An incomplete Summary of C Syntax:

Note: This <u>doesn't</u> cover all of the materials that you need to know for your midterm exam, so don't forget to review your lecture slides.

Two typical file structures:

```
int main()                              /* This could be a complete program */
{
  sequence_of_statements
  return 0;
}
```

```
#include <stdio.h>
#include ...     more header files from which definitions are needed
global and/or external declarations and/or type definitions
int main()                      /* main function and other functions  */
{                               /* in same file                       */
  local_declarations
  sequence_of_statements
  return 0;
}

void function_1( int i, double x, more_parameters )
                      /* function definitions can NOT be nested */
                      /* flat name space, make all function names unique */
{
  local_declarations
  sequence_of_statements
  /* no return needed at end of function when return type is 'void' */
}

some_type function_2( void )
{
  local_declarations
  sequence_of_statements
  return something-of-some_type;
}                               /* as many functions as desired       */
```

Executable statements:

Statements end with a semicolon (;).

Assignment statements

```
  x = 10;
  x = x+1;
  ++x;                /* same as x = x+1; */
  x++;                /* same as x = x+1;  sets a false flag to true */
```

```
--x;                /* same as x = x-1; decrements x then uses value*/
x--;                /* same as x = x-1; uses x then decrements value*/
x = x+20;
x += 20;            /* same as x = x+20; add */
x -= 20;            /* same as x = x-20; subtract */
x *= 20;            /* same as x = x*20; multiply */
x /= 20;            /* same as x = x/20; divide */
x %= 20;            /* same as x = x%20; modulo */
x = who ? 3 : y;    /* same as if(who)x=3; else x=y; */
y = sin(x) ;        /* function call use #include <math.h>  */
```

Function call with return ignored:

```
go_do_it();         /* actual parameters are optional              */
exit(0);            /* special function that terminates execution  */
```

Unconditional branches:

```
break;              /* immediate exit from loop or switch          */

continue;           /* immediate jump to end of loop ( not exit )  */
                    /* also used to exit a switch in a loop        */

return;             /* immediate exit from function, no value      */
return exp;         /* exit from function returning value of exp   */
```

Conditional branching:

```
if ( condition ) statement ;  /* basic form of if statement */

if ( condition ) statement ;
else statement_2 ;            /* optional  else  clause */

if ( condition_1 )
   if ( condition_2 ) statement ;
   else statement_2 ;         /* else  belongs to closest previous if */

if ( expression )           /* condition may be any expression        */
{                           /* zero value is false, non zero is true  */
   statement_sequence       /* { } may enclose sequence of statements */
}
else
{                           /* it is good to use braces, safer for    */
   statement_sequence       /* updates to code later.                 */
}

if-else-if ladder structure (must be space in  'else if')

if(exp_1)             /*  the first expression that is true, e.g. exp_i */
     statement_1;     /*  causes statement_i to be executed             */
else if(exp_2)        /*  then jump to end after statement_n            */
     statement_2;
else if(exp_3)
```

```
     statement_3;
...
else
     statement_n;     /*  executed if no exp_i was true, else skipped   */
```

Switch statement

```
switch ( expression )      /* constants must be unique              */
{
    case constant_1:       /* do nothing for this case              */
       break;
    case constant_2:       /* drop through and do same as constant_3*/
    case constant_3:
       statement_sequence  /* can have but does not need  { }       */
       break;
    case constant_4:
       statement_sequence  /* does this and next */
                           /* statement_sequence also*/
    case constant_5:
       statement_sequence
       break;
    default:               /* default executes if no constant equals*/
       statement_sequence  /* the expression. This is optional      */
}
```

Iteration statements

```
for ( initialization ; condition ; increment ) statement ;
                                /* The flow sequence is :              */
                                /*    initialization                   */
                                /*    test condition and exit if false */
                                /*    statement                        */
                                /*    increment                        */
                                /*    loop back to test condition      */

while ( condition ) statement ; /* keep repeating statement until the */
                                /* condition is false, zero. This is */
                                /* classical C thinking, rather than saying */
                                /* keep repeating while condition is true*/

do                              /* another form of the 'while statement'*/
{                               /* always execute once before testing   */
  statement_sequence
} while ( condition ) ;         /* test for finished at the end, all    */
                                /* other loops test first               */

for(;;) statement;              /* infinite loop, break; can get out    */
                                /* return and exit() can also get out   */
while(1) statement;             /* another form of infinite loop        */
                                /* this version is called an endless    */
                                /* and must have something else to      */
                                /* terminate the loop                   */
```

```
Note: in all iteration statements,  break; causes an immediate exit from loop
                                    continue; causes an immediate increment
```

Function definition

Function Prototype

```
  type function_name(type_1 var_1, type_2 var_2,...);
                             /* e.g. double sin(double x);   */
/* a prefix of  static  makes the function visible only in this file */
```

Function Definition

```
  type function_name(int a, float b,...)
  { function_body }

  /* in the calling function, local copy can be modified*/
```

Example of a Function Call

```
  type function_name(int a)
  { function_body }

  /* if type is not void, function call is something like below */
  type var_name = function_name(5); /* calling function by passing 5 to a */

  /* if type is void, function call is something like below */
  function_name(5);
```

Reserved words

Like everything else, must be lower case. (exactly 32 reserved words)

term	Description
auto	optional local declaration
break	used to exit loop and used to exit switch
case	choice in a switch
char	basic declaration of a type character
const	prefix declaration meaning variable can not be changed
continue	go to bottom of loop in for, while and do loops
default	optional last case of a switch
do	executable statement, do-while loop
double	basic declaration double precision floating point
else	executable statement, part of "if" structure
enum	basic declaration of enumeration type

extern	prefix declaration meaning variable is defined externally
float	basic declaration of floating point
for	executable statement, for loop
goto	jump within function to a label
if	executable statement
int	basic declaration of integer
long	prefix declaration applying to many types
register	prefix declaration meaning keep variable in register
return	executable statement with or without a value
short	prefix declaration applying to many types
signed	prefix declaration applying to some types
sizeof	operator applying to variables and types, gives size in bytes
static	prefix declaration to make local variable static
struct	declaration of a structure, like a record
switch	executable statement for cases
typedef	creates a new type name for an existing type
union	declaration of variables that are in the same memory locations
unsigned	prefix declaration applying to some types
void	declaration of a typeless variable
volatile	prefix declaration meaning the variable can be changed at any time
while	executable statement, while loop or do-while loop

Declarations have the forms

```
basic_type variable ; /* int a; */

basic_type variable_1, variable_2=value;  /* only initializes variable_2 */

prefix_type(s) basic_type modifier_type variable_1, variable_2 ,... ;
```

Basic types

Type	Description
char	character type, usually one byte (a string is array of char)
int	integer type, usually 2 or 4 bytes (default)
float	floating point type, usually 4 bytes
double	floating point type, usually 8 bytes
void	no type, typeless
enum	enumeration type (user defines the type name)

Type modifiers, prefix for basic types

Modifiers	Description
signed	has a sign (default)
unsigned	no sign bit in variable
long	longer version of type (short or long alone means short int or
short	shorter version of type long int because int is the default)
const	variable can not be stored into

Storage class, prefix for types

Prefix	Description
auto	local variable (default)
static	permanent when function exits, not auto
volatile	can change from outside influence
extern	variables are defined elsewhere, externally
register	assign variable to register

Samples of variable declarations

```
auto int xyz;            /* int no longer is default */
unsigned long int pqr;
extern int global_stuff, remote_function();
register int quick;
void just_do_it();   /* called a function prototype  */
```

Expressions

```
octal and hexadecimal

   01777  0x248fff

   long decimal, octal and hexadecimal

   123456789L  017777777L  oxFFFFFFFFL
```

variable names	1 to 32 characters including alphabetic, numeric and underscore some_name ELSE Else z999
numbers	integer and floating point like in FORTRAN 1 12345 .1 1. 1.5E-20
character	single character between apostrophes plus special backslash codes 'a' 'Z' '\n'
string	characters between quotes plus special backslash codes a null is automatically placed at end of every string "Hello" "good bye \n"

```
let exp, exp1, exp2, exp3  etc. stand for expressions
```

```
expressions are:

  variables
  numbers
  characters
  strings
  unary_operator expression
  expression binary_operator expression
  ( expression )
  variable post_operator    ( ++ and -- )
```

operator precedence and associativity is

from highest to lowest:

LR	**( ) [ ] -> . x++ x--**
RL	! ~ - + ++x --x * & sizeof (type)
LR	* / %
LR	+ -
LR	<< >>
LR	< <= > >=
LR	== !=
LR	&
LR	^
LR	|
LR	&&
LR	||
RL	? :
RL	= += -= *= /= %= >>= <<= &= ^= |=
LR	,

```
LR means associate left to right, RL means associate right to left
This is why  a = b = c = 1;  /* works, a,b and c get set to 1 */
But          int a,b,c = 1;  /* fails, only c gets set to 1   */
```

operator definitions :

```
  ( )  grouping parenthesis, function call
  [ ]  array indexing, also  [ ][ ]  etc.
  !    relational not, complement, ! a  yields true or false
  ++   increment, pre or post to a variable
  --   decrement, pre or post to a variable
  -    unary minus, - a
  +    unary plus,  + a
  sizeof  size in bytes,   sizeof a     or  sizeof (int)
  (type)  a cast, explicit type conversion,  (float) i
  *    multiply, a * b
```

```
  /    divide, a / b
  %    modulo, a % b
  +    add, a + b
  -    subtract, a - b
  <    less than, result is true or false,  a %lt; b
  <=   less than or equal, result is true or false,  a <= b
  >    greater than, result is true or false,  a > b
  >=   greater than or equal, result is true or false, a >= b
  ==   equal, result is true or false,  a == b
  !=   not equal, result is true or false,  a != b
  &&   relational and, result is true or false,  a < b && c >= d
  ||   relational or, result is true or false,  a < b || c >= d
  ?     exp1 ? exp2 : exp3  result is exp2 if exp1 != 0, else result is exp3
  =    store
  +=   add and store
  -=   subtract and store
  *=   multiply and store
  /=   divide and store
  %=   modulo and store

  zero evaluates to false ( a null pointer has a zero value )
  non zero evaluates to true ( !null_pointer is true )
```

Backslash codes for in character constants and strings

```
These can be used as characters when enclosed in apostrophies
  \a  alert, audible alarm, bell, ^G
  \b  Backspace, ^H
  \f  Form feed, new page, ^L
  \n  New line, carriage return and line feed, ^M^J
  \o  Octal constant, \oddd, \ddd
  \r  Carriage return, no line feed, ^M
  \t  Horizontal tab, tab, ^I
  \v  Vertical tab, ^K
  \x  Hexadecimal constant,  \xdd    0 <= d <= F
  \0  null, zero value character, ^@
  \"  Quote character
  \'  Apostrophe character
  \\  Backslash character
  \?  Question mark character
  \ddd  Octal character value 0 <= d <= 7
```

Format commands for printf() and scanf()

```
  %c  a single character, char
  %d  a decimal number, int   %hd is for short  %ld is for long
  %e  a floating point number, float in scientific notation, %E for 1.0E-3
      %le is for double, %Le is for long double
  %f  a floating point number with decimal point   %10.4f  10 wide  .dddd
      %lf is for double, %Lf is for long double
  %g  a floating point number, %f or %e as needed, %G for capital E
      %lg is for double, %Lg is for long double
```

```
  %h  an unsigned hexadecimal short integer (scanf only), old usage
  %i  an integer, int  %hi is for short int,  %li is for long int
  %n  pointer to integer to receive number of characters so far, int *i
  %o  an unsigned octal number, unsigned int  %ho and %lo for short and long
  %s  a string ( must be null terminated ! ), use %s for scanf
  %u  an unsigned decimal integer (printf only), unsigned int  %hu %lu
  %x  a hexadecimal number, %X for capital ABCDEF, unsigned int  %hx %lx
  [...]  string, x[]
  %   none
Text may be around format characters for printf(), The list follows the
single quoted format string.

Output is right justified unless format  %-s or %-10.4f is used.
%+... causes plus sign to be printed.

Field is expanded if not enough room to print.
numeric data into scanf() must be separated by space, tab or newline, not
comma.
characters between % in scanf() format must be in input and are ignored.
```

Source: http://www.csee.umbc.edu/courses/104/spring02/burt/C_summary.html

Edit: Sadegh Charmchi