

Bits and pieces

Comments and questions to [John Rowe](#).

Today we will wrap up some useful features for single-file programs, next week we shall discuss programs split between multiple files and touch on the pre-processor.

Named constants and enumerations

Suppose we need to solve quadratic equations, and let's assume we have defined a structure to represent them.

Their roots can be one of three types: two real roots, one repeated real root or two complex roots. It might be useful for our structure to be able to store the solutions and what type they are. Obviously we can do the latter by adding a new member to the structure and setting it to '1' for one root, '2' for two real roots and '3' for two complex roots but then we need to remember what '1', '2' and '3' mean. It's much better to give a name to these constants.

C provides two ways of naming constants. We've met one already, `#define`, but C provides another way specifically designed for our situation, **enumerations**:

```
enum eqnstatus { EQN_UNSOLVED, EQN_ONEROOT, EQN_REALROOTS,
EQN_COMPLEX_ROOTS };
```

Enumerated constants are integers

Enumerated constants are integers and by default the first enumerated constant has the value zero.

Now anywhere in our program we can use the named constants `EQN_UNSOLVED`, `EQN_REALROOTS`, `EQN_COMPLEX_ROOTS` to mean zero, one, two or three respectively, but with user-friendly names:

```
enum eqnstatus { EQN_UNSOLVED, EQN_ONEROOT, EQN_REALROOTS,
EQN_COMPLEX_ROOTS };

main() {
    enum eqnstatus eqn_status = EQN_UNSOLVED;
}
```

Printing enumerations out with `%d` just prints their integer values but debuggers usually understand them and will show an enumerated value as, for example, "EQN_ONEROOT" rather than "1" which makes debugging easier. This is why they are better for this purpose than `#define`.

Enumerations can be combined with `typedefs` as in this example :

```
#include <stdio.h>

typedef enum { VANILLA, CHOCOLATE, STRAWBERRY } Flavour;

typedef struct {
    Flavour flavour;
    float fat;
    float sugar;
    float calories;
} Icecream;

main() {
    Icecream icecream;

    icecream.flavour = CHOCOLATE;
    printf("%d\n", icecream.flavour); /* Prints: 1 */
}
```

An occasionally-useful trick

We may want to have an array of Icecream structures, one for each flavour. To help with this we can use the useful trick of adding a dummy enumeration value at the end of the list:

```
typedef enum { VANILLA, CHOCOLATE, STRAWBERRY, NUM_FLAVOURS } Flavour;

typedef struct {
    Flavour flavour;
    float fat;
    float sugar;
    float calories;
} Icecream;

Icecream icecreams[NUM_FLAVOURS];
```

Now NUM_FLAVOURS has the value "3" and as we add new flavours then as long as we are careful to keep NUM_FLAVOURS as the last in the list its value will always be correct:

```
typedef enum { VANILLA, CHOCOLATE, STRAWBERRY, TOFFEE, NUM_FLAVOURS } Flavour;
```

Non-default values

It's possible to write:

```
enum something { TYPE1, TYPE2=76, TYPE3, TYPE4 };
```

In which case TYPE1 has the value 0, TYPE2 has the value 76 and TYPE3 has the value 77, etc. but it's very unusual to.

When to use enumerations

Enumerations are used for one thing only: when a variable is used to "enumerate" different, mutually-exclusive possibilities. For general named constants use #define.

Recursion

To iterate is human, to recurse divine.

Computer Science saying

God is subtle but he is not malicious.

Einstein

Functions that call themselves

Recursion is an extremely simple concept: a function simply calls itself. Of course, a function can't always call itself, that would create an infinite loop, so all recursive functions have some sort of **test** before they call themselves.

```
void myfun(some_args) {
    /* Do some stuff */
    if ( some_test )
        myfun(some_other_args);
}
```

A variation on the above is to do the test inside the function:

```
void myfun(some_args) {
    if ( I_didnt_need_to_be_called )
        return;

    /* Do some stuff */
    myfun(some_other_args);
}
```

Where it works this is neater (as we do the test only once).

When functions call themselves each instance has its own local variables, unless they are static.

Example: flood fill

A common situation, with a number of variations, is when we have a rectangular grid of integer values, typically representing an image, and we wish to identify "islands", i.e. sets of neighbouring pixels with the same value.

In the following example we wish to identify and number "islands" in an empty "sea".

The initial description

```
#include<stdio.h>

/*
 * Demonstrate a flood fill, using it to count and label non-empty
 * contiguous regions in an image.
 *
 * An image point that is empty has the value
 * zero, if it is non-empty but not yet assigned to a region
 * it has the value minus one. Pixels that have been assigned
 * to regions have the values one, two, three, etc. according
 * to the region they belong to.
 */

#define NX 10000
#define NY 10000
```

The flood fill function

```
/*
 * Simple flood filler: set all contiguous spaces with value "-1"
 * to the value "area_id".
 */
void floodfill(int image[NX][NY], int x, int y, int area_id) {
    int ix, iy;

    if (x < 0 || x >= NX || y < 0 || y >= NY || image[x][y] != -1)
        return;

    image[x][y] = area_id;

    for (ix = x-1; ix <= x+1; ++ix)
        for (iy = y-1; iy <= y+1; ++iy)
            floodfill(image, ix, iy, area_id);
}
```

Notes

1. The first thing we do is to see if the function needs to do anything, if not we immediately return.
2. We check the co-ordinates are in range before checking the pixel (image) value at that point.
3. Then the **first** thing we do is to mark the pixel as filled.
4. Then we try each of its neighbours. (We also try the original point but that doesn't matter as it immediately returns.)

Had we reversed the last two steps then the function would have been forever calling itself. It would have done this even if we had put in an `if ()` statement to avoid refilling itself:

```
for (ix = x-1; ix <= x+1; ++ix)
    for (iy = y-1; iy <= y+1; ++iy)
        if ( ix != x || iy != y)
            floodfill(image, ix, iy, area_id);

image[x][y] = area_id; // Too late!!
```

The outer function

All we need to do now is to go over each point in the image and call `.tt.floodfill()` for each point that needs it:

```
/*
 * Count and label contiguous areas in an image.
 * On input: 0 means empty space, -1 means non-empty
 *
 * On exit the contiguous areas have the values 1, 2, 3 etc
 * and the number of areas is returned.
 */
int count_and_label(int image[NX][NY]) {
    int x, y, area_id = 0;

    for (x = 0; x < NX; ++x)
        for (y = 0; y < NY; ++y)
            if (image[x][y] == -1)
                floodfill(image, x, y, ++area_id);
}
```

```
    return area_id;
}
```

This will fill the first island/region with the value "1", the second with the value "2", etc. and return the total number of regions found.

When recursion goes wrong

The biggest problem is the internal test or tests: either the recursion never stops or some possibilities get missed out. (We saw an example of this above when we erroneously marked a point as filled **after** we flood-filled its neighbours.)

Consider the following rather over-the-top way of calculating a factorial:

```
int factorial(int i) {
    if ( 1 > 0 ) // Bug!
        return i * factorial(i - 1);
    return 1;
}
```

You can see what it's meant to do but we've written '1' instead of 'i' so it will go on for ever. The first stage of debugging is just to print out its arguments:

```
int factorial(int i) {
    fprintf(stderr, "i: %d\n", i);
    if ( 1 > 0 ) // Bug!
        return i * factorial(i - 1);
    return 1;
}
```

We can get even more details by adding an extra debugging argument to tell us the depth:

```
int factorial(int i, int depth) {
    int d;

    for(d = 0; d <= depth; ++d)
        fprintf(stderr, " ");
    fprintf(stderr, "i: %d (depth %d)\n", i, depth);
    if ( depth > 12) {
        fprintf(stderr, "Too deep!\n");
        abort();
    }

    if ( 1 > 0 ) // Bug!
        return i * factorial(i - 1);
    return 1;
}
```

Here we have done several things:

- We have added an extra 'depth' argument which we increment by one every time.
- We have indented the output according to the depth. This isn't necessary but it is nice.
- Finally, and rather extremely, we have called `abort()` if `factorial()` goes too deep. This is only useful when we are running under the debugger as we can look at the "layers" of recursive call.

Be prepared to add some debugging output to your recursive routine.

Using a wrapper

It may be very inconvenient to add an extra argument to your routine because it is called elsewhere, there are header files, etc. In that case just create a simple wrapper:

```
int factorial(int i) {
    return factorial_debug(i, 0);
}
```

Bitwise operators

Bitwise operators operate on the individual bits of a variable, rather than the arithmetic value of the whole variable as the arithmetic operators do.

The operators

| Operator | Description |
|----------|--|
| & | Bitwise and (both bits are one) |
| | Bitwise or (either or both bits are one) |
| ^ | Bitwise exclusive or (just one bit is one) |
| << | Left shift (multiplies by power of 2) |
| >> | Right shift (divides by power of 2) |
| ~ | One's complement |

NB the One's complement operator takes a single argument, the others two.

Examples

We take the example of an unsigned char (one byte, ie 8 bits). Curiously there is no way to write numbers as binary in C; the nearest is hexadecimal, but for clarity we shall show the calculation in binary.

| Expression | Calculation |
|---------------------|---|
| 00110011 & 11110000 | 00110011 11110000 ----- 00110000 |
| 00110011 11110000 | 00110011 11110000 ----- 11110011 |
| 00110011 ^ 11110000 | 00110011 11110000 ----- 11000011 |
| 10110011 << 2 | 10110011 ----- 11001100 |
| 11110000 >> 3 | 11110000 ----- 00011110 |
| ~11110000 | 11110000 ----- 00001111 |

Binary flags

The most common use of binary operators is to define **flags**, ie options that are either on or off. Typically there's a header file with various constants each of which is an exact power of 2 (ie has only one bit set):

```
#define OPTION1 0x01 /* 1 binary 00000001 */
#define OPTION2 0x02 /* 2 binary 00000010 */
#define OPTION3 0x04 /* 4 binary 00000100 */
#define OPTION4 0x08 /* 8 binary 00001000 */
#define OPTION5 0x10 /* 16 binary 00010000 */
#define OPTION6 0x20 /* 32 binary 00100000 */
```

The code sets, unsets and tests options as follows:

```
unsigned int flags;

int main(void) {

    flags |= OPTION3; /* Set OPTION3 */
    flags &= ~OPTION4; /* Unset OPTION4 */

    if ( flags & OPTION3)
        printf("Option 3 is set\n");

}
```

The setting and testing of flags is fairly clear, the unsetting of `OPTION4` is a little more complicated: `OPTION4` like all options, has just one bit set so

The setting and testing of flags is fairly clear, the unsetting of `OPTION4` is a little more complicated. `OPTION4`, like all options, has just one bit set so `~OPTION4` has every bit **except** that one set. So `flags &= ~OPTION4` has the following effect:

- The bit that was one in `OPTION4` is zero `~OPTION4` so the binary and for that bit must always be zero.
- The bit that was zero in `OPTION4` is one `~OPTION4` so the binary and for that bit will be one if and only if the corresponding bit in `flags` was one. That is to say, it is not changed.

Example in binary

```
00111011 & ~00001000
~00001000 = 11110111
```

```
00111011
11110111
-----
00110011
```

Pseudo-random numbers

The function `rand()` produces a pseudo-random number, for example:

```
int diceroll;

diceroll = 1 + (rand() % 6);
```

By default it always produces the same pseudo-random sequence each time the program is run. The traditional way round this is:

```
#include<time.h>
#include<stdlib.h>

int main() {
    int i;

    srand(time(NULL)); // Call this just once per program!

    for(i = 0; i < 20; ++i)
        printf("You have rolled %d\n", 1 + (rand() % 6));

    return 0;
}
```