Compound Statements

So far, we’ve mentioned statements or expressions, often we want to perform several in an selection or repetition. In those cases we group statements with braces:

i.e.

```{statement;
  statement;
  statement;
}
```

Pointers & Arrays

Pointers cause EVERYBODY problems at some time or another.

Pointers in C, are closely related with arrays.

To declare an array of char:

```char x[10] or char y[8][10] or char z[9][9][9] etc.
```

All arrays in C are indexed from 0. So for example (above) x has elements 0,1,2,3,4,5,6,7,8,9.

to declare a char pointer:

```char *cp;
```

The pointer declaration only allocates space in memory to hold the pointer (cp), NOT any data (*cp).

To refer to the pointer we use cp;

To refer to the pointed-at data we use *cp
We can say, \( cp = x \) or \( cp = &x[0] \)

\& is the “address of” operator

<table>
<thead>
<tr>
<th>Memory address</th>
<th>( N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>( x[4] ) 'o'</td>
</tr>
<tr>
<td>50</td>
<td>( x[3] ) 'l'</td>
</tr>
<tr>
<td>49</td>
<td>( x[2] ) 'l'</td>
</tr>
<tr>
<td>48</td>
<td>( x[1] ) 'e'</td>
</tr>
<tr>
<td>47</td>
<td>( x[0] ) 'H'</td>
</tr>
<tr>
<td>17</td>
<td>( cp = 47 )</td>
</tr>
</tbody>
</table>

Pointers are vital in C

Without pointers we would not be able to modify parameters passed to functions, and only pass a single value out.

In Pascal you can specify a parameter as \textit{Var} to avoid needing pointers for parameter passing. In technical terms you can pass parameters by value or by reference.

By value, means the function is given a copy of the current value of a variable.

By reference, means the function is told where in memory the variable is stored.

In C we can only pass parameters by value but can achieve the same effect as passing by reference by passing a pointer to the variable.
Imagine a function to update a bank balance. We want to add or subtract a value from the balance and return the new balance, but we want to set the return value of the function to tell us if the result leaves the account overdrawn.

Our program could look like:

```c
#include <stdio.h>

int update_balance(double *balance, double transaction);

void main(void)
{
    double current_balance = 37.5;
    if (update_balance(&current_balance, -55.30))
        printf("Overdrawn\n");
    else
        printf("OK\n");
}

int update_balance(double *balance, double transaction)
{
    *balance += transaction;
    if (*balance < 0.0)
        return 1;
    else
        return 0;
}
```

When we call `update_balance` we pass the address of `current_balance`

Inside our function we treat that address as a "pointer to double"
We alter the value pointed at by our pointer … … and then return 1 if the result is less than zero.

Our `main` function then prints a message depending on the integer value returned by our `update_balance` function. Essentially, we are using the integer as a true or false, or boolean value in this example.

This example, although trivial is important! In C this is how we pass complex data structures to functions so that those functions can modify the data.
The alternative?

Variables may be declared globally, that is outside the scope of any function. These variables may be accessed by any function.

Global variables – Unsafe, **BAD** practice (generally)  
Risk prone  
Makes code less reusable

Consider:

```c
double balance;
int x;

void my_function(void)
{
    int x;

    balance = 8.3;
    x = 7;
}
```

- Do we want this function to alter `balance`?  
  (possibly not, we might not even have written it)

- Did we mean this function to alter `x`?  
  (possibly, but how does the function writer know that `x` exists?)
Getting data out – printf();

C has a very confusing formatted print command, similar to but not the same as in Java.

printf("format string", other_arg1, other_arg2, ... other_argN);

printf("%d %d %d"), 1, 2, 3);

Produces as output: 1 2 3

Instead of constants we would usually have variables

If we want a newline at the end, we must include it in the format string

If we wish to print the value of a variable, we place a % followed by an appropriate type in the format string, then the variable as the next argument to printf().

Some appropriate types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>int in decimal</td>
</tr>
<tr>
<td>%i</td>
<td>int in decimal</td>
</tr>
<tr>
<td>%o</td>
<td>int in unsigned octal (no leading 0)</td>
</tr>
<tr>
<td>%x</td>
<td>int in unsigned hexadecimal, x = abc X = ABC</td>
</tr>
<tr>
<td>%X</td>
<td>int in unsigned hexadecimal, x = abc X = ABC</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned decimal</td>
</tr>
<tr>
<td>%c</td>
<td>a character</td>
</tr>
<tr>
<td>%s</td>
<td>a character string (terminated with a NULL char)</td>
</tr>
<tr>
<td>%f</td>
<td>double: no. of decimals specified by precision</td>
</tr>
<tr>
<td>%e</td>
<td>double: 1.47e3 notation</td>
</tr>
<tr>
<td>%E</td>
<td>double: 1.47e3 notation</td>
</tr>
<tr>
<td>%g</td>
<td>double: same as %e if exponent is less than –4, or greater than precision, else same as %f</td>
</tr>
<tr>
<td>%G</td>
<td>double: same as %e if exponent is less than –4, or greater than precision, else same as %f</td>
</tr>
<tr>
<td>%</td>
<td>print a percent sign</td>
</tr>
</tbody>
</table>

We can also specify flags to modify the behaviour

- left justify
+ prefix numeric value with a sign
' ' prefix with a space (if no sign present)
0 pad with leading zeros

We can also specify:
minimum field widths (optional),

"." precision

length_modifer  (h for short, l for long)

If we use these, all are optional but if they appear they must appear in the order:

flags – width – precision-length_modifer

E.g.

%3i       integer, three characters minimum
%03i      integer, three characters, padded with zeros
%.2f      float, 2 d.p.
%8.2f     float, 8 chars, 2 after the decimal point

For more examples, see *Tony Royce*, pages 46 & 47.
Try the exercises!

Printf can do much more than is discussed here! If you're interested consult man or a good C book.
Bitwise Operators

We've already introduced the operators:

Often for low level programming we want to operate at the bit level

Most programming languages only support operations on larger units (e.g. bytes or words)

C provides a set of low level operators

Left shift "<<" moves a bit along by one, moving a zero in.
(Another way of thinking of it is as multiplying the value by 2)

Right shift ">>" does the opposite:

And (&), Or (|), and X-Or (^) let us set or clear individual bits by using a mask.

If we AND a bit patterns with a mask, where a 1 is in the mask the value remains the same; where a 0 is the bit is set to 0.

AND is therefore useful for CLEARING a particular bit.

If we OR a bit pattern with a mask, where a 1 is in the mask the value is set to 1, where a 0 is the bit remains the same.

OR is useful for SETTING a particular bit.

X-OR is less commonly useful.

Consider the program on the next slide:
#include <stdio.h>
void genbits(char *p,unsigned char val);

int main(int argc, char *argv[]) {
    char bits[9];
    unsigned char i=0xff;
    int j;

    for (j=0; j<8; j++) {
        genbits(bits,i);
        printf("%3d %s %02X\n",i,bits,i);
        i = i >> 1;
    }
}

/**********************************************
 ** Name:   genbits                          **
 ** Author: IRJ                              **
 ** Date:   26-SEP-2001                      **
 ** Desc:   function to produce an 8 char    **
 **         ASCII string of the binary of val**
 **********************************************/
void genbits(char *p, unsigned char val) {
    int i;
    unsigned char mask;

    mask = 0x01;          /* init mask   */
*(p+8) = '\0';        /* insert null */
for(i=0; i<8; i++)      /* foreach bit */
    {    /* if its 1    */
        if (val & mask)
            *(p+7-i) = '1';
        else
            *(p+7-i) = '0';
        mask = mask << 1; /* move mask to next
                        bit            */
    }
}
This program will print:

```
255 11111111 FF
127 01111111 7F
 63 00111111 3F
 31 00011111 1F
 15 00001111 0F
   7 00000111 07
   3 00000011 03
   1 00000001 01
```

Decimal then Binary then Hexadecimal.

Since each hexadecimal digit represents 4 bits, it is a convenient way of representing bit patterns in our programs.

**Files**

Whilst we are talking here about C on UNIX systems (e.g. Linux or Solaris), much but **not all** will be the same for other operating systems.

Files in C can be handled in two ways.

Firstly at the operating system level using (integer) file descriptors and the `open()`, `read()` and `write()` functions.

These are generally documented in section 2 of the manual since they are **system calls**.

**WE WILL NOT BE USING THESE IN THIS MODULE.**
Secondly, Files may be accessed as *streams*.

Streams are higher level representations of files.

These functions (for example *fopen, fclose, fread* and *fwrite*) are documented in section 3 of the unix manual.

File I-O at the stream level is managed by **FILE** structures.

These are managed by the C library functions, but we will need to declare pointers to a FILE structure if we wish to use files.

E.g.

```c
#include <stdio.h>    /* needed for def. of FILE */

void main(void)
{
    FILE *myfile;    /* declare a file pointer myfile */

    char *filename="fred.dat";
    /* declare a string holding our filename */

    myfile = fopen(filename,"w");
    /* open the disk file fred.dat for writing */

    if (myfile == NULL)
    {
        printf("Can't open %s for writing\n",filename);
        return;
    }
    /* check for success */

    if (fclose(myfile) != 0)
        printf("Error closing %s\n",filename);
    /* check it closed OK */
}
```
Getting data out – putchar, putc, fputc, fputs;

As well as formatted output with printf, we also have standard functions to write strings and single characters to the screen or to a file.

On a UNIX system, the streams:

stdin – standard input (generally your terminal or Xterm)

stdout – standard output (ditto)

stderr – standard error (generally the same place as stdout)

Are predefined for you.

In other words you can imagine your program has

FILE *stdin, *stdout, *stderr;

declared in in, with these opened and closed automatically.

putc('X',stdout);

&

putchar('X');

Therefore do exactly the same!

Lets have a look at the manual page (from Linux, generated as html using groff –man –Thml).
NAME

fputc, fputs, putc, putchar, puts - output of characters and strings

SYNOPSIS

#include <stdio.h>

int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);

DESCRIPTION

fputc() writes the character c, cast to an unsigned char, to stream.

fputs() writes the string s to stream, without its trailing '0'.

putc() is equivalent to fputc() except that it may be implemented as a macro which evaluates stream more than once.

putchar(c); is equivalent to putc(c,stdout).

puts() writes the string s and a trailing newline to stdout.

Calls to the functions described here can be mixed with each other and with calls to other output functions from the stdio library for the same output stream.

RETURN VALUES

fputc(), putc() and putchar() return the character written as an unsigned char cast to an int or EOF on error.

puts() and fputs() return a non-negative number on success, or EOF on error.

CONFORMING TO

ANSI C, POSIX.1

BUGS

It is not advisable to mix calls to output functions from the stdio library with low level calls to write() for the file descriptor associated with the same output stream; the results will be undefined and very probably not what you want.

SEE ALSO

write(2), fopen(3), fwrite(3), scanf(3), gets(3), fseek(3), ferror(3)
Type casting

Consider:

```c
double x;
int  y=3;
int  z=5;

x =  y/z;
```

What is the result?

**ANSWER: 0.00**  Probably not what was expected!

We need to cast our integer variables to double if we want the expected result.

A cast is simply the desired type in brackets as a prefix

```c
x = (double) y / (double) z;
```

When we cast a floating point type to an integer type, the value is truncated.

We may also need to cast pointer types.

```c
int intarr[10];
char *charp;

charp = (char *) intarr;
```

If we add 1 to a pointer we step in memory by the size of the type pointed at.

```c
char = 1 byte.  int = (implementation dependent) 4 bytes;
```

If we want to know how large an object is in bytes, we can use the `sizeof()` operator
Passing program arguments

How do we pass arguments to programs?

C has a simple but elegant approach which brings together some of the points already discussed.

main() is usually defined as:

```c
int main(int argc, char *argv[])
```

main returns an int to whatever started the program, usually the shell.

If you are using a C-Shell (csh) you can obtain the value from status.

`argc` is a counter of the number of arguments (always at least 1)

`*argv[]` is an array of pointers to char, (or if you prefer a pointer to pointer to char).

Each element is a string, the first string being the program name, the rest being the program arguments.

Consider:

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    for (i=argc; i > 0; i--)
    {
        fputs(argv[i-1],stdout);
        fputc('
',stdout);
    }
    return 3;
}
```

If we compile this program with:

```bash
cc -o foo filename.c
```
And then type:

./foo fred joe bob

The output is:

bob
joe
fred
./foo

If in our shell (I'm assuming a C-shell here – ymmv)

echo $status

would print the value 3 (main's return value)

**Error checking input**

On most systems, the default for interactive I-O is buffered.

Processing input character by character and checking it is difficult.

For files however this is important!

Consider the following program:

What are potential problems?

What happens if I enter fred?
What happens if I enter 99999999999999999999999999999999? What happens if I enter 3.14159?

How would you deal with them?
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int getint(void);

void main(void)
{
    int i;
    i = getint();
    printf("you entered %d\n",i);
}

int getint(void)
{
    char buf[80];
    int tmp, i, c;
    i=0;
    c = getchar();
    while (((i < 80) && (c != '\n')))
    {
        if (isdigit(c))
        {
            buf[i] = (char) c;
            i++;
        }
        c = getchar();
    }
    buf[i] = '\0';
    tmp = atoi(buf);
    return tmp;
}