Intro to loops

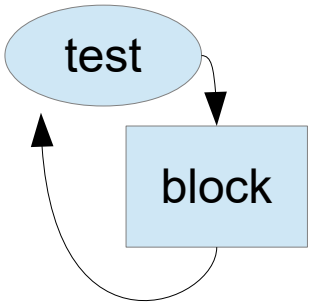
- repetition without recursion
- top-tested vs bottom-tested
- while loops
- do while loops
- for loops
- equivalence of loop types
- loop scope/local variables

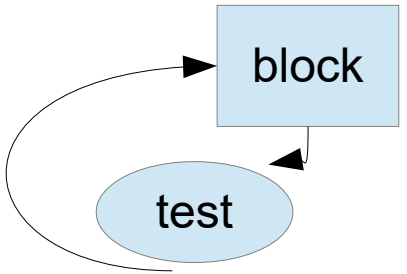
Repetition without recursion

- we've seen how to use recursion to get repetition in a program, but function calls/returns expensive compared to “regular” instructions (both in memory use and time)
- most languages (C++ included) provide a control structure that allows simpler repetition: informally called loops
- generally: a block of code that is to be performed repeatedly, plus a condition to checked periodically to see whether it should repeat again or leave the loop

Top-tested vs bottom-tested

- the condition-to-check is generally tested once each time the block of code may be run
- top-tested loops have the condition check before the block of code (at the top)
- bottom-tested loops have the condition check after the block of code (at the bottom)





while loops

- top tested
- tests a boolean expression, like those in if statements
- if condition is false then it skips past the block of code, goes on to rest of program
- if condition is true then is performs block of code, afterward coming back to check condition again

```
while (x < y) {
    cout << x << endl;
    x++;   // shorthand for x = x + 1
}
```

Example: countdown from N

```
// count down from N to 1 then display “done!”
int N = 10;
while (N > 0) {
    cout << N << endl;
    N--;  // shorthand for N = N - 1
}
cout << “done!” << endl;
// at start of each cycle it checks if N > 0
//    does the steps inside the loop if so,
//    otherwise skips just past the end of the loop
```

do while loops

- similar to while loops, but has the test at the bottom
- does the loop body, then at the bottom checks condition
- if condition is true it goes back to the top to do loop again
- otherwise it leaves loop

```
do {
    cout << x << endl;
    x++;
} while (x < y);
```

Example: get/check input

- get an int from user, repeat until they give value > 0

```
int userVal;
do {
   cout << “Enter a positive integer” << endl;
   cin >> userVal;
} while (userVal <= 0); // repeats if userVal still too small
cout << “Final answer: “ << userVal << endl;

// (so far we're not checking cin.fail)
```

Example: better error checking

- this time we'll check if cin failed (they entered non-integer), and flush the input buffer if so
- we'll also check if they entered an integer that was simply too small
- we'll use a boolean variable to keep track of whether or not the value they entered was a good one
- the loop test will simply look at the value in the variable to see if we need to repeat again

Err checking example continued

```
int userVal;

// true for bad val, false for good
bool badValue;

do {
   cout << “Enter positive int: “;
   cin >> userVal;

   // check for non-integer *first*
   if (cin.fail()) {
      badValue = true;
      cin.clear();
      cin.ignore(40, '\n');
      cout << “Try again” << endl;
   }
   // next check for too small
   else if (userVal <= 0) {
      badValue = true;
      cout << “Try again” << endl;
   }

   // otherwise it's ok
   else {
      badValue = false;
   }
} while (badValue);

// leaves loop if badValue false
cout << userVal << endl;
```

for loops

- top tested, fancier control structure
- control statement has spot for variable initialization, spot for the test condition, spot for the update statement
- the three “spots” separated by semi-colons
- initialization just happens once at the start
- condition is checked at top, before each pass thru body
- update statement is performed at the bottom, just after each pass through the body

Example: for loop

```
// count from 0 to 20 by twos
int x;
for (x=0; x<=20; x=x+2) {
   cout << x << endl;
}
cout << “done!” << endl;
```

```
// acts just like
int x;
x=0;
while (x<=20) {
   cout << x << endl;
   x=x+2;
}
cout << “done!” << endl;
```

equivalence of loop types

- anything written using one of the three loop types can be rewritten using either of the other two loop types
- might require slight tweaking of the logic, perhaps the addition or removal of an extra if test at the beginning
- which form is used is largely a programmer decision: whichever seems easiest/clearest in the current situation
- do while is often used for getting/checking input
- for loops often used when counting up/down through some sequence of values

loop scope/local variables

- scope is term used to describe where a variable or constant (or other item) is visible/accessible within code
- we've seen global scope, visible everywhere (after point of declaration)
- we've seen local (function) scope, visible only within current function (after point of declaration)
- now we'll also add local (loop) scope, visible only within current loop (after point of declaration)

Example: loop scope

```
int x;
x = 5;
while (x < 10) {
    int y; // loop local
    cout << “Enter int”;
    cin >> y;
    cout << (x*y) << endl;
}
// y is not usable outside loop
cout << x << endl;
```

Note: do-while and loop locals

- the condition check at the bottom of the loop is considered outside the loop body, so variables declared inside the loop are NOT usable there

```
do {
    int y;
    cin >> y:
} while (y < 1);  // compilation error, y not usable here!
```

- solution: declare the variable above the do {