**CS2100 Computer Organization**
**A Quick Introduction to C**

## 1. Introduction

This document gives a very quick introduction to the C Programming Language. It is assumed that the reader is already reasonably proficient in programming methodology, and hence this document does not explain basic programming concepts like data-types and functions. It will however explain concepts that are unique to C, like pointers. The objective of this document is not to teach how to program in C, but to make the relevant sections of the CS2100 lecture notes comprehensible to programmers who are unfamiliar with the C programming language.

## 2. Data Types in C

Unlike languages like Python and JavaScript, C is a strictly typed language. C also strictly requires that all variables are declared before being used. The code fragment below shows how to declare a signed integer and a floating point number:

```
int x; // x is a signed integer
float y; // y is a floating point number (i.e. it holds real numbers)
```

### 2.1 Numerical Data Types

The common C numerical data types are:

| **Type** | **Meaning** | **Size and Encoding** | **Range** |
|---|---|---|---|
| char | Signed byte | 8 bits, 2's complement | -128 to 127 |
| unsigned char | Unsigned byte | 8 bits, unsigned | 0 to 255 |
| short | Signed integers | 16 bits, 2's complement | -32768 to 32767 |
| unsigned short | Unsigned integers | 16 bits, unsigned | 0 to 65535 |
| int | Signed integers | 32 bits, 2's complement | -2,147,483,648 to 2,147,483,647 |
| unsigned | Unsigned integers | 32 bits, unsigned | 0 to 4,294,967,295 |
| long | Signed integers | 64 bits, 2's complement | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| unsigned long | | 64 bits, unsigned | 0 to 18,446,744,073,709,551,615 |
| float | Signed floating point | 32-bit IEEE 754 | +/- 1.2e-38 to 3.4e+38 |
| double | Signed float | 64-bit IEEE 754 | +/- 2.3e-308 to 1.7e+308 |

## 2.2 Alphanumeric Data

All data in C is (or can be) represented as integers. Characters are represented by 8-bit "char" integers (based on the ASCII table) and strings are represented as an array of char:

```
char mystr[128]; // This is a string of 127 characters.
```

Notice that an array of 128 char stores a string of 127 characters. This is because all strings in C must be terminated by a '\0' (ASCII 0) character (called a NULL character).

## 2.3 Boolean Values

C does not have Boolean values; True and false values are represented by integers, with 0 being False, and any non-zero value being True. Hence:

```
if(0) {
        // The statement here will never be executed since 0 is false
}

if(-1) {
        // The statement here will always be executed since -1 is true.
}
```

In C we do not assume that the "True" value is always 1. It can be any non-zero value. This can cause problems; The statement below shows the correct way to do comparisons:

```
if(x == 5) {
        // Executed if x is 5
}
```

But:

```
if(x = 5) {
        // Always executed. C assigns 5 to x, then takes the result 5 in the
        // if statement. Since 5 is true, this block is always executed.
}
```

3. C Statement Types

a. Statement Blocks

Similar to Javascript (but different from Python), C statement blocks are marked with curly {..} braces. For example the following statements form a block in C:

```
{
        printf("Hello world.\n");
        i = 4;
        printf("i = %d.", i);
}
```

Unlike Python, indentations in blocks is optional, but should be included for readability. Also unlike Javascript and Python, all C statements must be terminated by a semicolon ';'. In fact carriage returns in C are optional; C separates statements solely with the ';'. So the block above can be rewritten as:

```
{printf("Hello world.\n");i=4;printf("i=%d.",i);}
```

Because of this unique property, it is possible to write very fancy looking C source code and there the "International Obfuscated C Code Competition" held almost every year to celebrate creativity in writing C programs. The 2019 winner is shown below (this program converts text to sound using fonts as a spectrogram):

```
#include<stdio.h>
#include<stdlib.h>
#define M malloc(sizeof(
#define R fread(x,sizeof(float),Z*w,f)
#define E free(

char*G=
"   "
"    '"
"@(*AP(  "
"          "
"                  "
"                          "
"                                  "
"                                          "
"0   #@ .  \"C^*/XH\"Q2_U(D&(5*E0C 4B54DV   X    \"!9A@   89H$ !"
"0(/@@4 @(/@@(   #@P  @(\" @(    P, , P# , #Y1244^   7R 0#%)14,A#9)24$B 1"
"_)!0,$Y145%R Y145$^&!8!T! #9)24DV#Y)24DP  ;&P     -C4 $$B% @ !04%!04 @4(D$ #!(1"
"4 @#U574$^'^0D)!_&Z1D9'_$*!@8%^'Z!@8'_(&1D9'_(\"0D)#_%Z1D8%^/\\0$!#_ \"!_X$ /X! 0"
"$&(-$*!#_ $! 0'_7) ,$#_7\\&&&#_'Z!@8%^&\"0D)#_'V\"A8%^&>8D)#_$Z1D9%B(\" _X\" /X"
"! 0'^/ ,_PSP/P#' /\\,<H$\"C', @'R# ,&AD8F'(&!_P   ,,,,     _X&!! @0\" 0 "
"$! 0$!  @0(   \\5%14\" X1$0G_!$1$1$./\\2$1$. T5%14. \"0D'\\'  'Y)"
"144X \\0$ C_  !7Q    7A$! $Z# 3_ $I_H   \\0'Q ? \\0"
"$! ? X1$1$.#A$1\"1_']{1$0X @0$ @? (5%14)"
"  1$?X0!\\\" 0$>!@& 0B8!X!#@$>"
"!$*! H1#X)!04X!$9%1,1"
"(&!;A 0   _P "
" ! 0;H&!"
" @$\""
"! ("
"\0";                           int s,p,e,c,t,r,a,l;   /* l e a k a g e */

                                                    int
                                                  F(int o
                                                ,int n, int t
                                              ){char*p=(o*(136-o)
                                            >=1260)*7*(o-9)+G; for(
                                           n=!(o=-1); n=*p++64*n-*G
                                          ,(o+=3)>>2<5-t; ); return 255
                                         &n>>2*o%8; } float L[0x400];int
                                       T(float a,float b,float l,float*e)
                                      {return L[0x3ff]||!(e[l]=b*a+l**e)||
                                     !T(l*a-b**e,b,l,e+l); } float f(int x)
                                    {int w=s<<!!T(1,6.135885e-3,-1.882472e-5
                                   +l,l); x=(x%w+w)%w<<9; return(L[x/8]*(8-x%
                                  8)+L[x/8+1]*(x%8))/8; } int C(float*h,float*
                                 i,int r,int P){ int x,y,c; for(y=0; h<i; ++y,h
                                 +=r){ c=P<1; for(*h=x=0; x<8; c+=(l&P>>x&&P+(*
                                h+=f(e*(l+a*(2*x+l))*y)*f(s/2-e*a*y+e*a*y*y*(t/p
                               )/(s/p)))),++x); float z=f(y*t)*f(y*t); *h=z*(l+3*
                               z+2*z*z)/6**h/c; } return h-i-r+1; } void d(int w,
                              int n,FILE*f){int Z=w/t; int S[5]; float**P=M float*
                              )*c);for(r=0;r<c;++r){ P[r]=M float)*Z); C(P[r],P[r]
                             +Z,l,r); } float*x=M float)*Z*w); for (; ; ){ for(r=0;
                r<5         ; ++r){ int o=1; for(R,a=256; o&&--a; )for(o=t=0; t<Z;        o+=
                P[a][       t]!=x[w*t+n],++t); S[r]=a; } if(R-Z*w)break; for(r=a= -      12;r<
               5&&(a+=     a<11?22:l)<127; )for(r=-1; ++r<5&&F(a,l,r)==S[r]; );if(a     <127&&r
    ==5      )putchar(     a); } for(; --c; E P[c]}); E P); fclose(f);}int main(int     q,char**v)      {s=
   44100     ;p=25; e=     55;c=256; t=75; r=q>1?v[1][0]*c+v[1][1]:0; a=7;l=16;if(r     ==11620){d      (atoi
  (v[2]),    atoi(v[3]),  q<57stdin:fopen(v[4],"r")); return 0; } char*H=malloc(c),*  h=H; sprintf    (H,"\0"
 "Usage\n"   "    %s [-"   "h] [-d #c cid] file1 file2 ... (or stdin)"+(q-1&&r==11624  ),v[0]);if(*  H)q=1;int
 w=2>q?1:q -1; FILE**f=M FILE*)*w); f[0]=stdin; if(q>1)for(r=0; r<w; f[r]=fopen(v[r +1],"r"),++r); int l=s/t
 *w; float *b=M float)*l ),*t=b+l; int*i=M int)*w); for(; ; ){ int x=*h&&l>ungetc(* H++,*f); for(r =w; r--;)
{ i[r]=getc (f[r]); if(i[r]<0)i[r]=!++x; } if(x==w)break; while(++r<6){ while(C(b++,t,w,F(*i++,r< 5,r))); i-=
w; fwrite(b-=w,sizeof(float),l,stdout); } fflush(stdout); } while(fclose(f[--w]),w); E f); E b); E i); E h);}
```

b. Iterations

C supports several types of iterations with slight differences:

i) The For Loop

The most basic C iteration is the for loop. It consists of 3 parts: An initializer, a continuation condition, and a modification operation, separated by semi-colons. To count from 0 to 9 we would do:

```
for(i=0; i<10; i++) {
        … C statements …
}
```

Due to its flexibility, the for statement is very powerful; we can for example count downwards from 9 to 0:

```
for(i=9; i>=0; i—) {
        … C statements …
}
```

Each part is optional; If we don't want to initialize i, we can do:

```
for(; i <=9; i++) {
        … C statements …
}
```

You can also implement a loop that counts infinitely from any initial value:

```
i=5; // Some statement that sets i to 5

for(;;i++) {
        // i will start from 5 and increment indefinitely.
}
```

If we had a string "mystr" we can count how many characters are in the string using:

```
for(ctr=0; mystr[ctr] != 0; ctr++);
```

When this for loop ends ctr will contain the number of characters in mystr.

Lastly the for loop can also be used for infinite loops, simply by leaving out every part:

```
for(; ; ;) {
        // This is an infinite loop
}
```

ii) The While Loop

The while loop works similarly to the for loop, except that the "while" statement itself contains only the continuation condition; initialization and update are done separately. Our "string count" operation would be written as:

```
ctr = 0;

while(mystr[ctr] != 0) {
        ctr++; // Increment ctr
}
```

Since any non-zero value is true, we can do an infinite loop using while by doing:

```
while(1) {
        // This body executes infinitely
}
```

iii) The Do..While Loop

The do..while loop is fairly unique to C. Since the continuation condition is tested only at the end of the body, this means that the body will always be executed at least once:

```
do {
        ctr++;
} while(mystr[ctr] != 0)
```

Here ctr will be incremented as least once, even if mystr[0] is 0.

c. <u>Conditional Statements</u>

i) The if..else statement

The if..else statement in C is fairly straightforward:

```
if( .. condition.. ) {
        // This part is executed if the ..condition.. is true (non-zero)
} else {
        // Otherwise this part is executed.
}
```

C however does not have an explicit “elif” (else if) statement, and this must be handled with:

```
if(condition1) {
        …
} else
        if(condition2) {
                …
        }
        else
        {
                … Executed if condition2 is false…
        }
```

ii) The ternary if..else statement

Just as in Javascript, we can use the ternary if..else operation:

```
(<condition> ? <if part> : <else part>)
```

So if we did:

```
y = (x > 0 ? 3 : 2);
```

The “y” variable would be set to 3 if x is greater than 0, and 2 otherwise.

iii) The switch statement:

The C Programming Language has the switch statement, which can handle multiple choices without having to deeply nest if..else statements. The statement takes the form:

```
switch(<integer variable>) {
        case 1:
                … something …
                break;

        case 2:
                … something else …
                break;

        case 3:
                .. something else again …
                break;

        default:
                .. Every other case is handled here ..
}
```

The switch statement only works with integer variables (including “char”, which if you recall is a 1-byte integer). You also need a “break” statement after each case, to force execution to exit from the switch statement, instead of falling through all the other cases. The “default” keyword is used to catch all other cases that are not explicitly stated. You can use switch to implement a menu system:

```
choice = readMenuOption(); // Get menu option from user

switch(choice) {

        case ‘a’: callOptionA();
                break;

        case ‘b’: callOptionB();
                break;

        case ‘c’: callOptionC();
                break;

        default:
                printf(“Don’t understand. Please choose a, b or c.”);
}
```

4. Pointers

The idea of pointer variables is arguably the most difficult concept in C to grasp. Every memory location in a computer is indexed with an address (also called a "memory location"). All variables in C must be stored in memory, and a pointer variable simply stores the address of another variable.

So if we had:

```
int *x;
```

This declares a pointer variable x, and it will point to another variable of type int.

Let's suppose variable y is stored in location 0x5004. The figure below shows memory contents from addresses 0x5000 to 0x5008.

| Address | Value | Description |
|---|---|---|
| 0x5000 | 55 | Variable z |
| 0x5001 | | |
| 0x5002 | | |
| 0x5003 | | |
| 0x5004 | 72 | Variable y |
| 0x5005 | | |
| 0x5006 | | |
| 0x5007 | | |
| 0x5008 | | |
| … | … | … |

Notice firstly that every memory location has an "address", and stores "values". The Description column tells us what the memory holds, but is not actually stored anywhere – it is there just for our information. Also notice that each variable occupies four memory locations; this is because each int is 32-bits long, which is four bytes, and each address refers to an individual byte in memory (byte-addressable memory).

Now if we did:

```
x = &y;
```

The "&" operator, called the "address-of" operator, returns the address of y. This is stored into x, which is a pointer variable and stores addresses of other variables. Now if we did:

```
printf("x is %x", x);
```

The program will tell us 0x5004, which is the address of y.

Now what can we do with this? We could make two address variables point to exactly the same variable:

```
int *a;

a = x; // Now both a and x point to y.
```

This is very useful in “call by reference” parameter passing which we will see shortly.

Now that x (and a) point to y, can we use them to access (and change) y’s value? Yes, using the “de-referencing” operation.

In a fit of ingenuity, the designers of C decided to use the same “*” operation (the one used to declare pointer variables) to also access the value pointed to.

So if we did:

```
printf(“The value pointed to by x is %d.”, *x);
```

We would get:

```
The value pointed to by x is 72.
```

This is the value that is in y.

Now remember that a is also pointing to y. If we do:

```
*a = 123;
```

Our memory will now look like this:

| Address | Value | Description |
|---|---|---|
| 0x5000 | 55 | Variable z |
| 0x5001 | | |
| 0x5002 | | |
| 0x5003 | | |
| 0x5004 | 123 | Variable y |
| 0x5005 | | |
| 0x5006 | | |
| 0x5007 | | |
| 0x5008 | | |
| … | … | … |

This is because C uses the “*” de-referencing operator to access a, getting the address 0x5004, then going to that address and storing 123.

5. Functions

As in other languages, a function in C is a unit that takes inputs, performs some sort of transformation on the input and produces an output. However as with other languages, a function may not necessary take inputs (called arguments) and may not necessary produce a value.

A C function is declared as follows:

```
<return type> <function name>(<param1 type><param1>, <param2 type>
        <param2>, …) {

        .. function body ..
}
```

The function below returns the sum of two integers:

```
int sum(int a, int b) {
        return a + b;
}
```

The function below halves the argument:

```
float half(int a) {

        return a / 2.0;
}
```

The function below doesn't take any arguments nor does it return a value:

```
void say() {
        printf("Say what?\n");
}
```

## 5.1 Call by Value vs. Call by Pointer

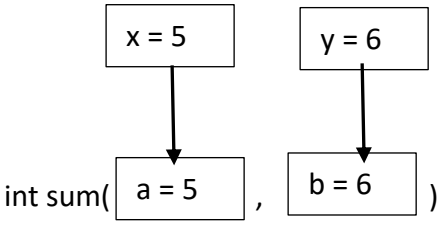
All arguments are "passed by value" to functions (i.e. in C, all function arguments are "call by value"). To understand what this means, let's look at our earlier sum function:

```
int sum(int a, int b) {

        return a + b;
}
```

Now let's declare two variables x and y and call sum with them:

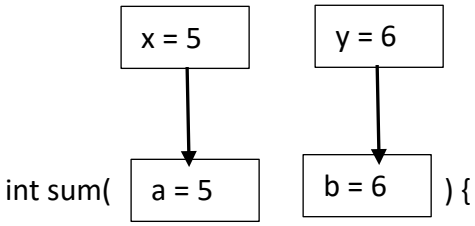
```
int x = 5, y = 6;

z = sum(x, y);
```

The diagram below shows what happens:



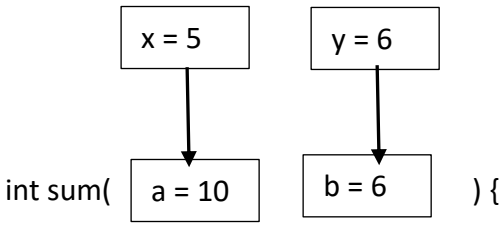

Here 5 is copied from the argument x into the parameter a, and 6 is copied from the argument y into the parameter b. This means that a and b are second copies of x and y.

This has implications. Supposed we modify a within the body:




```
        a = 10;
}
```

The picture below shows what happens:




Notice that while a is changed to 10, x remains as 5. Therefore we cannot use this as a means of passing back values through the parameters.

This is where call-by-pointer comes in. We rewrite our function as:

```
int sum(int *a, int *b) {

        *a = 10;
        return *a + * b;
}
```

Now we call it with:

```
sum(&x, &y);
```

What happens now is that the ADDRESSES of x and y are copied “by value” into a and b. When we de-reference a (which points to x) in this statement:

```
*a = 10;          // a contains address of x
```

C will look at the address in a, go to that address and write in 10. Since a is pointing to x, this causes x to be modified to 10.

## 6. Conclusion

This is a very short brief introduction to the C programming language. It is not sufficient to get you started in programming C, but should be enough for you to understand the lectures.