Lecture 2: Iterations

Introduction

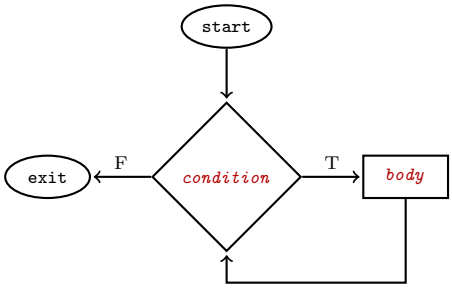
The second type of control structure is the *loop*. It repeatedly executes a set of statements until a condition becomes false. Every loop contains a body and a condition and can be nested for more complex operations.

While Loop

The *while loop* is a general purpose loop, typically, used for authentication such as a *sentinel loop*, a loop that terminates only when a specific *sentinel value* is encountered. Its syntax is

`while(`*condition*`)` {*body*}

It continually evaluates its condition and executes its body until the condition becomes false.



Example:
A guessing game that requires the user to determine a specific number from a range of numbers is a scenario that requires a sentinel loop as illustrated in the code segment below

```
int value = rand() % 10 + 1; // a random number between 1 and 10
int guess;
std::cout << "Guess the number between 1 and 10:";
std::cin >> guess;

while(guess != value)
{
  std::cout << "Try again:";
  std::cin >> guess;
}
std::cout << "You got it!\n";
```

The statement `"You got it!"` will not display until the guess equals the desired value.

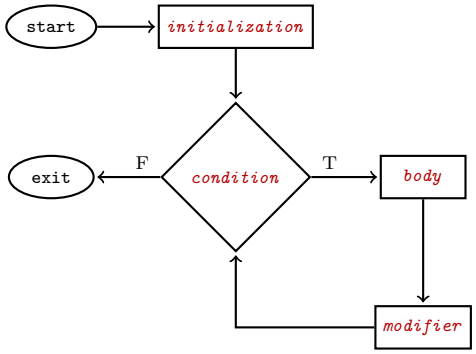
For Loop

The *for loop* is a sequential loop, it is primarily used to traverse through sequences or collections. Its syntax is

`for(`*initialization*`;`*condition*`;`*modifier*`)` {*body*}

Its header consists of three components that represent the start, end, and step of the traversal of a sequence, respectively. The initialization section can either initialize (or declare) multiple local variables of the loop with the same type or assign values to preexisting variables. While the modifier section is, typically, used to update the values of variables from the initialization section.

It begins by executing its initialization section; afterward, it repeatedly evaluates its condition, executes its body, and then its modifier section until its condition evaluation becomes false.



Example:
To display the first 100 even integers starting from 6 (a sequence) can easily be done with a for loop as the code segment illustrates below

```
for(int i = 0;i < 100;i += 1)
{
  std::cout << (6 + 2 * i) << "\n";
}
```

Since 100 values need to be displayed, the for loop initializes the variable *i* to 0, will stop when *i* becomes 100, and increments *i* by 1 after each *iteration* (its body execution). Hence, *i* will become the values 0 through 99 (a total of 100 integers). Next, the expression `(6 + 2 * i)` implies that twice *i* plus 6 will be displayed for each iteration; thus, even numbers from 6 through 204 will display as desired.

There is a simplified version of the for loop known as a *for-each loop*. Its syntax is

`for(`*declaration* `:` *container*`)` {*body*}

where *declaration* is a variable declaration and *container* is any collection that supports iteration such as standard library container classes, C arrays, etc. The declaration variable holds the next element in the collection for each loop iteration. Elements of the collection can be modified if the declaration variable is a reference variable.

Example:
Displaying the characters of a string variable on separate lines can be accomplished easily with the for-each loop as the code segment illustrates below

```
std::string str = "Hello World";
for(char ch : str)
{
  std::cout << ch << "\n";
}
```

Interrupt Statements

Sometimes it is necessary to interrupt the flow of a loop. There exist two statements that allow such actions:

- **Break Statement:** It terminates the loop immediately. Its syntax is

  `break;`

- **Continue Statement:** It skips the current iteration and proceeds to the next. Its syntax is

  `continue;`

These statements are ideal interrupting for infinite loop scenarios such as games and simulations and omitting the execution of several body statements.

Example:
The game hangman should run until the player either guesses all the letters of the selected word or runs out of tries. Additionally, it skips invalid inputs and repeated guesses. The code segment below illustrates a hangman game that utilizes interrupt statements.

```
std::string word = "instruction", grid = std::string(word.length(),'_'), guess;
char ch;
int count = 0;
const int TRIES = 5;

while(true)
{
  std::cout << "Word: " << grid << "\nGuesses: " << guess << "\nEnter letter: ";
  std::cin >> ch;
  ch = tolower(ch);

  if(!isalpha(ch) || grid.find(ch) == std::string::npos)
  {
    continue; // skips iteration if invalid or repeated character is read
  }
  else if(word.find(ch) == std::string::npos)
  {
    //traverse word to modify the grid to include the valid character
    //in its appropriate locations
    for(int n = word.find(ch);n != std::string::npos;n = word.find(ch,n+1))
    {
      grid.at(n) = ch;
    }
  }
  else
  {
    count += 1; //increment count of incorrect character guesses
  }
  guess += ch;

  if(count == TRIES && grid == word)
  {
    std::cout << ((grid == word)?("Success!"):("Failure!")) << "\n";
    break; //terminates loop given that an exit condition a has been reached
  }
}
```