COMP1917: Computing 1

3. Making Choices

Reading: Moffat, Chapter 3.

Outline

- the `if` construct
- relational and logical operators
- `if-else`
- conditional expressions (optional topic)
- `switch` statements (optional topic)

The `if` construct

```
if( expression )
    statement
```

- used to decide if *statement* should be executed.
- There is no "boolean" type in C. Instead, zero is regarded as "FALSE" and anything non-zero is regarded as "TRUE".
- *statement* is executed if the evaluation of *expression* is non-zero.
- *statement* is NOT executed if the evaluation of *expression* is zero.
- "*statement*" could be a single instruction, or a series of instructions enclosed in { }

The `if` construct *cont.*

```
if( expression )
    statement1
else
    statement2
```

- Used to decide if *statement1* should be executed or *statement2*.
- *statement1* is executed when the evaluation of *expression* is non-zero.
- *statement2* is executed if the evaluation of *expression* is zero.

The `if` construct *cont.*

Here is an example:

```
if( x ) {
   printf( "x is non-zero\n" );
}
else {
   printf( "x is zero\n" );
}
```

The `if` construct *cont.*

- Notice that '{' and '}' are used to enclose / group a number of statements together.
- For this course, we insist you always use curly braces { } even when there is only one statement inside.
- Indentation is used to make the code clearer and easier to read.
- Take note of where the semicolons ';' are being used.

Style

- As you can see from the code examples, indentation is very important in promoting the readability of the code.
- Each logical block of code is indented.
- Each '{' and '}' are indented to the appropriate logical block level.

Style 1:

```
if(x)
{
    statement;
}
```

Style 2: `(preferred)`

```
if( x ) {
    statement;
}
```

Relational and logical operators

Relational Operators		
>	*a* > *b*	*a* Greater Than *b*
>=	*a* >= *b*	*a* Greater Than Or Equal *b*
<	*a* < *b*	*a* Less Than *b*
<=	*a* <= *b*	*a* Less Than Or Equal *b*
Equality Operators		
==	*a* == *b*	*a* Equal to *b*
!=	*a* != *b*	*a* Not Equal to *b*
Logical Operators		
&&	*a* && *b*	*a* logical AND *b*
||	*a* || *b*	*a* logical OR *b*
Unary Operator		
!	! *a*	Logical NOT *a*

Single or double equals

- note the difference between = and ==

  `x = y;`

  (store the value of `y` into `x`)

  `if( x == y ) ...`

  (check whether the values of `x` and `y` are equal)

- in C, an assignment evaluates to the value assigned.
  Thus it is legal (but not encouraged) to write

  `x = y = z = 0;`

- if you accidentally use = instead of == the program will malfunction
  but the compiler will not warn you (unless you use the `-Wall` option)

Lazy evaluation

Both `&&` and `||` are evaluated in a lazy manner from left to right.

Once the truth or falsehood is determined, the evaluation stops.

Example:

```
if(( x != 0.0 )&&( sin( 1.0/x ) > 0.5 )) {
    ...
}
```

If the first expression `( x != 0.0 )` fails, the program will not attempt to evaluate the second expression (thus avoiding a divide-by-zero error). "Non-lazy" equivalents are `&` and `|` .

Unary Negation operator

The unary negation operator converts a non-zero operand into 0 (zero) and a zero operand into 1 (one). For example,

```
if ( !( height <= 130 && width <= 240 ) {
    printf("Envelope too large!\n");
}
```

.. is the same as ..

```
if ( height > 130 || width > 240 ) {
    printf("Envelope too large!\n");
}
```

Operator Precedence

Operators	Associativity
(function call) `[]` `->` .	L → R
`!` `~` `++` `--` `+` `-` `*` `&` (*type*) sizeof	R → L
`*` / %	L → R
+ −	L → R
`<<` `>>`	L → R
`<` `<=` `>` `>=`	L → R
== !=	L → R
`&`	L → R
`^`	L → R
`|`	L → R
`&&`	L → R
`||`	L → R
?:	R → L
= += −= *= /= %= `&=` `^=` `|=` `<<=` `>>=`	R → L
,	L → R

The `if-else` statement

So far you've seen an `if` statement that looks like:

```
if( expression )
    statement1
else
    statement2
```

But what if you have more conditions to test?

Complex `if-else` statement

When you nest two or more `if` statements together, *e.g.*:

```
if( expression1 )
    if( expression2 )
        if( expression3 )
            statement1
        else
            statement2
```

The rule is that the last `else` is associated with the closest previous `if` statement that does not (yet) have an `else` component.

Avoid "dangling `else`"

To force the `else` to be associated differently, *e.g.*: say you want the else to be associated with the second `if`, then:

```
if( expression1 ) {
    if( expression2 ) {
        if( expression3 )
            statement1
    }
    else
        statement2
}
```

By using `{ }` braces, you force the association you want.

It is good programming style to always include braces, for clarity.

The `else if` statement

You can use the "`else if`" statement to create a multi-way decision chain, i.e.:

```
if( expression1 )
    statement1
else if( expression2 )
    statement2
else if( expression3 )
    statement3
else
    statement4
```

Each expression is evaluated in order until one is found to be "true", which then results in that respective statement being executed.

The `else if` statement *cont.*

If none of the expressions is found to be true, then the statement associated with the last `else` is the “catch-all” (i.e. “none of the above”) case that will be executed.

Please note the indentation used, it is suggested that you follow the same indentation strategy in order to make the code more uniform and more readable.

Example: Days in the Month

```
if( month == 2 ) {       // February
   if( year% 4 == 0 &&( year%100 != 0 || year%400 == 0 )) {
       num_days = 29;    // Leap Year
   }
   else {
       num_days = 28;
   }
}
else if( month==4 || month==6 || month==9 || month==11 ) {
   num_days = 30;        // April, June, September, November
}
else {
   num_days = 31;
}
```

Conditional Expressions (Optional Topic)

Conditional expressions have the form:

$$expr_1 \; ? \; expr_2 \; : \; expr_3$$

For example,

```
z = ( x < a ) ? x : a ;
```

.. is the same as ..

```
if( x < a )
  z = x;
else
  z = a;
```

The `switch` statement (Optional Topic)

Like the multi-way `else if` statement, the `switch` statement behaves in a similar manner:

```
switch( expression ) {
  case const-expr:
     statements
  case const-expr:
     statements
  default:
     statements
}
```

Example of `switch` (Optional Topic)

```
switch( month ) {
 case 2:
     num_days =
   ( year% 4 == 0 &&( year%100 != 0 || year%400 == 0 )) ?
           29 : 28 ;
     break;
 case 4: case 6: case 9: case 11:
     num_days = 30;
     break;
 default:
     num_days = 31;
     break;
}
```

The `switch` statement (Optional Topic)

Things to note:

- each `case` must be a constant integer and not an expression;
- the `default` is optional;
- if a `case` matches the *expression* value, then execution starts at that case;
- if none of the `cases` match, then the `default` action is executed;
- if there is no `default` and no `case`'s match, then no action takes place;
- the `case` and `default` can occur in any order (but only one `default` is allowed per `switch` statement).

The `switch` statement (Optional Topic)

Things to note:

- `break` is used to force an immediate exit from the `switch` statement upon a `case` *const-expr* match.
- if `break` is omitted, then execution will flow on into the next `case` label; this is called “falling through” from one case to another.
- “falling through” is not considered good practice and should be avoided where possible. If you must do it, then make sure you flag it in your comments and make it very obvious.
- it is good practice to put a `break` at the end of the the last case even though it is not strictly necessary.