Coin Change

Remember the task:
We want to write a program that
▶ ask the user for an amount of money
▶ calculates the coins needed for this amount
▶ outputs the number of each coin
Recall that solution was very ugly – different constants for each coin type, multi-branch conditionals, and so on. Moreover, the coin values were hard-wired – suppose we wanted US coins!
This was because we didn’t know about arrays.
So here is Coin Change done as we would now do it:

Type of Coins

Coins range from 1p to £2

/* array of coin values in decreasing order */
const int coinValues[] = { 200, 100, 50, 20, 10, 5, 2, 1 };

/* number of different types of coin –
   using a sneaky way to avoid counting them */
const int NUM_VALUES = sizeof(coinValues)/sizeof(int);

/* names for the coins */
const char *coinNames[] = { "two pound", "one pound",
   "50p", "20p", "10p", "5p", "2p", "1p" };
Missing out the error handling (do it as before):

```c
int main(void) {
    int amount;
    int coinNums[NUM_VALUES];

    ReadInput(&amount);
    CalculateCoins(amount, NUM_VALUES, coinValues, coinNums);
    PrintAmount(amount, NUM_VALUES, coinNames, coinNums);

    return EXIT_SUCCESS;
}
```

**Output to User**

```c
int PrintAmount(int amount, int len, const char *cNames[],
                const int cNums[]){
    printf("%dp may be returned using the following ", amount);
    int i;
    for (i=0; i<len; i++){
        printf("%d %s coins\n", cNums[i], cNames[i]);
    }
    return EXIT_SUCCESS;
}
```

**Exercises**

1. It's rather ugly that we have separate arrays for coin values and names – suppose we get them out of sync!
   Define a type `struct coin {
   int value;
   char *name;
   ` and rewrite the program that way.

2. Handle the punctuation between lines of output, and the use of plurals ('coin'/'coins') correctly. (This is tedious!)
### Booleans

&& (“and”):
- **usage** is \( d \&\& s \), for \( d, s \) booleans.
- **meaning** is like ‘and’ in English, eg, “it is dry and it is sunny”.

|| (“or”):
- **usage** is \( t \|\| s \), for \( t, s \) booleans.
- **meaning** is like ‘or’ in English, eg “Tesco or Scotmid will be open”.
- **NOT exclusive or**: \( t \|\| s \) also holds if both \( t \) and \( s \) hold.

! (“not”):
- \( !p \) is true if and only \( p \) is false.

### Examples

```c
char c='F';
const int false=0; true=1;

(1 < 9) || (2 == 5)
IsSunny(today) || true
('A' <= c) && (c <= 'Z')
false && (1 == 1)
```

### Boolean as int

- Booleans are represented as integers in C.
- \( 1 \) is the value of a true expression: \( (x == x) \) is 1
- \( 0 \) is the value of a false expression: \( x < x \) is 0
- Non-zero values are treated as true:
  ```c
  while(45){ };
  /* loop forever */
  ```

### Truth Table

| expr1 | expr2 | !expr1 | expr1 && expr2 | expr1 || expr2 |
|-------|-------|--------|---------------|------------|
| false | false | true   | false         | false      |
| false | true  | true   | false         | true       |
| true  | false | false  | false         | true       |
| true  | true  | false  | true          | true       |
Truth Table (as int)

| expr1 | expr2 | !expr1 | expr1 && expr2 | expr1 || expr2 |
|-------|-------|--------|---------------|------------|
| 0     | 0     | 1      | 0             | 0          |
| 0     | non-zero | 1    | 0             | 1          |
| non-zero | 0     | 0      | 0             | 1          |
| non-zero | non-zero | 0    | 1             | 1          |

“short-circuit” to testing

&& and || expressions are evaluated in order:
- eg, first && second
- Arithmetic expressions DO NOT have this property
For Boolean expressions, evaluation ends as soon as the outcome is known:
- eg false && never
- eg (x == x) || never

Testing elements of an array

int CheckRange(int max, int *array, int length) {
    int i = 0;
    while (i < length) {
        if (array[i] > max)
            break;
        i++;
    }
    if (i < length) /* We broke out of the loop early */
        return 0;
    else return 1;
}

Testing elements ... “short-circuit” version

int CheckRange2(int max, int *array, int length) {
    int i = 0;
    while (i < length) && (array[i] <= max)) {
        i++;
    }
    if (i < length) /* We broke out of the loop early */
        return 0;
    else return 1;
}
Watch out!

Don’t assume that arithmetic expressions will evaluate in order. For example:

```plaintext
x = 10;
y = ++x + x;
```

In practice, depending on compiler, this could evaluate as either of the following:

```plaintext
y = 11 + 11;  /* ++x; y = x + x; */
y = 11 + 10;   /* y = x; ++x; y += x; */
```

Avoid writing code with these ambiguous interpretations.

---

The common mathematical short-hand $3 < j < 6$ is evaluated as $(3 < j) < 6$

Suppose $j$ is 7. Then the sequence of evaluations is:

```plaintext
(3 < 7) < 5
= 1 < 5  /* 1 is the result (true) of 3 < 7 */
= 1     /* representing true */
```

Must be clear and write $(3 < j) \&\& (j < 6)$

---

**Precedence – highest to lowest**

<table>
<thead>
<tr>
<th>()</th>
<th>[]</th>
<th>++</th>
<th>--</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>/</td>
<td>%</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;=</td>
<td>&gt;</td>
<td>&gt;=</td>
</tr>
<tr>
<td>==</td>
<td>!=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>|</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Left to right ordering within same precedence level.

Precedence determines bracketing of expression.

Precedence **does not** determine order of evaluation.