REFACTORING CONDITIONAL STATEMENTS

Code quality

We consider code quality to be an aspect of software quality that can be determined by just looking at the source code, i.e., without any form of testing, or checking against specifications. *Good code is simple, easy to understand and well documented.*

For example, consider the function "Verdict" which takes one input parameter (a boolean called "doubt"), and which then prints either "guilty" or "not guilty" depending on the value of "doubt".

Here is one possible implementation:

```
void Verdict(int doubt)
{
      int printNot;

      if (doubt == 1) {
            printNot = 1;
      } else {
            printNot = 0;
      }

      if (printNot == 1) {
            printf("not ");
            printf("guilty");
      }
      if (printNot == 0) {
            printf("guilty");

      }
}
```

And here is an equivalent implementation – it functions ***exactly the same*** but is simpler to follow. Both versions of the function do the ***same*** thing, but this version uses *better style*:

```
void Verdict(int doubt)
{
      if (doubt) {
            printf("not ");
      }
      printf("guilty");
}
```

Why we should care about code quality?

Code quality is very important in industry: simple code is easier to read and understand and facilitates code enhancement (adding new functionality to an existing application) and code maintenance (upgrading, porting etc).

Refactoring describes the process of restructuring or reorganising code - changing the way it *looks* - without modifying the way it *behaves*. Refactoring is used to simplify and improve the design and structure of code.

Refactoring: changing the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.

Refactoring is a bit like proofreading an essay – once the content is there, you revise and reorder or simplify sentences to make it more readable.

In large software projects, the focus of refactoring is on the way methods and classes are structured. In small programs, like the ones we are writing at this stage, there are often opportunities for improving the structure of short fragments of code (for example, within functions).

FOUR RULES

Below are four rules that we can apply to refactor conditional statements:

Rule 1 – Use Booleans directly

Original code	Refactored code

`if (a == 1) {`
`        printf("it is true");`
`}`
	`if (a) {`
	`        printf("it is true");`
	`}`

`if (a == 1) {`
`        b = 0;`
`} else {`
`        b = 1;`
`}`
	`b = !a;`

`if (a > 10 || b < 50) {`
`        return 1;`
`} else {`
`        return 0;`
`}`
	`return (a > 10 || b < 50);`

Rule 2 – Collapse if statements

Original code	Refactored code

`if (a > b) {`
`        if (c > d) {`
`                printf("it is true");`
`        }`
`}`
	`if ((a > b) && (c > d)) {`
	`        printf("it is true");`
	`}`

`if (value < 2) {`
`        return 0;`
`}`
`if (value > 15) {`
`        return 0;`
`}`
	`if ((value < 2) || (value > 15)) {`
	`        return 0;`
	`}`

Rule 3 – Remove redundant conditions

Original code	Refactored code

`if (x != 10) {`
`        printf("x is not 10");`
`} else if (x == 10) {`
`        printf("x is 10");`
`}`
	`if (x != 10) {`
	`        printf("x is not 10");`
	`} else {`
	`        printf("x is 10");`
	`}`

`if (x % 2 == 0) {`
`        printf("is even");`
`}`
`if (x % 2 == 1) {`
`        printf("is odd");`
`}`
	`if (x % 2 == 0) {`
	`        printf("is even");`
	`} else {`
	`        printf("is odd");`
	`}`

<u>Rule 4</u> – Look for and remove redundant statements
Reducing repetition in code is generally a good idea. When a statement appears more than once, you should ask yourself whether the repetition can be removed by reorganising the code.

Original code	Refactored code

`if (sold > DISCOUNT_AMOUNT) {`
`      total = sold * DISCOUNT_PRICE`
`      printf("Total = %f", total);`
`} else {`
`      total = sold * PRICE`
`      printf("Total = %f", total);`
`}`
	`if (sold > DISCOUNT_AMOUNT) {`
	`      total = sold * DISCOUNT_PRICE`
	`} else {`
	`      total = sold * PRICE`
	`}`
	`printf("Total = %f", total);`

`if (value > 10) {`
`      p = p + ((c + 10)*TAX_RATE) / 100;`
`} else {`
`      p = p - ((c + 10)*TAX_RATE) / 100;`
`}`
	`double adj = ((c + 10)*TAX_RATE) / 100;`
	`if (value > 10) {`
	`      p = p + adj;`
	`} else {`
	`      p = p - adj;`
	`}`

<u>REFACTORING PRACTICE TASKS</u>

<u>Refactoring Practice A</u>

Consider the following function definition which is provided to you:

```
int Divisible(int rem)
{
    int divisible;
    if (rem == 0) {
        divisible = 1;
    } else {
        divisible = 0;
    }
    return divisible;
}
```

Can you *modify the definition* of this function (i.e. the lines of code inside the curly braces) to make it simpler? Hint: it is possible to rewrite this function using just a single line of code (inside the curly braces), so that it behaves exactly the same as defined above.

<u>Refactoring Practice B</u>

Consider the following function definition which is provided to you:

```
void Twice(int x, int twice)
{
    if (x < 100) {
        if (twice == 1) {
            x = x * 2;
        }
    }
    printf("%d", x);
}
```

Can you *modify the definition* of this function (i.e. the lines of code inside the curly braces) to make it simpler? Hint: in this case, try to remove the nested if statement.

Refactoring Practice C

Consider the following function definition that solves a mathematical puzzle. On a farm there are a number of chicken and cows. You can assume all chickens have 2 legs and all cows have 4 legs! Given only the total number of heads and the total number of legs across all animals, the function below returns **the number of chickens** on the farm (or returns -1 if the input is not valid):

```
int NumChickens(int heads, int legs) {
    int odd = legs % 2 != 0;
    if (odd == 1) {
        // Invalid input
        return -1;
      }
    if (legs < 2*heads) {
        // Invalid input
        return -1;
      }
    if (legs > 4*heads) {
        // Invalid input
        return -1;
      }
    if (2*heads == legs) {
        // All chickens
        return heads;
      }
    if (legs > 2*heads) {
        int cows = (legs - 2*heads) / 2;
        return  heads - cows;
      }
      return -1;  // ensure not all returns are inside conditions
}
```

There are several ways that this code could be refactored so that it is simpler. Modify the code so that the function behaves exactly the same way, but without some of the unnecessary complexity.

EXAMPLE SOLUTIONS

Refactoring Practice A Solution

```
int Divisible(int rem)
{
       return !rem;
}
```

Refactoring Practice B Solution

```
void Twice(int x, int twice)
{
       if (x < 100 && twice == 1) {
              x = x * 2;
       }
       printf("%d", x);
}
```

Refactoring Practice C Solution

```
int NumChickens(int heads, int legs) {
       if (legs % 2 != 0 || legs < 2*heads || legs > 4*heads) {
              // Invalid input
              return -1;
       } else {
              int cows = (legs - 2*heads) / 2;
              return heads - cows;
       }
}
```