR EQUAL

Break Statements

- Can stop a loop early by using the break statement

```
int i = 1000;
while ( i < 2000) {
// Stop when we find a multiple of 87
if ( i % 87 == 0) { break ; }
i ++;
}
printf ( " % d \ n " , i );
```

Loops

While loops are the simplest loop in C. If condition is true then the loop is executed over and over until it becomes false.

```
while ( condition ) {
// body of loop
}
```

Do...while loops are like while loops, but the condition is evaluated after the body so the body is always executed at least once.

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

A "For" Loop is used to repeat a specific block of code a known number of times.

```
for ( start_statement ; condition ;
    end_statement ) {
// body of loop
}
```

Assignments in conditions

Assignments are expressions, so we can get a condition to do double duty!

```
int i;
// Keep scanning as long as we get something
while ( r = scanf ("%d,%d",&i,&j) > 0) {
// process i , j , r
   }
```

Strings

- In C, a string is **array of char**
- "A" is a **string** constant
- 'A' is a **char** constant
- Strings end with a **NULL** character
- String length must be specified to ensure it does not read/write beyon the end of the memory allocated

Arrays

Arrays can be used to store:
- Numbers in a Vector
- Records in a databse
- Characters in a string

Initialisation:

```
// initialise with list , size = length of list
int a [] = { 1 , 2 , 3 };
// initialise with list , size explicit
// If list is to short , remaining elements set
    to 0
int b [4] = { 1 , 2 , 3 };
// Initialise with a loop
int c [4];
for ( int i = 0; i < 4; i ++) {
c [ i ] = -1;
}
```

We can also do multi-dimensional arrays which use more than one index:

```
// Store a matrix in a 2 d array
double matrix [3][3] = { { 1 , 0 , 0} ,
{ 0 , 1 , 0} ,
{ 0 , 0 , 1} };
// Store a b & w image
unsigned char picture [1920][1020];
```

Array variables are just pointers!

```
- int * pX = &x
in English means an integer pointer named 'pX'
    is set to the address of x
- int A[10]; int* p = A; p[0] = 0; makes
    variable p point to the first member of
    array A.
```

Pointers

Pointer Definition

A pointer is a variable that **stores the memory address of another variable as its value**. A pointer is created with the * operator.

scanf() and strings

```
// We must allocate some space for a string
char buffer [100];
// scan in a string until the next whitespace
// Note : buffer is already a pointer ! No &
scanf ( " % s " , buffer );
// Scanf will happily write beyond the end of
    buffer .
```

Printing strings

```
// Use %s to print a string with printf ()
char * mystring = " Hello world ! " ;
printf ( " mystring = % s \ n " , mystring );
```

```
// Using a string variable as a format string :
char * myformat = " mystring = % s " ;
printf ( myformat , mystring );
```

```
// puts () prints the string followed by newline
puts ( " Mystring = " );
puts ( mystring );
```

rand()

The rand() function returns pseudorandom numbers between 0 and RAND_MAX:
rand() is called only once using a specified seed.
If no **seed** is specified then it will always be the **default seed**.

```
int r = rand (); // between 0 and RAND_MAX
int N = 100;
int s = rand () % N // 0 <= s < N
```

By default rand() will always return the same sequence of numbers!. This is why we provide different seeds using srand()

srand()

- Provide seed using srand() before calling rand()
- If the seed changes on each run then we get a new sequence of numbers
- Common to use current time as a seed

Functions

Functions allow reusing same code in many places and on different data

```
int myfunction (int a , int b) {
return a + b ;
}
// Using a function
int x = myfunction (3 , 5);
// x = 8
// myfunction can now be called anywhere in the
    code
```

Functions *can* be passed like variables using pointers

```
int addInt(int n, int m) {
    return n+m;
}
//define a pointer to a function which receives
    2 ints and returns an int:
int (*functionPtr)(int,int);

//use as value
int sum = (*functionPtr)(2, 3); // sum == 5

//use as function
int add2to3(int (*functionPtr)(int, int)) {
    return (*functionPtr)(2, 3);
}
```

Pass by Reference Vs Value

Pass By Value	**Pass By Reference**
Makes a copy of the actual param	Address of the actual param passed to func
Changes made inside function **do not reflect** original value	Changes made inside function **reflect** original value
Function gets a copy of the actual content	Function acesses the original variable's content

The stack

A stack is a **linear data structure** in which insertions and deletions are allowed only **at the end**, called the **top of the stack**

```
// Space to store items
char stack [256];
// Stack starts at highest address and grows
    down !
char * stack_pointer = stack + 255;
// Puts the value x at the top of the stack
void push ( char x ) {
* stack_pointer = x ;
stack_pointer - -;
}
// Removes the last value by the function push
char pop () {
stack_pointer ++;
return * stack_pointer ;
}
```

Bitwise Operations

Operator	Name	Operation
$\sim$	bitwise NOT	1 if 0, 0 if 1
&=	bitwise AND	1 if both are 1, otherwise 0
|	bitwise OR	0 if both 0, otherwise 1
^	bitwise XOR	0 if the same, 1 if different
<<	Left shift	move all bits left, fill right with 0
>>	Right shift	move all bits right, fill left with 0

OR and AND for any bit b

```
b | 1 == 1
b | 0 == b
b & 1 == b
b & 0 == 0
```

Bitwise Masks

```
Setting Bits:
x = 0b1010 ;
mask = 0b0011 ;
z = x | mask ;
// z == 0b1011

Clearing Bits:
x = 0b1010 ;
mask = 0b0011 ;
z = x ^ mask ;
// z == 0b1001

Testing Bits
x = 0b1010 ;
mask = 0b0011 ;
z = x & mask ;
// z == 0 b0010
```

Micro-Controllers

Registers

Storage with **8 bits capacity**
- Connected to **CPU** (accumulator)
- Operations on their content require **only one** instruction

Data Direction Registers (DDRx)

Configured to specifiy which of the 8 bits is used for **output(1)** or **input(0)**

Data Register (PORTx)

Writes output data to port

Input Pins Address (PINx)

Reads input data from port

```
unsigned char temp; // temporary variable
temp = PINB; // read input
```

Serial Communications

Serial Vs Parallel

Serial	Parallel
One data bit is transceived at a time	Multiple data bits are transceived at a time
Slower	Faster
Less number of cables required	Higher number of cables required

Synchronous Vs Asynchronous

Synchronous	Asynchronous
Send & Receiver clocks synchronised	Not synchronised
Faster	Bytes are enclosed between start and stop bits
Example: Serial Peripheral Interface (SPI)	Example: Universal Sync/**Async** Receiver/Transmitter (USART)

USART

1) **Asynchronous Normal Mode**
— Data is transferred at the BAUD rate set in UBBR register
— Data is transmitted/received asynchronously, not using clock pulses.
2) **Asynchronous Double Speed Mode**
— Everything is like normal mode but doubled
3) **Synchronous Mode**
— Requires both data and a clock
— The data is transmitted at a fixed rate

To use must specify

1) Baud Rate
2) Number of data bits encoding a frame or char
— Usually 10: 1 start, 8 data, 1 stop.
3) The sense of the parity bit
4) Number of stop bits
Minimally you'll need:

- uart_init(ubrr) //Initialise the baud rate
- uart_putchar(unsigned char data)
- unsigned char c = uart_getchar(void)

Timers

- AVR timers run asynchronous to main AVR core
- Use clock pulse as parallel increment
- When timer reaches TOP value, reset to 0 (overflowed)
- When overflowed, sends signal which can be used for interrupts

Pin	**Output Compare Register**
Pin 3	**OC2B**
Pin 5	**OC0B**
Pin 6	**OC0A**
Pin 9	**OC1A**
Pin 10	**OC1B**
Pin 11	**OC2A**

ADC

ADC Defintion

Analog to Digital converters are used to create an **electric quantity**(Bit value) to vary directly like a **continuous variable** (temperature)

Registers Involved

1. **ADMUX** - used to select reference voltage source
2. **ADCSRx** - used to tune aspects of the conversion(Auto-Trigger, Off/On, Interrupt Enable)

PWM

PWM Definition

It is a way of simulating an **analog signal** via a **digital pin**. It does this buy turning **off** and **on** at different **frequencies** to achieve desired effects

Registers Involved

1. **TCCRx** - used to determine when to turn off and on