8.5. Sequence points

Associated with, but distinct from, the problems of real-time programming are *sequence points*. These are the Standard's attempt to define when certain sorts of optimization may and may not be permitted to be in effect. For example, look at this program:

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

int i_var;
void func(void);

main(){
        while(i_var != 10000){
                func();
                i_var++;
        }
        exit(EXIT_SUCCESS);
}

void
func(void){
        printf("in func, i_var is %d\n", i_var);
}
```

Example 8.6

The compiler might want to optimize the loop so that `i_var` can be stored in a machine register for speed. However, the function needs to have access to the correct value of `i_var`so that it can print the right value. This means that the register must be stored back into `i_var` at each function call (at least). When and where these conditions must occur are described by the Standard. At each sequence point, the side effects of all previous expressions will be completed. This is why you cannot rely on expressions such as:

```
a[i] = i++;
```

because there is no sequence point specified for the assignment, increment or index operators, you don't know when the effect of the increment on `i` occurs.

The sequence points laid down in the Standard are the following:

- The point of calling a function, after evaluating its arguments.
- The end of the first operand of the `&&` operator.
- The end of the first operand of the `||` operator.
- The end of the first operand of the `?:` conditional operator.
- The end of the each operand of the comma operator.
- Completing the evaluation of a full expression. They are the following:
  - Evaluating the initializer of an `auto` object.
  - The expression in an ‘ordinary’ statement—an expression followed by semicolon.
  - The controlling expressions in `do`, `while`,`if`, `switch` or `for`statements.
  - The other two expressions in a for statement.
  - The expression in a `return` statement.
