Introduction to looping and repetition in programming

CS 10A – LOOPING PART 1

Allen Zhao - SRJC CS 10A

Repeating Commands

- More often than not, programming requires some form of repetition to better control the output and process the inputs on any application.
- In any programming language, repetition is known as looping. This allows us to repeat blocks of commands but we can introduce variable control to make it more useful than just repeating the exact same thing.
- By far, looping will be used the most in all manners of problem solving in programming.

Aspects of Loops

- Loops are controlled by logic. If the logic statement is true, then the loop will execute/repeat.
- To master the control of loops, make sure you understand how to update variables.
  - Incrementation (i.e. i++, i--, ++i, --i)
  - Use of +=, -=, *=, etc. (See slides on Shorthand)
- Loops use at least one variable to control their number of repeats. They’re often not fixed in value.
- It’s possible for a loop to be skipped altogether by starting out the initial condition as false.
- There are multiple ways to perform loops.

For Loops

- For loops is one of the most common ways to do loops.
- For loops are ideal of when you need to control specifically how many loops you want to run.
- You can also change the total number of loops to run mid-way through the loop execution.
- for(initializer; condition; changing factor)
  - The initializer specifies where the control variable starts.
  - When the condition becomes false, the loop stops and moves on.
  - The changing factor specifies how the control variable should update at the end of every loop.

For Loop Syntax

```
for(int i = 0; i < 5; i++)          // Variable i is declared specifically for this loop
{
        // Commands in there repeat 5 times.
        // Bracket use rules here are the same as if statements.
}
for(i = 0; i <= x; ++i)
{
        // In most loops, pre and post-increments behave the same.
        // In most cases, you want to use other variables (x in this case) that
        // can also update within a loop to dynamically control the limit.
        // This loop assumes i and x have already been declared elsewhere.
}
```

For Loop Example

Program

```
int x;
int main()
{
        cout << “Enter repeats: ”;
        cin >> x;
        for(int i = 0; i < x; i++)
                cout << i << endl;
        return 0;
}
```

Console

```
➢ ./a.exe
Enter repeats: 6
0
1
2
3
4
5
```

While Loops

- While loops are another popular form of loops.
- While loops are loops that, instead of using an actual counter, continue until a logic condition becomes false.
- A while loop basically controls a loop with an if statement, so you don’t have to control it.
- while(condition)
  - The condition is the same as how you would put a logic statement inside an if or else if statement.
  - The condition is checked at the end of every loop. Repeats if true.
  - It is critical that you remember to update the condition’s control variable within the loop.

While Loop Syntax

```
while(i < 5)
{
        // In this regard, a while loop can be identical to a for loop
        i++;    // Remember to include an update factor in the loop
}
while(someBoolean || someChar == ‘a’ && someStr == “test”)
{
        // Since the while loop relies on logic conditions instead of
        // numbers, you don’t have to use numbers. All the rules for
        // logic statements used in if and else if apply here as well.
}
```

While Loop Example

Program

```
int x = 0;
int main()
{       // End a program with a specific input
        while(x != 5)
        {
                cout << “Enter a value: ”;
                cin >> x;
        }
        cout << “Program terminated.\n”;
        return 0;
}
```

Console

```
➢ ./a.exe
Enter a value: 8
Enter a value: 1
Enter a value: 0
Enter a value: 4
Enter a value: 9
Enter a value: 2
Enter a value: 3
Enter a value: 5
Program terminated.
```

Common Loop Applications

- In calculation programs, loops can be useful for dealing with various input sizes. Loops can be used to repeat logic statements to check on the nature of the input.
- For programs that need to deal with ever changing data sets, loops are self-maintaining. Your program will automatically accommodate new values and inputs if you configure your loops correctly.
- Loops can be used to keep a program running continuously and stopped only at the user's command.

Tips on Applications

- In any kind of loop, control variables that avoid using hard numbers are far more useful.
- x = someString.length();
  - If you use x in a loop now, your loop will automatically configure its number of repeats to accommodate for varying string lengths.
- for(int i = 0; i < x; i++) {x /= 2; /*other commands*/}
  - In for loops, you're not limited to changing just the initial control variable. Nothing is stopping you from changing the condition as well if needed so long as you're using a variable.
- Remember that loops can also be used to count down, even if the most common application to count up.

Choosing For or While

- The two most common loop types are for and while. Knowing which one to choose for a given situation can help you design better code.
- Use a for loop when you want to specifically control the number of loops to be executed.
- Use a while loop when the number of loops is more nuanced and cannot be readily calculated.
- A while loop can easily mimic a for loop, but the reverse not so much. A for loop is usually easier to read though.

Looping Equivalency

For Loop Version

```
int main()
{
	for(int x = 0; x < 10; x++)
		cout << x << endl;

	return 0;
}
```

While Loop Version

```
int main()
{
	int x = 0;
	while(x < 10)
	{
		cout << x << endl;
		x++;
	}
	return 0;
}
```

Updating a Loop Control Variable

- The most common way to update loop control variables is to increment by 1. That is, x++ or x += 1.
- However, you can use other methods of incrementing and certainly not limited to just increments of 1. You can use other shorthand methods or even decrement.
- The changing factor of the for loop can be anything. It's just a command that is executed at the end of a loop.
- In a for loop, the update factor always happens at the end of the loop. In a while loop, you can update the variable anywhere in the loop, but this usually doesn't change much.
- There is no difference between a pre and post-increment inside a loop, but it still follows the same rules as before in assigning.

Advanced Iteration Application Example

Program

```
int main()
{
        for(int i = 1; i < 10; i *= 2)
                cout << i << endl;
        // i <<= 1 does the same thing

        return 0;
}
```

Console

```
➢ ./a.exe
1
2
4
8
```

Nesting Loops

- Just like with logic statements, you can also nest loops within each other.
- Nesting loops is part of an entire field in programming: algorithms. We'll only touch upon the basics of this later in the semester.
- More loops means more possibilities of making a mistake and causing an unintentional infinite loop. Be careful!
- Generally, every loop nested is that many times more repeats in the program, so the number of repeats can grow exponentially. Try to not nest more than two loops.

Nesting Example

Program

```
int main()
{
        for(int i = 0; i < 2; i++)
                for(int j = 0; j < 4; j++)
                        cout << j << endl;
        // So 2x4 = 8 repeats
        return 0;
}
```

Example

- ./a.exe

```
0
1
2
3
0
1
2
3
```

Infinite Loops

- Whenever you have a loop where the condition is perpetually true, then an infinite loop is created.
- Some times, this can be by design (more on that at a later date), but usually it's unintentional.
- Common mistakes include:
  - Setting a variable value inside the loop. Effectively a reset.
  - The condition is incorrect, often a typo.
  - Forgetting to do any kind of update in a while loop.
  - Using multiplication to update, but starting the variable at 0.
  - Using the wrong (single-letter) variable while nesting.
- Terminal users should use Ctrl+c to forcibly end programs in the event an infinite loop is encountered.

Tips on Debugging Loops

- Generally, it's a good idea to run the repeats one by one when you want debug your loops so that you can take your time to check your values with cout.
- Visual Studio users are used to using system("pause") in their programs to make it pause. This command doesn't work in Terminal and Linux systems. A more universal pause command would be cin.get()
  - cin.get() works across all C++ compilers.
- To advance past the pause command, just press Enter.

Using the Pause Command

- Sometimes, cin.get() fails. This is because you had used cin beforehand to bring in a value into the program. That time you pressed the Enter key sits in limbo only to then get picked up by cin.get().
- To avoid this bug, simply use cin.ignore() immediately after any kind of cin input.
- From now on, make sure that you always use cin.ignore() immediately after any use of cin >> … so that you may use cin.get() to freely pause your program at any time.

Using the Pause Command

```
int x;
int main()
{
        cout << “Enter a number: ”;
        cin >> x;               // Hitting the Enter key brings \n into the cin stream
        cin.ignore();           // Flushes out only the next character from the cin stream, namely the \n

        // Any code takes place here

        cout << “Paused. Press Enter to continue”;
        cin.get();              // Immediately looks into the input stream to get the first character
                                // Without ignore(), get() would have immediately picked up the floating \n,
                                // thus get() would fail to pause your program at all
        return 0;
}
```