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:

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?)
Strings - A special problem!

Pointers let us access data indirectly, but for characters strings, e.g.

"Hello"

We already do that!
We therefore needs two levels of indirection, if we want a called function to modify character array (i.e. string) values

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

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_modifier (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.

It is also worth remembering that as well as printf, two useful variants exist -

- `fprintf` - printf to a file
- `sprintf` - printf to a buffer (e.g. a character array)

**Design diversion 2**

By now, you should have had the opportunity to look at the first term assignment 😊

Are JSP like structure charts (see Royce) an appropriate design tool?
Finite State Machines

Finite State Machines come in many flavours, and are used in many software design methodologies (modern object orientated methods included!)

There are two major classes of state machines.

*Moore* state machines - outputs are a function of state alone

*Mealy* state machines - outputs depend on state and input

Hardware designers as well as software designers employ them, although usually with subtle differences.

We tend to draw Finite State Diagrams (FSD), also referred to as State Transition Diagrams (STD) to describe the design of our programs.

Different "styles" of finite state machine are often presented using subtly different representations. Different authors may present finite state machines in different ways, but essentially they are all very similar.

The basic idea is that we represent our systems using only two components: States & Transitions, Transitions however have associated with them Events (which cause them) and Actions (which are performed when they occur).

Like all design methods, for any reasonably complex problem, there are likely to be several appropriate designs.

Whilst for some problems there may be an "ideal" design, the designer chooses the design that she considers appropriate.
**States**

States are usually drawn as a circle labelled with a name and a number.

A program may only be in a single state at any given time. Technically states are discrete and mutually exclusive.

States represent periods of stability within our system. As long as we remain in a single state little changes in our programs output. Our program could continue doing something (e.g. dimming a light, displaying a welcome screen) but if our program is going to change what it is doing, this will usually involve a change of state.

We have no idea how long a state might last (consider the long clock)

States in hardware often change in nanoseconds, although both hardware and software states may last a very long time.

States are essentially periods when we are waiting for something.

**Transitions**

Transitions are when the action takes place!

A transition shows a valid path between two states and occurs when an event is detected. Events are usually sensed as input (e.g. detecting a switch has been pressed).

Transitions present us with a choice,

1. we can let our transitions occur as quickly or slowly as events,

2. we can detect and store events, and then check whether an event has occurred after a given period,
(3) we can always have a transition every given period regardless of whether an event has occurred.

Option (1) can cause implementation problems, although it is by far the easiest to implement.

Option (2) is generally the best approach for software systems since it gives us a known transition rate.

Option (3) is a more "hardwarey" approach!

**Events**

Events are "triggers" for transitions.

We can think of events as inputs in which we are interested, and which cause a transition to occur. (Events don't have to be inputs however)

Events are normally written above a line above our transition.

**Actions**

Actions are what we do when an event occurs.

More correctly actions are what we do when we enter a new states

Actions are normally written below or after a line above our transition.
A simple example

The following example is taken from:

This book is concerned with hardware design rather than software design!

The following problem is a particularly dangerous set of traffic lights to control a minor road crossing a major road. The lights have red and green lights only, the default is green for the major road. If a car is detected on the minor road, the lights change, a timer is started, and when the timer completes, the lights change back again.

We could draw this as:
Algorithmic State Machines (ASM’s)

Many hardware designers prefer algorithmic state machines

They can be clearer, less ambiguous and are closely related to the clock cycle of the controlling clock.

Unfortunately they look like flow charts 😊

The basic element is again the state, drawn as a square with the name given top left, outputs listed in the box.

Decision boxes are shown (cf. flowcharts) as triangles. Decisions are made on the basis of inputs and must follow a state.

Conditional outputs are shown as lozenges

We could draw our previous example as an ASM:
Having derived the state machine for our problem, how do we code it?

1. Inline coding. We can code a solution using goto’s in line, sequentially following our model.

This is one of the few occasions in C when the use of goto’s is probably appropriate!

Goto

Caveat Emptor!

E.W. Dijkstra famous letter to *Communications of the ACM* argued that the use of goto was considered harmful.

Kernighan & Ritchie (p65):

“Formally, the goto is never necessary, and in practice it is almost always easy to write code without it”

“Code that relies on goto statements is generally harder to understand and to maintain than code without gotos”

Having dealt with the health warnings, we need two things:

A goto statement:

    goto label;

A label:

    label: statement

Labels have the same format as variable names but finish with a colon, and are attached to the next statement. gotos and their associated labels must be in the same function.
e.g.

#define TIMEDELAY 20
#define GREEN 0xf0
#define RED 0x0f

void setlights(unsigned char major, unsigned char minor);

void main(void)
{
    long starttime;

    Major_go:
    setlights(GREEN, RED);
    car = read_sensor();
    if (car)
    {
        starttime = read_timer();
        goto Minor_go;
    }
    goto Major_go;

    Minor_go:
    setlights(RED, GREEN);
    if ((starttime + TIMEDELAY) < read_timer())
        goto Major_go;
    goto Minor_go
}

Now would be a useful point to discuss two further C statements break and continue.

break;

Causes an immediate exit from the innermost construct containing it.

break works with do, while, for or switch constructs

Whilst break is normally used in a switch, it is rarely used elsewhere!

When it is, it is essentially a goto!
continue;

Continue is much more rarely seen, continue causes the next iteration in a loop, (while, do, for). In a while or do loop continue jumps to the test, in a for loop, continue jumps to the increment.

Again, continue is essentially a goto!
2. We could implement our finite state machine using a switch:

```c
#define TIMEDELAY 20
#define MAJORGO 0
#define MINORGO 1
#define GREEN 0xf0
#define RED 0x0f

void setlights(unsigned char major,
               unsigned char minor);

void main(void)
{
    long starttime;
    int state;

    state = MAJORGO;

    while (1)
    {
        switch (state)
        {
        case MAJORGO:
            setlights(GREEN,RED);
            car = read_sensor();
            if (car)
            {
                starttime = read_timer();
                state = MINORGO;
            }
        break;
```
```c
    case MINORGO:
        setlights(RED, GREEN);
        if ((starttime + TIMEDELAY) <
            read_timer())
            state = MAJORGO;
        break;
    default:
        fprintf(stderr,
            "state machine broken!\n");
        exit(1);
    }
```

We also want to consider dispatcher based state machines, but before we do, we need a fair bit more quite advanced C.

**Structs**

*(see Royce section 3.1->3.7, p.p. 101-111)*

A Struct (short for structure) is a C record.

Often we may wish to group variables together.

Sometimes this is for convenience, at other times it may be fundamental to the logic of our program. At other times we may be using a struct as the unit for which we allocate and deallocate memory.

C lets us define and declare structures in a number of ways.

For example a point could have x and y coordinates and a colour where colour is an array of three bytes (Red,Green,Blue).

If we wished to declare a structure to hold these we could have:

```c
    struct point
    {
        unsigned char colour[3];
        int x;
        int y;
    };
```

This definition does not allocate any memory or declare any variables. It simply creates a new type *struct point*. 
If we want to declare variables we can either do so after the definition:

```c
struct point
{
    unsigned char colour[3];
    int x;
    int y;
} from, to;
```

OR:

we can type:

```c
struct point from, to, *pointp;
```

later in our program.

In essence we have created a new type “struct point”, which we can now declare variables of or functions to return.

## Using a structure

The only operations possible on a whole structure are:

- Copying
- Assigning as a unit
- Taking its address

If we want to refer to a member of our structure, we use a “.”:

```c
from.x = 3;
```

However, if we have a pointer to a structure, a sensible shorthand exists:

If

```c
pointp = &from;
```

then

```c
(*pointp).x = 3;
```
would perform the same operation as above, this could be written as:

```c
pointp->x = 3;
```

We can initialise structures, or even arrays of structures.

If we want to initialise our point structure:

```c
struct point the_point = { {255,255,0},43,-5 };```

**Typedef**

Now would be a good time to have a look at typedef!

*(Royce page 228)*

C provides a facility to define our own types. For example:

```c
typedef char *String;
```

would create a new type String. The definition of String is based on existing types.

We can also build compound types from structure declarations, e.g.

```c
typedef struct node *treenode;
typedef struct node
{  
treenode left;
    char *data;
    treenode right;
} treenode;
```

would create types treeptr and treenode for building binary trees with
After we have defined our new type we can declare variables of it.

e.g.

```c
String src, dest, argv[];
```

Often this facility can be abused, occasionally it can be used to make our code more readable!

---

**Pointers to Functions**

Pointers to functions can be headache inducing 😃

Generally, we create pointers to functions since we either want to be able to change the function we call easily (e.g. an array of functions) or we want to pass a function as a parameter (e.g. the comparison function passed to `qsort()`).

Using these is straightforward, declaring them is not!

To declare a pointer to a function we write:

```c
return_type (*func_pointer) (argument list)
```

If we have the following prototypes:

```c
int test(int (*compare)(void));
```

and

```c
int compare(void);
```

We can call `test` as

```c
x = test(compare);
```

Lets consider a few examples:
An array of functions

#include <stdio.h>

void a(void);
void b(void);

int main(void)
{
    void (*myarr[4])(void);

    myarr[0] = a;
    myarr[1] = b;
    myarr[2] = a;
    myarr[3] = b;

    for (i=0;i<4;i++)
        (*myarr[i])();

    return 0;
}

void a(void)
{
    printf("Hello ");
}

void b(void)
{
    printf("World\n");
}

This example would be much clearer if we used typedef!
An array of functions with typedef

#include <stdio.h>

typedef void (*PTF)(void);

void a(void);
void b(void);

int main(void)
{
    int i;
    PTF myarr[4];

    myarr[0] = a;
    myarr[1] = b;
    myarr[2] = a;
    myarr[3] = b;

    for (i=0; i<4; i++)
        (*myarr[i])();

    return 0;
}

void a(void)
{
    printf("Hello ");
}

void b(void)
{
    printf("World\n");
}
Qsort

Qsort or more correctly quicksort was invented by C.A.R. Hoare.

It is an extremely efficient recursive in place sort based on a divide and
conquer algorithm.

A digression on recursion follows!

For lots more information than you need see Sedgewick & Bentley's
"Quick Sort is optimal" pdf.

For most problems, most of the time qsort is probably the best sort
algorithm

Suprisingly 😊 The C library contains this sort as a function:

An extract from the man page:

#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
        int(*compar)(const void *, const void *));

DESCRIPTION
The qsort() function sorts an array with nmemb
elements of size size. The base argument points
to the start of the array.

The contents of the array are sorted in ascending
order according to a comparison function pointed
to by compar, which is called with two arguments
that point to the objects being compared.

The comparison function must return an integer
less than, equal to, or greater than zero if the
first argument is considered to be respectively
less than, equal to, or greater than the second.
If two members compare as equal, their order in
the sorted array is undefined.

An Example sort program
```c
#include <stdio.h>
#include <stdlib.h>

We know we want compare to compare unsigned ints, but we need to match the prototype

int compare( const void *x, const void *y) 
{ 
  unsigned char *a, *b;
  a = (unsigned char *) x;
  b = (unsigned char *) y;

  technically faulty, and may lead to suboptimal performance - why?

  return (((*a) > (*b)) ? 1 : -1);
}

main()
{ 
  unsigned char values[9] = 
    {47,3,2,59,9,12,1,4,99};
  int i;

  qsort(values,9,sizeof(unsigned char),compare);

  for (i=0;i<9;i++)
    printf("%d ", values[i]);
}
```
Recursion

"To iterate is human, to recurse divine"
(unknown)

Recursive functions call themselves.

The classic example is the mathematical factorial (!) function.

x! is defined as x *(x-1)! if x>1, 1 if x=0 or x=1, else undefined.

A simple recursive factorial function

```c
int factorial (int x)
{
    if (x > 1)
        return (x * factorial(x-1));
    else if (x > 0)
        return 1;
    else
        return -1;
}
```

Type casting

(Royce page 264)

We've already met typecasting many times, but never considered it formally.

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 we cast a longer integer type to a shorter integer type we take the lower portion of it.

e.g.

```c
return (unsigned char) y;
```

would return the least significant byte of the integer y

We may also need to cast pointer types. This is the most common reason to type cast.

```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

```c
<ctype.h>
```
Ctype.h is a standard C library.

It contains tests for characters (stored as int).

**Warning:**
Modern environments do not assume an ASCII world.
Setting your locale will alter the behaviour of these functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalpha(int c)</td>
<td>isupper()</td>
</tr>
<tr>
<td>isupper(int c)</td>
<td>[A-Z]</td>
</tr>
<tr>
<td>islower(int c)</td>
<td>[a-z]</td>
</tr>
<tr>
<td>isdigit(int c)</td>
<td>[0-9]</td>
</tr>
<tr>
<td>isxdigit(int c)</td>
<td>[0-9],[a-f],[A-F]</td>
</tr>
<tr>
<td>isalnum(int c)</td>
<td>isalpha()</td>
</tr>
<tr>
<td>isspace(int c)</td>
<td>[' ', '\n', '\t', '\v', '\f']</td>
</tr>
<tr>
<td>ispunct(int c)</td>
<td>isprint() &amp;&amp; !isspace() &amp;&amp; !isdigit() &amp;&amp; !isalpha()</td>
</tr>
<tr>
<td>isprint(int c)</td>
<td>!iscntrl()</td>
</tr>
<tr>
<td>isgraph(int c)</td>
<td>printing chars other than space</td>
</tr>
<tr>
<td>iscntrl(int c)</td>
<td>control characters</td>
</tr>
<tr>
<td>isascii(int c)</td>
<td>7 bit clean ASCII</td>
</tr>
</tbody>
</table>

These tests let us check character values.

Ctype also contains:

- `toupper()`
- `tolower()`
<time.h>

Yet another ANSI/ISO C standard library.

If we want to manipulate time or date functions, we need this header file.

Some functions available here are:

clock_t clock(void)

This returns the number of processor "ticks" used by the program. To convert this to "real" time, the constant CLOCKS_PER_SEC is defined.

Be careful none of the above is true to life! (although they'll do for our purposes shortly!)

Also be careful sleeping, waiting etc.

clock_t is a system dependent sized integer type

- where is it defined?
- how large is it?

Clock wrapping

Imagine that the system has:

#define clock_t char;

then:

clock_t ticks;

ticks = clock();

assigns the value 125 to ticks.

25 ticks later, what is the value?
Curses is the standard Unix C terminal I-O library.

To use curses, we not only need to have:

```
#include <curses.h>
```

in our program, we also need to link with the library, e.g.

```
gcc -Wall -lcurses -o myprog myprog.c
```

To cover much of curses here would take too long, however we need to use curses for:

Non-Blocking Keyboard Input

Imagine we want to get a key *if one is there*, but not pause our program otherwise.

Just using curses simply is easy:

- start your program with a call to initscr();
- end it *always* with a call to endwin();
- printw() replaces printf
- you need to refresh() to update the screen
- `nodelay(stdscr,TRUE)` enables non-blocking …
- … after which `getch()` is non-blocking

Curses however offers much more, see a tutorial or the man pages!
/*****
** simple traffic light (as in Zwolinski) **
** simulation **
** i/o via screen/keyboard, dispatcher based **
**
*****************************************************************************

#include <stdio.h>
#include <curses.h>
#include <ctype.h>
#include <time.h>

#define MILLISEC CLOCKS_PER_SEC/1000
#define EVENTSN 2
#define STATESN 2

#define CAR_DETECTED 0x01
#define TIMER_EXPIRED 0x02

void null(void);
void major_go(void);
void minor_go(void);
int getmask(int state_num);
int geteventnum(int events, int state);

typedef void (*PTF) ();

struct state_table
{
    int active; /* events
            permissable? */
    int next_state;
    PTF action;
};
struct state_table states[EVENTSN][STATESN] =
    {{1,1, minor_go}, {0,1,null}},         /*car */
    {{0,0, null},    {1,0,major_go}} /* time up */
};

/* Global Variables */
clock_t target_time =0L;
clock_t ticks, old_ticks;

int main(void)
{
    int ch;
    int state=0, last_state=0;
    int events=0;
    int event=0, last_event=0;

    initscr();
    cbreak();
    nodelay(stdscr,TRUE);

    old_ticks = clock();

    while (1) /* forever */
    {
        ticks = clock();
        ch = getch();
        if (ch != ERR)
            if (toupper(ch) == 'C') /* car detected */
                events = events | CAR_DETECTED;
            else if (target_time)
                if (ticks > target_time)
                    events = events | TIMER_EXPIRED;
    }
if (ticks > (old_ticks + (200 * MILLISEC))) /* state change */
{
    old_ticks = ticks;
    if (( event = geteventnum(events, state)) != ~0)
    {
        last_event = event;
        last_state = state;
        (*(states[event][state].action))();
        state =
          states[event][state].next_state;
        if (state != last_state)
            events = 0;
    }
}
endwin();
return 0;
}

int geteventnum(int events, int state)
{
    int state_events;
    int i, bitmask = 1;

    state_events = events & getmask(state);
    for (i=0; i < EVENTSN; i++)
        {
            if (state_events & bitmask)
            {
                events = events & ~bitmask;
                return i;
            }
            bitmask <<=1;
        }
    return ~0;
}
int getmask(int state_num)
{
    int mask=0, i;

    for (i=EVENTSN-1; i >= 0; i--)
    {
        mask = mask << 1;
        mask = mask | states[i][state_num].active;
    }
    return mask;
}

void null(void)
{
    return;
}

void major_go(void)
{
    printf("MAJOR: GREEN\tMINOR: RED\n");
    refresh();
    return;
}

void minor_go(void)
{
    printf("MAJOR: RED\tMINOR: GREEN\n");
    refresh();
    target_time = ticks + (2000 * MILLISEC);
    return;
}

There are some relatively serious flaws in this program:

- initial state setting
- timer wrap!