Loops

Ajit Rajwade

Refer: Chapter 7 of the book by Abhiram Ranade

Loops

- We have seen the `repeat` loop already.
- As you see, the loop runs for a fixed number of iterations.
- But `repeat` is not part of original C++, but part of the simplecpp package.
- `repeat` is easy to use, but there are other loops in C++ that potentially accomplish more things.
- Consider that you wanted to print the squares of all integers from -50 to +50.
- We will do so by means of a `while` loop, a very commonly used loop in C++.

`while` Loop

- The syntax of the while loop is:

  ```
  while (condition) body
  ```

- `body` can be a single statement or else an entire block.
- If `condition` evaluates to `true`, then `body` is executed, otherwise it is ignored and you exit the `while` loop. In the latter case, we say that the condition “failed”.
- After the body is executed, the condition is again evaluated. If it is true, the body is executed, otherwise the while loop is exited.
- This continues until the condition is false (and then the `while` loop is exited).

`while` loop versus `repeat` loop

```
main_program{

int i=-50;

while (i <= 50) {

cout << “The square of “ << i <<
“is “ << i*i;

i=i+1;

}

cout << “done”;

}
```

```
main_program{

int i=-50;

repeat (101) {

cout << “The square of “ << i <<
“is “ << i*i;

i=i+1;

}

cout << “done”;

}
```

`while` loop versus `repeat` loop: the **difference**

```
main_program{

int i=-50;

while (i <= 50) {

cout << "The square of " << i << "is
" << i*i;

i=i+1;

}

cout << "done";

}
```

```
main_program{

int i=-50; int n = 101;

repeat (n) {

cout << "The square of " << i << "is
" << i*i;

i=i+1; if (n > 0) n = n - 10;

}

cout << "done";

}
```

The condition in the `while` loop uses variables whose values may **change** inside the body of the while loop. This **flexibility** exists in `while` but not in `repeat`. In the latter case, the loop runs for a **fixed** number of iterations. Even if you used a variable `n` for the number of iterations of `repeat`, and changed the value inside the body of `repeat`, it will still run for a number of iterations equal to the **initial** value of `n`.

`while` loop: average of a bunch of numbers

```
main_program{
bool flag; // flag indicates whether or not you want to read in another number
float avg = 0.0; int num,n=0;
cout << "do you want to continue: (yes=1), (no=0)?"; cin >> flag;
while (flag){ // run while flag is true
cout << "enter the next number: "; cin >> num;
n++;
avg += num;
cout << "do you want to continue: (yes=1), (no=0)?"; cin >> flag;
}
if (n > 0) {avg = avg/n;
cout << "The average is " << avg;
}
}
```

The `break` statement

- A `break` statement inside a `while` loop leads to termination of the `while` loop.
- Often, `break` in a `while` loop is used after an `if` statement is executed, i.e. you decide to terminate the loop if so and so condition is satisfied.
- We will re-write the “average of numbers” program using a `break` statement.
- Earlier on (go to the previous slide), the boolean variable `flag` was used in the `while` loop.
- A `break` statement must always be nested inside a loop or `switch` statement, otherwise you will get a **syntax error**.

The `break` statement

```
main_program{
bool flag; // flag indicates whether or not you want to read in another
number
float avg = 0.0; int num,n=0;
while (true){
cout << "do you want to continue: (yes=1), (no=0)?"; cin >> flag;
if (!flag) break; // ensures that the loop will terminate some time
cout << "enter the next number: "; cin >> num;
n++;
avg += num;
}
if (n > 0) {avg = avg/n;
cout << "The average is " << avg;
}
```

The condition in the `while` loop is simply `true`. Hence the while loop could **perhaps never terminate**. But the body of the `while` loop asks the user whether he/she wishes to continue by entering another number. If the answer to that is no, as indicated by the value of the variable `flag`, the `while` loop breaks. This program is slightly more compact than the previous one, as the `cout << "do you want to continue?";` is written out only once.

The `continue` statement

- Suppose we decided to write a program which computed the average of only non-negative numbers input by the user.
- In other words, if the user enters a negative number, then it is to be ignored.
- In such cases, the `continue` statement is used.
- When you encounter the `continue` statement inside a `while` loop, the rest of the loop is ignored and the control passes to the top of the loop.
- That is, the `while` loop condition is evaluated and if `true`, the body of the `while` loop is executed from the first statement onwards.

The `continue` statement

```
main_program{
bool flag; // flag indicates whether or not you want to read in another
number
float avg = 0.0; int num,n=0;
while (true){
cout << "do you want to continue: (yes=1), (no=0)?"; cin >> flag;
if (!flag) break; // ensures that the loop will terminate some time
cout << "enter the next number: "; cin >> num;
if (num < 0) continue; // ignore negative numbers - do not update n or avg
n++;
avg += num;
}
if (n > 0) {avg = avg/n;
cout << "The average is " << avg;
}
}
```

Can you rewrite this code snippet without using `continue`?

Beware: Infinite loops

- For various reasons, a `while` loop may never terminate.
- This happens when the condition of the `while` loop always evaluates to `true`.
- It is the **responsibility of the programmer** to ensure that the `while` loop terminates: **the compiler cannot point it out**!
- Of course, in some cases, an infinite loop is desired.
- For example, Google is running in an infinite loop, always waiting for users to enter their queries.
- But this is **undesirable** in the programs that you write in this course. Your program must **always terminate** (in a fairly short amount of time).
- For the purpose of this course, an infinite loop will imply a logical programming error.

Examples: infinite loops?

Which of these program snippets will cause an infinite loop? Find out for yourself. More importantly, find out **why** they will or will not cause an infinite loop. Note that none of these snippets will cause a syntax error.

```
int a = 0;
while (a < 10) {
      cout << a;
      if (a = 5)
         cout << "a equals 5";
      a++;
     }
```

```
float x = 0.1;
while (x != 1.1) {
  cout << "x = " << x;
  x += 0.1;
  }
```

```
while(cond);
{
//code
}
```

```
int i = 10;
for( ; ;)
{
cout << i;
}
```

```
unsigned int i;
for (i = 1; i != 0; i++) {
  /* loop code */
  }
```

First take a look at `for` loops and then answer these last two.

The `do while` loop

- Syntax: `do body; while (condition);`
- This is similar to the `while` loop except that the body is executed once **first** and **then** the condition is evaluated.
- Thereafter the body keeps getting executed until the condition evaluates to false.
- Thus the body of a `do while` loop is always executed **at least once**, whereas it may happen that the body of the `while` loop is never executed.
- The `do while` loop is handy to write more compact code if you know that the condition does not need to be evaluated the first time.
- For example in the earlier “computing the averages” program, let us assume that the user would always enter at least one number.
- A `do while` loop is useful here. See next slide.

The `do while` loop

```
main_program{
bool flag; // flag indicates whether or not you want to read in another
number
float avg = 0.0; int num,n=0;
do{
cout << "enter the next number: "; cin >> num;
n++;
avg += num;
cout << "do you want to continue: (yes=1), (no=0)?";   cin >> flag;
} while (flag);
avg = avg/n;
cout << "The average is " << avg;
}
```

The `for` loop

- A very popular language construct - perhaps even more than the `while` loop.
- Provides for compact code!
- Syntax: `for (initialization; condition; update) body;`
- Using a for loop to print the squares of all integers from -50 to 50:

```
for (i=-50;i<=50;i++) cout << i*i;
```

- Contrast with a `while` loop:

```
int i = -50;

while (i <= 50) {

cout << i*i;

i = i +1;

}
```

The `for` loop

- The `initialization` and `update` are required to be expressions.
- Usually, these expressions will involve some assignment (=) operations.
- One can also **declare and initialize a variable** inside `initialization` of a `for` loop. For example: `for (int i=-50;i<=50;i++) cout << i*i;`
- `condition` is a Boolean/logical (true or false) expression, similar to a `while` loop.
- Note: `initialization` is executed first, then the `condition` is evaluated.
- If the `condition` is `true`, the `body` is executed and lastly update is executed. `condition` is evaluated again.

The `for` loop

- If `condition` is `false`, the `for` loop terminates (and `update` is not executed).
- The `for` loop is executed for as many iterations until `condition` evaluates to `false`.
- Note: `initialization`, `condition`, `update`, `body` can be empty. An empty condition is always considered to be `true`.
- `for (;;) ;` is a syntactically valid, but useless `for` loop. In fact, it will be an infinite loop!
- Note that badly programmed `for` loops can result in infinite looping!

The `for` loop

- The variable(s) used in initialization and update are called **control variables**.
- A program written using a `for` loop can be replaced with an equivalent one that uses `while`, and vice versa.
- The choice of `for` or `while` is left to the programmer.
- Just like in a `while` loop, you can use `break` or `continue` inside a `for` loop.
- A `break` statement inside a `for` loop will cause the `for` loop to exit.
- If a `continue;` statement is encountered, then the rest of the `for` loop is ignored, but the **update part is executed** and *then* the condition is checked again.

Nesting of Loops

- `for` and `while` loops can be nested inside each other, any number of times.
- In fact, `for` and `while` loops can also nest `if else` loops or `switch` case statements, and they can be nested inside `if else` loops or `switch` case statements.
- You will see a few examples of these in later programs and chapters.
- One example of nesting of `for` loops: a program to take in information about `n` students. For each student, the program wants to take in the scores in some `m` subjects. And you want to compute an average of scores across `m` subjects for each student.
- Code for this will look something like what you see on the next slide.

Nesting of Loops

```
main_program{
int n = 10,m=3; double avg,score;
for (int i = 0; i < n ; i++) // outer loop for i = student index
{
     cout << "enter marks for student#: " << i << endl;
     avg = 0;
     for (int j = 0; j < m; j++) // inner loop, j = subject index
     {
          cout << "enter marks for subject# " << j << " of student #" << i;
          cin >> score;
          avg += score;
     } // close inner for loop (control variable j)
     avg = avg/m; cout << "Avg. marks for this student are: " << avg << endl;
} // close outer for loop (control variable i)
} // close main_program
```