Topic 3 – Selection

Decisions, Decisions

- Remember that algorithms can only do three things:
    - Sequence
    - *Selection*
    - Iteration

- We first introduced the concept of selection in Topic 1 and said that it was important because it allows algorithms to make choices and perform differently under different inputs.

- We're now going to talk about selection in more detail and show how to use it in writing algorithms and programs.

BOOLEAN LOGIC

Boolean Data

- You might remember from Topic 1 that selection-based expressions that make up an algorithm "decide" what to do based upon the truth or falsity of some expression.

- Most of the expressions you will have seen so far involve computation, often arithmetic.
  - For example: a = b * 5

- If we assume that *a* and *b* are both integers then this expression will have an integer result.
- As we've seen in the previous topic, the *type* of a piece of data is very important when writing a program (although it's also important to consider when developing algorithms).

- So what type is the result we get from an expression that relates to truth or falsity?

- In computer science we call data with a true or false result *boolean* after mathematician George Boole.

- While integers can store any whole number value (within extensive limits), boolean data can only store the value TRUE or the value FALSE.

Boolean Data in Programming Languages

- Different languages deal with boolean data slightly differently.
- Some languages, like Java, actually have a specific data type for boolean data.
- Others like C just treat boolean data as an integer (generally an `int`).
- In C if the integer expression has the value zero then this means *false*, otherwise it means *true*.
- We'll talk about this more a little later and give some examples.

Relational Operators

- Just as there are arithmetic operators that give numeric results, there are also operators that give boolean results.
- These are very important in writing selection logic for algorithms since they define how the selection statement makes its decision.
- Relational operators work on different types of data, primarily numeric data.
- So although relational operators don't work with boolean data, they produce boolean results, i.e., either true or false.
- For the table below assume that *a* and *b* are integers but they can be virtually any simple data type.

Operator	***Example***	***Description***
`<`	`a < b`	Is *a* less than *b*?
`<=`	`a <= b`	Is *a* less than or equal to *b*?
`>`	`a > b`	Is *a* greater than *b*?
`>=`	`a >= b`	Is *a* greater than or equal to *b*?
`==`	`a == b`	Is *a* equal to *b*?
`!=`	`a != b`	Is *a* not equal to *b*?

Notice how although the two operands *a* and *b* are integers, the relational operators cause the expression to have a boolean (true/false) result.

Boolean Operators

- Although relational operators are used to produce boolean results that can be used in selection statements, they don't actually operate on boolean data.

- However, there are several operators that can be applied to these boolean results to form very sophisticated boolean expressions.

- For many simple expressions these are not needed but sometimes they are essential for constructing the appropriate logic.

- Just as an arithmetic operator (such as +) can be applied to two items of numeric data to produce a numeric result, these boolean operators can be applied to two items of boolean data to produce a result.

- There are four boolean operators and these are:
    - AND
    - OR
    - XOR
    - NOT

We will now consider each of these.

AND

- Firstly it is important to remember that:
  - Boolean data can only be true or false.
  - Therefore a boolean expression can only have a true or false result.

- The boolean AND operation produces a true result only if both its boolean operands are true.

- You can read this as:

  x AND *y* gives true, if and only if *x* is true **and** *y* is true.

- We can use a device called a *truth table* to show the behaviour of the AND operation:

a	***b***	***a AND b***
0	0	0
0	1	0
1	1	1
1	0	0

- When reading this table, remember that zero means false and one means true.

- You should not memorise this table but you should be able to understand it such that you could re-generate it for yourself.

OR

- The boolean operator OR produces a true result if either of its operands is true.

- You can read this as:

  x OR *y* gives true if either *x* is true or *y* is true or both are true.

- Here is the truth table for OR:

a	***b***	***a OR b***
0	0	0
0	1	1
1	1	1
1	0	1

XOR

- XOR is slightly trickier and is not used in logical boolean expressions but it is covered here for completeness.

- XOR gives a true result when one and only one of its operands is true.

- You can think of this as:

x XOR *y* gives true if either *x* is true or *y* is true but not both are true.

- Here is a truth table for XOR:

a	***b***	***a XOR b***
0	0	0
0	1	1
1	1	0
1	0	1

- XOR means “exclusive OR” because only one of its operands can be true but not both.

NOT

- Unlike the previous boolean operators which have two operands, NOT only has one.

- NOT is the *complementary* operator in that it will give the opposite of whatever its operand is.

- If the operand is true then the result will be false and if the operand is false then the result will be true.

- Here is a truth table for NOT:

x	***NOT x***
0	1
1	0

- In other words, NOT simply reverses the value of whatever boolean expression is given.

Boolean Operators in Programming

- So far we have been using the English words “AND” and “OR” etc. to represent the boolean operators.
- However, most programming languages (including C) use a shorthand notation for these.

- The following table describes this shorthand:

Operator	***Example***	***Means***
`&&`	`a && b`	*a* AND *b*
`||`	`a || b`	*a* OR *b*
`!`	`!a`	NOT *a*

In your algorithms you can use the words, e.g., AND.

Note that there is no boolean XOR operator in most languages, including C. There is a bitwise XOR operator but this does something quite different.

Using Boolean Logic

- Now we have a set of relational operators that produce boolean expressions and logical boolean operators to process these results.
- So how is this useful?

- Essentially we can combine these to produce expressions for any particular selection logic we require.

- For example, we have a value called *score* and we need to find out if that score is between 80 and 100.
- Using the relational and boolean operators we can produce an expression which will be true if this is the case:
- The two expressions below are the same except the second one uses C syntax.

score >= 80 AND score <=100

```
(score >= 80) && (score <= 100)
```

- Another example might be if we want to check whether a user wishes to proceed with some operation in a program.
- If the user enters 'y' or 'Y' then we want to proceed so we can write the following boolean expression:

response == 'y' OR response == 'Y'

```
(response == 'y') || (response == 'Y')
```

- If this expression is true then the user wants to proceed.

- However, being able to produce these expressions is no use if we can't do something as a result.
- So we will now look at some selection constructs that exist within the C language.

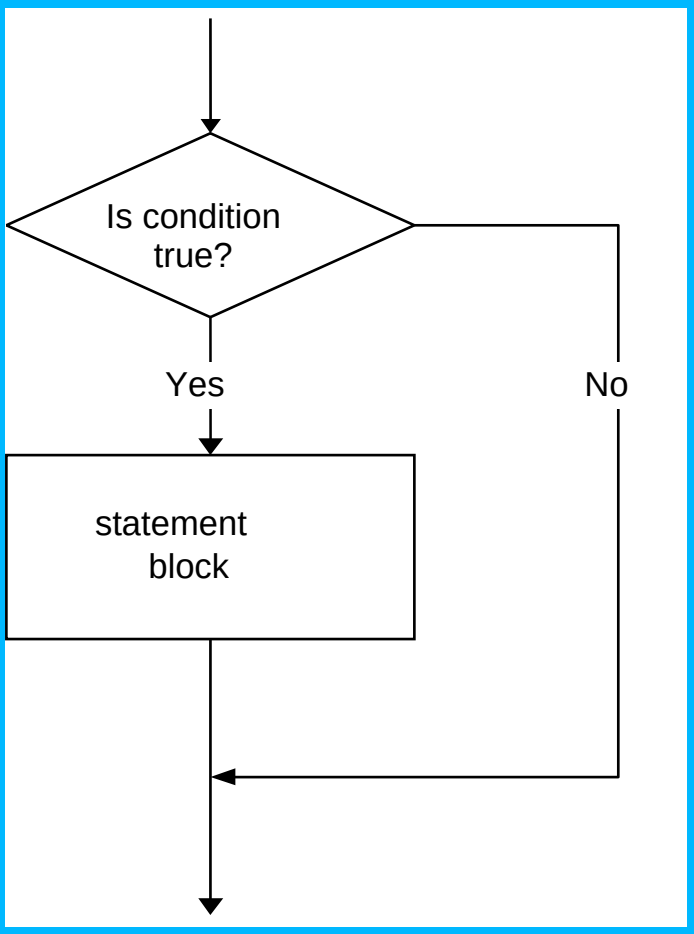
IF STATEMENTS

A Basic If *Statement*

- The principal selection construct in the C language is the *if* statement.

- The basic version of the *if* statement allows the program to execute a particular block of code if some boolean expression is true.
- Once this block of code has finished executing then the program continues on with the rest of the code.
- If the expression is not true then the conditional block is skipped and execution continues as above.

- The flow chart opposite illustrates this behaviour:



The general syntax in C for a basic *if* statement is

```
if(condition)
{
    ...
    ...
}

/* Execution
   continues here
*/
```

Basic If *Example*

- Remember the "menu" algorithm example from Topic 1:

```
print "you have 3 choices"
print "enter a for option 1"
print "enter b for option 2"
print "enter c for option 3"

response <-- keyboard input

if response == 'a'
      print "You have selected option 1"
if response == 'b'
      print "You have selected option 2"
if response == 'c'
      print "You have selected option 3"
```

The part with the red bracket around it is obviously the pseudocode that will be implemented with an *if* statement.

- Assuming the existence of a variable called `response` of type `char` which holds the user's menu selection, the following is the C code to implement this part:

```
if(response == 'a')
{
   printf("Your have selected option 1");
}

if(response == 'b')
{
   printf("You have selected option 2");
}

if(response == 'c')
{
   printf("You have selected option 3");
}
```

There are some important things to note about this code:

- Firstly the use of the == relational operator.
  - This is a *test for equality*, that is it tests to see whether the two operands are equal in value.
  - So each of the *if* statements tests to see whether `response` is equal to the appropriate letter.
  - A mistake almost every beginner programmer makes is to use the assignment operator = rather than the test for equality relational operator ==.
  - This has the effect of assigning the value on the right to the variable on the left and then treating the whole lot as a boolean expression (zero = false, non-zero = true in C).
  - **This leads to very strange and hard to diagnose problems in your code!**

- Secondly note the use of single quotes are for the letters like `'a'`.
  - This tells the C compiler that these are of type `char` and is important because this is the same type as response.
  - **You can only compare variables of the same type.**

- Thirdly notice that each *if* statement is independent from the others.
  - If the first one matches then the second (and third) *if* statements will still be executed, even though their conditions cannot possibly be true.
  - Clearly this is less efficient than it could be.

- Finally note in this case the curly brackets aren't used after the *if* statement to indicate the code to be executed.
  - You only need to use these brackets when there is more than one line to be executed as part of the *if* statement.
  - However, you can still put them in anyway if you like.

Complete C Program

```
#include <stdio.h>

int main()
{
   char response;

   /* Print menu giving choices */
   printf("You have three choices.\n");
   printf("Enter a for option 1\n");
   printf("Enter b for option 2\n");
   printf("Enter c for option 3\n");

   /* Read in user's response */
   scanf("%c%*c", &response);

   /* Perform appropriate response */
   if(response == 'a')
   {
    printf("Your have selected option 1\n");
   }

   if(response == 'b')
   {
    printf("You have selected option 2\n");
   }

   if(response == 'c')
   {
    printf("You have selected option 3\n");
   }

   return(0);
}
```

Problems with this Program

There are a number of problems and limitations with this code.

- Firstly the program assumes that the user will input the letter in lower case.
  - While the program implicitly requests a lower case letter, the user may accidentally input a capital.
  - This is something the program *should* be able to check for to be considered reasonably robust.

- Secondly, as noted above, the code is quite inefficient since each of the conditions is checked independently.
  - This is despite the fact they are all mutually exclusive and, if one is found to match, the others should be ignored.
  - The code is therefore both inelegant and inefficient.

- Thirdly notice that there is no default condition.
  - If the value of `response` does not match any of the three tested then the program will just continue through and do nothing.
  - If the user fails to select a valid option then the program should detect this and output a suitable error message.

- Finally, the program just exits after performing the specified task.
  - Although obviously just a demo program that doesn't do anything useful, in a real menu program you would probably keep displaying the menu until the user told it to exit.

We will now fix each of these problems (except for the last which will have to wait for a later topic!)

Using the Boolean Operators

- We already know how to solve the first problem:
  - A previous example gave us the answer!

- Using the boolean operators (AND, OR, NOT, etc.) we can construct a boolean expression to cater for both capital and lower case inputs using the expression:

  response == 'a' OR response == 'A'

So now the code becomes:

```
if((response == 'a') || (response == 'A'))
   printf("Your have selected option 1");

if((response == 'b') || (response == 'B'))
   printf("You have selected option 2");

if((response == 'c') || (response == 'C'))
   printf("You have selected option 3");
```

Note the use of an extra pair of brackets to separate each boolean expression that we are combining.

- We *could* also use the boolean operator NOT to construct a very complex expression to test for the default condition:

```
if(!((response == 'a') || (response == 'A') ||
(response == 'b') || (response == 'B') || (response
== 'c') || (response == 'C')))
   printf("Not a valid option!\n");
```

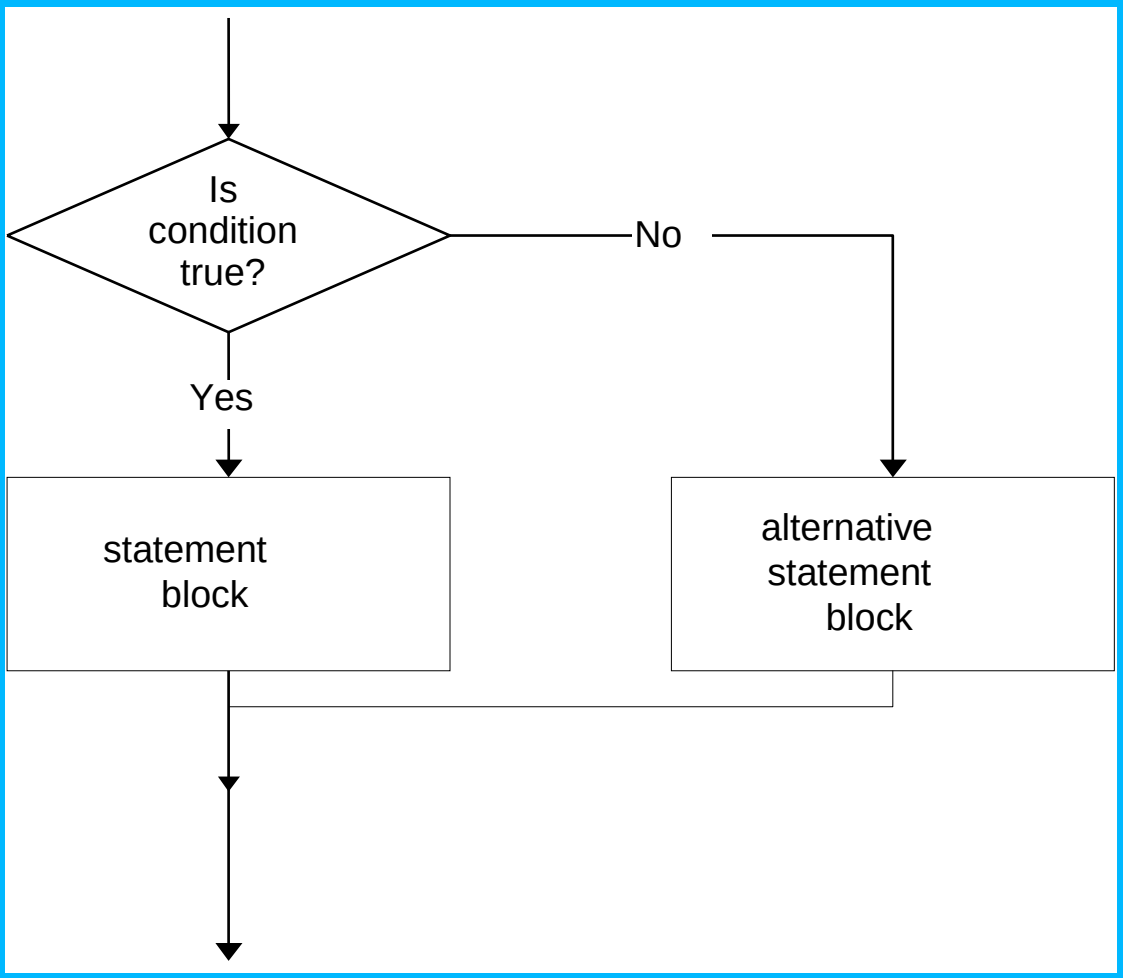
- But this is complex, ugly and lots of typing – there is a much better way...

THE *IF-ELSE* STATEMENT

- The basic *if* statement is clearly very useful by allowing us to write programs that can behave differently in different situations.
- However, already we have discovered serious limitations with it.

- A slightly more advanced approach is the *if-else* statement.

- Unlike the basic *if*, *if-else* allows us to not just do something if a condition is true but also to do something else if it is not true.



The basic structure of an *if-else* statement looks like:

```
if(condition)
{
   /*   This code executed if the condition
        is true and is called the "true clause"
   */
}
else
{
     /* This code executed if the condition
     is false and is called the "false clause"
     */
}
/* Execution continues here */
```

- Again note that the curly brackets for each block of the if-else statement are only needed when more than one line of code is contained in that block:

```
if((selection == 'y') || (selection == 'Y'))
{
   printf("You have chosen the affirmative!\n");
}
else
{
   printf("You have not chosen the affirmative.\n");
}

printf("But you still end up at the same place,
which is here.\n");
```

- This gives us a much more flexible construct than a basic *if* statement but it alone can't solve our problems with the menu program.

Nested if *Statements*

- Part of the problem is that all of the *if* statements we're using are independent from one another.
- What we need to do is be able to link these together.

- This involves putting *if* statements inside other *if* statements which is called *nesting*.

- Basic *if* statements can be nested, for example:

```
if(age >= 18)
{
   if(height >= 175)
   {
        printf("Potential police recruit.\n");
   }
}
```

- Notice how this approach allows us to "filter" various conditions.
- Only if the first condition is true will the second condition be tried and we know if the code is executed that both conditions will be true.
- This makes this directly equivalent to the boolean AND operator:

```
if((age >= 18) && (height >= 175))
{
   printf("You are a potential police
   officer.\n");
}
```

- So ultimately nested *if* statements aren't anything new.

- However, nested *if-else* statements are very powerful.

Nested if-else *Statements*

- Nesting *if-else* statements allows us to link these statements together which forms a very useful construct.
- We can use this to solve the second and third problems with our menu program.

Here is the pseudocode for our new solution:

```
if response is 'a' or response is 'A'
   print "You have selected option 1"
else
   if response is 'b' or response is 'B'
      print "You have selected option 2"
   else
      if response is 'c' or response is 'C'
         print "You have selected option 3"
      else
         print “You haven't selected a valid
      option.”
```

And in C code:

```
if((response == 'a') || (response == 'A'))
{
   printf("Your have selected option 1\n");
}
else
   if((response == 'b') ||
            (response == 'B'))
   {
      printf("You have selected option 2\n");
   }
   else
      if((response == 'c') ||
            (response == 'C'))
      {
         printf("You have selected option 3\n");
      }
      else
      {
         printf("You haven't selected a valid
         option.\n");
      }
```

- This solves both our previous problems.
- Nested if-else statements are extremely useful and powerful technique very commonly used in programming.

SWITCH-CASE STATEMENTS

- You will find nested *if-else* statements are needed quite often in writing your algorithms and programs.

- However, they have a disadvantage that if there are many different conditions that need to be tested for, this involves quite a lot of typing.

- It is also common for the test conditions to be very, very similar making a lot of that typing quite redundant and repetitive.
- For example, in the code for our menu system the condition always began with `response ==`

- Many languages including C provide a construct called a *switch-case* which provides an alternative to nested *if-else* for certain tasks.
- Although for a small number of conditions *if-else* is suitable, *switch-case* can often save quite a bit of time and typing.

- However, you can only use the *switch-case* construct when you are comparing a variable and a constant for equality.
  - So a *switch-case* is far less flexible than an *if-else*.

Switch-Case *Example*

Here is the nested *if-else* example from above converted to a *switch-case*:

```
switch(response)
{
   case 'a':
      printf("You have selected option 1\n");
      break;
   case 'b':
      printf("You have selected option 2\n");
      break;
   case 'c':
      printf("You have selected option 3\n");
      break;
   default:
      printf("You haven't selected a valid
   option.\n");
}
```

Things to note:

- You can put as many statements as you like for each case, although in the example above there is just one.
- The `break` statements must be put at the end of each case otherwise execution will "fall through" to the next case even if the condition doesn't match.
- If no case matches then the default case is executed.
- However, the default case is optional and, if left out, the *switch-case* will do nothing if none of the options match.
- Although the effect is the same, the *switch-case* is somewhat easier to read than the nested *if-else*, particularly if there are many cases.
- There's also a lot less typing involved!

Full Menu Example

Here is a complete example of the menu program using *switch-case*. Note the enhancement to the code above to handle capital input.

```
/* Basic Menu Program in C demonstrating
   switch-case statements.
*/

#include <stdio.h>

int main()
{
   char response;

   /* Print menu giving choices */
   printf("You have three choices.\n");
   printf("Enter a for option 1\n");
   printf("Enter b for option 2\n");
   printf("Enter c for option 3\n");

   /* Read in user's response */
   scanf("%c%*c", &response);

   /* Perform appropriate response */
   switch(response)
   {
     case 'a':
     case 'A':
        printf("You have selected option 1\n");
        break;
     case 'b':
     case 'B':
        printf("You have selected option 2\n");
        break;
     case 'c':
```

```
        case 'C':
            printf("You have selected option 3\n");
            break;
        default:
            printf("You haven't selected a valid
        option.\n");
    }

    return(0);
}
```

- Note how the "fall through" behaviour is used to allow matching both capital and lower case input.

- This program is now quite complete except it still doesn't solve our final problem:
  - After selecting a single option, the program exits rather than letting the user go back to the menu to select another option.

- To do this, we need to use *iteration* which is the next topic.

SUMMARY

- Our code to date has just been *sequential*, however, *selection* constructs allow us to write programs that behave differently depending on certain conditions.

- We can use the relational operators (like ==, >, != etc.) to construct boolean (T/F) expressions used to make decisions about what a program will do.
- We can also combine these boolean expressions with boolean operators like AND, OR and NOT (`&&`, `||` and `!` in C) to construct virtually any boolean logical expressions needed.

- *if* and *if-else* statements are used to make simple decisions in C.
  - *if* statements will execute a block of code if a condition is true.
  - *if-else* statements will execute a block of code if the condition is true and execute a different block if it is false.

- We can nest *if-else* statements together to build very sophisticated selection constructs, for example in our menu program.

- However, this involves a lot of typing and so often a *switch-case* statement proves easier to write.