Chapter 16: Selection Statements

Section 16.1: if () Statements

One of the simplest ways to control program flow is by using `if` selection statements. Whether a block of code is to be executed or not to be executed can be decided by this statement.

The syntax for `if` selection statement in C could be as follows:

```
if(cond)
{
  statement(s);  /*to be executed, on condition being true*/
}
```

For example,

```
if (a > 1) {
    puts("a is larger than 1");
}
```

Where `a > 1` is a *condition* that has to evaluate to **`true`** in order to execute the statements inside the `if` block. In this example "a is larger than 1" is only printed if `a > 1` is true.

`if` selection statements can omit the wrapping braces `{` and `}` if there is only one statement within the block. The above example can be rewritten to

```
if (a > 1)
    puts("a is larger than 1");
```

However for executing multiple statements within block the braces have to used.

The *condition* for `if` can include multiple expressions. `if` will only perform the action if the end result of expression is true.

For example

```
if ((a > 1) && (b > 1)) {
    puts("a is larger than 1");
    a++;
}
```

will only execute the `printf` and `a++` if **both** `a` and `b` are greater than `1`.

Section 16.2: Nested if()...else VS if()..else Ladder

Nested `if()...else` statements take more execution time (they are slower) in comparison to an `if()...else` ladder because the nested `if()...else` statements check all the inner conditional statements once the outer conditional `if()` statement is satisfied, whereas the `if()..else` ladder will stop condition testing once any of the `if()` or the `else if()` conditional statements are true.

An `if()...else` ladder:

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if ((a < b) && (a < c))
  {
    printf("\na = %d is the smallest.", a);
  }
  else if ((b < a) && (b < c))
  {
    printf("\nb = %d is the smallest.", b);
  }
  else if ((c < a) && (c < b))
  {
    printf("\nc = %d is the smallest.", c);
  }
  else
  {
    printf("\nImprove your coding logic");
  }
  return 0;
}
```

Is, in the general case, considered to be better than the equivalent nested `if()...else`:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if (a < b)
  {
    if (a < c)
      {
        printf("\na = %d is the smallest.", a);
      }
    else
      {
        printf("\nc = %d is the smallest.", c);
      }
  }
  else
  {
    if(b < c)
    {
      printf("\nb = %d is the smallest.", b);
    }
    else
    {
      printf("\nc = %d is the smallest.", c);
    }
  }
  return 0;
}
```

Section 16.3: switch () Statements

`switch` statements are useful when you want to have your program do many different things according to the value of a particular test variable.

An example usage of `switch` statement is like this:

```
int a = 1;

switch (a) {
case 1:
    puts("a is 1");
    break;
case 2:
    puts("a is 2");
    break;
default:
    puts("a is neither 1 nor 2");
    break;
}
```

This example is equivalent to

```
int a = 1;

if (a == 1) {
    puts("a is 1");
} else if (a == 2) {
    puts("a is 2");
} else {
    puts("a is neither 1 nor 2");
}
```

If the value of `a` is 1 when the `switch` statement is used, `a is 1` will be printed. If the value of `a` is 2 then, `a is 2` will be printed. Otherwise, `a is neither 1 nor 2` will be printed.

`case n:` is used to describe where the execution flow will jump in when the value passed to `switch` statement is *n*. *n* must be compile-time constant and the same *n* can exist at most once in one `switch` statement.

`default:` is used to describe that when the value didn't match any of the choices for `case n:`. It is a good practice to include a `default` case in every switch statement to catch unexpected behavior.

A **`break`**`;` statement is required to jump out of the `switch` block.

Note: If you accidentally forget to add a **`break`** after the end of a `case`, the compiler will assume that you intend to "fall through" and all the subsequent case statements, if any, will be executed (unless a *break* statement is found in any of the subsequent cases), regardless of whether the subsequent case statement(s) match or not. This particular property is used to implement Duff's Device. This behavior is often considered a flaw in the C language specification.

Below is an example that shows effects of the absence of **`break`**`;`:

```
int a = 1;

switch (a) {
case 1:
case 2:
```

```
    puts("a is 1 or 2");
case 3:
    puts("a is 1, 2 or 3");
    break;
default:
    puts("a is neither 1, 2 nor 3");
    break;
}
```

When the value of `a` is 1 or 2, `a is 1 or 2` and `a is 1, 2 or 3` will both be printed. When `a` is 3, only `a is 1, 2 or 3` will be printed. Otherwise, `a is neither 1, 2 nor 3` will be printed.

Note that the `default` case is not necessary, especially when the set of values you get in the `switch` is finished and known at compile time.

The best example is using a `switch` on an **`enum`**.

```
enum msg_type { ACK, PING, ERROR };
void f(enum msg_type t)
{
  switch (t) {
  case ACK:
    // do nothing
    break;
  case PING:
    // do something
    break;
  case ERROR:
    // do something else
    break;
  }
}
```

There are multiple advantages of doing this:

- most compilers will report a warning if you don't handle a value (this would not be reported if a `default` case were present)
- for the same reason, if you add a new value to the **`enum`**, you will be notified of all the places where you forgot to handle the new value (with a `default` case, you would need to manually explore your code searching for such cases)
- The reader does not need to figure out "what is hidden by the `default:`", whether there other **`enum`** values or whether it is a protection for "just in case". And if there are other **`enum`** values, did the coder intentionally use the `default` case for them or is there a bug that was introduced when he added the value?
- handling each **`enum`** value makes the code self explanatory as you can't hide behind a wild card, you must explicitly handle each of them.

Nevertheless, you can't prevent someone to write evil code like:

```
enum msg_type t = (enum msg_type)666; // I'm evil
```

Thus you may add an extra check before your switch to detect it, if you really need it.

```
void f(enum msg_type t)
{
   if (!is_msg_type_valid(t)) {
      // Handle this unlikely error
   }
```

```
    switch(t) {
     // Same code than before
    }
}
```

Section 16.4: if () ... else statements and syntax

While `if` performs an action only when its condition evaluate to **`true`**, `if` / `else` allows you to specify the different actions when the condition **`true`** and when the condition is **`false`**.

Example:

```
if (a > 1)
    puts("a is larger than 1");
else
    puts("a is not larger than 1");
```

Just like the `if` statement, when the block within `if` or `else` is consisting of only one statement, then the braces can be omitted (but doing so is not recommended as it can easily introduce problems involuntarily). However if there's more than one statement within the `if` or `else` block, then the braces have to be used on that particular block.

```
if (a > 1)
{
    puts("a is larger than 1");
    a--;
}
else
{
    puts("a is not larger than 1");
    a++;
}
```

Section 16.5: if()...else Ladder Chaining two or more if () ... else statements

While the `if ()... else` statement allows to define only one (default) behaviour which occurs when the condition within the `if ()` is not met, chaining two or more `if () ... else` statements allow to define a couple more behaviours before going to the last `else` branch acting as a "default", if any.

Example:

```
int a = ... /* initialise to some value. */

if (a >= 1)
{
    printf("a is greater than or equals 1.\n");
}
else if (a == 0) //we already know that a is smaller than 1
{
    printf("a equals 0.\n");
}
else /* a is smaller than 1 and not equals 0, hence: */
{
    printf("a is negative.\n");
}
```