From Selection to Repetition

- The *if* statement and *if/else* statement allow a *block* of statements to be executed selectively: based on a guard/test

```cpp
if (area > 20.0)
{
    cout << area << " is large" << endl;
}
```

- The *while* statement repeatedly executes a block of statements while the guard/test is true

```cpp
int month = 0;
while (month < 12)
{
    PrintCalendar(month, 1999);
    month += 1; // month = month + 1;
}
```
Semantics of while loop

```
if (test)
{
    statements;
    statements;
}
```

```
while (test)
{
    statements;
    statements;
}
```

Flowchart for while loop:
- **test**
  - **true**: statements
  - **false**: Next statement
- **false**: Statement list

Flowchart for if loop:
- **test**
  - **true**: statements
  - **false**: Next statement

A Computer Science Tapestry 5.2
Print a string backwards

- **Determine # characters in string, access each character**
  - What string functions do we have?
  - How many times should the loop iterate?

```cpp
    cout << "enter string: ";
    cin >> s;
    cout << s << " reversed is ";

    k = s.length() - 1; // index of last character in s
    while (k >\= 0)
    {
        cout << s.substr(k,1);
        k -= 1;
    }
    cout << endl;
```

- **Modify to create a new string that’s the reverse of a string.**
ReverseString as a function

● **First step, what is the prototype?**

```cpp
string Reverse(string s)
// pre: s = c_0c_1c_2...c_{n-1}
// post: return c_{n-1}...c_2c_1c_0
```

● **Second step, how do we build a new string?**
  ➤ Start with an empty string, 
  ➤ Add one character at a time using concatenation, +

```cpp
rev = rev + s.substr(k, 0);
```

● **Use Reverse to determine if a string is a palindrome**
Anatomy of a loop

- Initialize variables used in loop/loop test (before loop)
  - Loop test affected by initial values of variables
- The loop test or guard is evaluated before each loop iteration
  - NOT evaluated after each statement in loop
- The loop body must update some variable/expression used in the loop test so that the loop eventually terminates
  - If loop test is always true, loop is infinite

```java
k = s.length() - 1;
string rev = ""
while (k >= 0)
{
    rev = rev + s.substr(k, 1);
    k -= 1;
}
return rev;
```
Infinite loops

- Sometimes your program will be “stuck”, control-C to stop
  - What’s the problem in the loop below? Fixable?

```cpp
    cin >> num;
    int start = 0;
    while (start != 0)
    {
        start += 2;
        cout << start << endl;
    }
```

- It’s impossible to write one program that detects all infinite loops (the compiler doesn’t do the job, for example)
  - This can be proven mathematically, Halting Problem
  - Some detection possible, but not universally
Developing Loops

● Some loops are easy to develop code for, others are not
  ➤ Sometimes the proper loop test/body are hard to design
  ➤ Techniques from formal reasoning/logic can help

● Practice helps, but remember
  ➤ Good design comes from experience, experience comes from bad design

● There are other looping statements in addition to while, but they don’t offer anything more powerful, just some syntactic convenience
  ➤ for loop
  ➤ do–while loop
Factorial

- \( N! = 1 \times 2 \times \ldots \times N \) is “\( N \) factorial”, used in math, statistics

```cpp
int factorial(int n)
// pre: 0 <= n
// post: returns n! (1 \times 2 \times \ldots \times n)
```

- We’ll return the value of a variable `product`, we’ll need to accumulate the answer in `product`
  - The loop will iterate \( n \) times, multiplying by 1, 2, …, \( n \)
  - Alternatives: how many multiplications are needed?
  - If `product` holds the answer, then `product == n!` when the loop terminates
    - Use this to help develop the loop
Factorial continued

- If product holds the answer, then \texttt{product} == n! when the loop terminates, replace n with count, the looping variable
  - Invariant: \texttt{product} == count!

```java
long Factorial(int num)
// precondition: num >= 0
// postcondition returns num!
{
    long product = 1;
    int count = 0;
    while (count < num)
    {
        count += 1;
        product *= count;
    }
    return product;
}
```
Long, int, and BigInt

- On some systems the type `long int` (`long`) provides a greater range than `int`
  - With 32-bit (modern) compilers/operating systems `int` is roughly \(-2\) billion to \(2\) billion, but on 16-bit machines the range is usually \(-32,768\) to \(32,767\) [how many values?]
  - 13! Is 1,932,053,504, so what happens with 14!

- The type `BigInt`, accessible via `#include "bigint.h"` can be used like an `int`, but gets as big as you want it to be
  - Really arbitrarily large?
  - Disadvantages of using `BigInt` compared to `int`?
Determining if a number is prime

- Cryptographic protocols depend on prime numbers
  - Determining if a number is prime must be “easy”
  - Actually factoring a number must be “hard”
  - What does hard mean? What factors affect difficulty?

- PGP (pretty good privacy) and e-commerce depend on secure/encrypted transactions
  - What are government restrictions on exporting PGP?
  - Different versions of Netscape in US and other countries?

- Sophisticated mathematics used for easy prime-testing, we’ll do basic prime testing that’s reasonably fast, but not good enough for encryption (why not?)
Determining Primality (continued)

- 2 is prime, 3 is prime, 5 is prime, 17 is prime, ... 137, 193?
  - To check 137, divide it by 3, 5, 7, 9, 11, 13
  - To check 193, divide it by 3, 5, 7, 9, 11, 13
    - Note that 14x14 = 196, why is 13 largest potential factor?
    - How do we determine if a number is divisible by another?

- We’ll check odd numbers as potential divisors
  - Treat even numbers as special case, avoid lengthy testing
  - Watch out for 2, special case of even number
  - Instead of odd numbers, what would be better as tests?
  - How many times will our testing loop iterate to determine if n is prime?
  - See primes.cpp for code
Details of *IsPrime* in *primes.cpp*

- Several different return statements are written, only one is executed when function executes
  - The *return* statement immediately tops, return to call
  - Some people think functions should have one return
    - Potentially easier to debug and reason about,
    - Often introduces extraneous variables/tests

- To assign a *double* value to an *int*, a *typecast* is used, tell the compiler that the loss of precision is ok
  - Fix all compiler warnings whenever possible
  - Make casts explicit, tell the compiler you know what you are doing

- What about complexity/efficiency of *IsPrime*?
C++ details: syntax and shorthand

- With while loops and variables we can write a program to do anything a program can be written for
  - Other language features make programs easier to develop and maintain: functions, if statements, other statements
  - Yet, we want to avoid needing to understand many, many language features if we don’t have to
  - You’ll read code written by others who may use features

- Loops are statements, can be combined with other loops, with if statements, in functions, etc.
- Other kinds of looping statements can make programming simpler to develop and maintain
- Similar shorthand for other language features: \( x = x + 1; \)
The **for** loop

- In many coding problems a definite loop is needed
  - Number of iterations known before loop begins and simple to calculate and use in loop (counting loop)
    - Example: length of string: print a string vertically

```cpp
void Vertical(string s)
// post: chars of s printed vertically

int len = s.length();  // for loop alternative
int k = 0;
for(k=0; k < len; k+= 1)
while (k < len)          {  cout << s.substr(k,0);
{ cout << s.substr(k,0);   }
k += 1;
}
```

- Initialization, test, update are localized into one place, harder to leave update out, for example
Example: add up digits of a number

- If we have a number like 27 or 1,618 what expression yields the number of digits in the number (hint, think log)

  ➤ Which digit is easiest to get, how can we access it?
  ➤ How can we chop off one digit at-a-time?

```c
int digitSum(int n)
// post: returns sum of digits in n
{
    int sum = 0; // what’s needed here?
    while (n > 0) // for loop alternative?
    {
        sum += n % 10; // what’s needed here?
        n /= 10; // for loop alternative?
    }
    return sum;
}
```
Shorthand for increment/decrement

● Lots of code requires incrementing a variable by one
  ➢ Three methods, using +, using +=, and using ++

    num = num + 1;
    num += 1;
    num++;  

● We use postincrement ++, also possible to write ++num
  ➢ These differ on when the increment is performed, but this
difference doesn’t matter when used as abbreviation for
the statement n += 1; in a single statement

● Similarly there are postdecrement (and predecrement)

    num = num - 1;     num -= 1;    num--;

The do-while loop

- The while loop may never execute, some loops should execute at least once
  - Prompt for a number between 0 and 100, loop until entered

```cpp
    do
      {   cout << "num in range [0..100] ";
          cin >> num;
      } while (num < 0 || 100 < num);
```

- Execute while the test/guard is true, in example above what must be true when loop terminates (de Morgan)?
Primming, loop-and-half problems

- Problem: enter numbers, add them up, stop when 0 entered
  ➤ What should loop test be?

```cpp
int sum = 0;
int num;
cin >> num; // prime the loop
while (num != 0)
{   sum += num;
    cin >> num;
}
cout << "total = " << sum << end;
```

➤ Code duplication problem: input (and perhaps prompt)
  code is repeated before loop and in loop
  • Why is duplicated code a bad thing? Alternatives?
Loop and a half: quasi infinite solution

- To avoid repeating code, include it in the body of the loop only, use a test to break out of the loop
  ➤ break statement exits (inner-most) loop

```cpp
int sum = 0;
int num;
while (true)
{
    cin >> num;
    if (num == 0) // get out of loop
    {
        break;
    }
    sum += num;
}
cout << "total = " << sum << end;
```
Alternative priming solution

- Force loop to execute once by giving tested variable a value
  ➤ What’s wrong with the solution below?

```c++
int sum = 0;
int num=-1;
while (num != 0)
{
    cin >> num;
    if (num != 0)
    {
        sum += num;
    }
}
cout << "total = " << sum << end;
```
Nested loops

- Sometimes one loop occurs in another
  - Generating tabular data
  - Sorting vectors (which is studied much later)
- Often code is simpler to reason about if inner loop is moved to another function

```cpp
int j,k;
for(j=1; j <= 6; j++)
{
    cout << j;
    for(k=0; k < j; k++)
    {
        cout << " \t " << j*k;
    }
    cout << endl;
}
```

- What’s printed? What’s the purpose of the inner loop?
Using classes

- Using only strings, ints, and doubles limits the kinds of programs we can write
  - What about graphics?
  - What about calendars, address books?
  - What about web-servers, games, …?

- Using object-oriented techniques means we develop new types that correspond to the real-world artifact we’re writing code for
  - What about an online roulette game?
  - What about appointment book that synchs with PalmV?

- New types are called classes, variables are called objects and objects are instances of a class, e.g., 3 for int, “hello” for string
The class **Date**

- **The class Date is accessible to client programmers by**
  - `#include "date.h"` to get access to the class
    - The compiler needs this information, it may contain documentation for the programmer
  - **Link** the implementation in `date.cpp`, which has been compiled to `date.o` (and maybe stored in a library)

- **The class Date models a calendar date:**
  - Month, day, and year make up the *state* of a *Date* object
  - *Dates* can be printed, compared to each other, day-of-week determined, # days in month determined, many other *behaviors*
    - Behaviors are called methods or member functions
**Constructing Date objects**

- See *usedate.cpp*

```cpp
int main()
{
    Date today;
    Date birthDay(7, 4, 1776);
    Date million(1000000L);
    Date badDate(3, 38, 1999);
    Date y2k(1, 1, 2000);

    cout << "today \t: " << today << endl;
    cout << "US bday \t: " << birthDay << endl;
    cout << "million \t: " << million << endl;
    cout << "bad date \t: " << badDate << endl;
    cout << y2k << " is a " << y2k.DayName() << endl;
}
```
Constructing/defining an object

- **Date** objects (like **string** objects) are constructed when they’re first defined
  - Three ways to construct a **Date**, what are they?
  - How have we constructed **string** objects?

- **Constructors for Date** objects look like function calls
  - We’ll see that constructor is special member function
  - Different parameter lists means different constructors

- **Once constructed many ways to manipulate a Date**
  - Increment it, subtract an int from it, print it, ...
  - **MonthName()**, **DayName()**, **DaysIn()**, ...
Finding Thanksgiving in the US

● Thanksgiving occurs on fourth Thursday in November

    Date Thanksgiving(int year)
    // post: return date for Thanksgiving in year

    cout << "what year ";
    cin >> year;
    cout << "bird day is " << Thanksgiving(year) << endl;

● How do we write the function?
  ➤ How is it similar to Labor Day, Mother’s Day, Flag Day?
  ➤ Can we generalize the function?
The class **Dice**

- **Accessible to client programmers using `#include "dice.h"`**
  - How do clients get access to implementation?
  - Why are quotes used instead of angle brackets `< .. >`?

- **What do we do with **Dice** outside of programs (real world)**
  - What would be nice to model with the class **Dice**?
  - What would be hard?

- **Dice objects will work as pseudo-random number generators**
  - Not truly random in a strict mathematical sense
  - Still useful to introduce randomness into programs
  - Some random numbers are more random than others
Using the class Dice

```cpp
int main()
{
    Dice cube(6);          // six-sided die
    Dice dodeca(12);       // twelve-sided die

    cout << "rolling " << cube.NumSides()
         << " sided die" << endl;
    cout << cube.Roll() << endl;
    cout << cube.Roll() << endl;
    cout << cube.Roll() << endl;
    cout << "rolled " << cube.NumRolls()
         << " times" << endl;

    // more here

    ● See roll.cpp, how is a Dice object constructed?
```
What you can and cannot do with Dice

- Cannot define a Dice object without specifying # sides
  
  ```
  Dice d(1);     // ok, but what is it?
  Dice cube;     // NOT ok, won’t compile
  ```

- How random is a Dice object – how can we test this?
  
  - Roll two Dice 10,000 times, count how many 2’s and 12’s
  - How can we test every valid roll? For n-sided Dice?
  - How many rolls needed to get a “pure Yahtzee”? (five six-sided Dice rolled, all yield the same value)
    - What techniques help in developing this loop/program?
    - What about two Dice, three Dice
Grace Murray Hopper (1906-1992)

- One of the first programmers on one of the first computers in the US
  - “third programmer on world’s first large-scale digital computer”
  - US Navy, later Admiral

“It’s better to show that something can be done and apologize for not asking permission, than to try to persuade the powers that be at the beginning”

- ACM Hopper award given for contributions before 30
  1994, Bjarne Stroustrup/C++
Loop development case study

- To calculate $a^n$ what are the options?
  - Use `pow` in `<cmath>`, when can’t `pow` be used?
  - Multiply $a \times a \times \ldots \times a$, n times?

- Using 1,024 multiplications to calculate $6^{1024}$ probably ok, but what about BigInt values raised to powers?
  $3 \times 3 = 9 \quad 9 \times 9 = 81 \quad 81 \times 81 = 6561 \quad 6561 \times 6561 = 43,046,721$
  - Number of multiplications needed for $3^{16}$?
  - Does this matter?

- How do we calculate $4^{125}$ or $17^{67}$?
  - Divide exponent in half
double Power(double base, int expo)
  // precondition: expo >= 0
  // postcondition: returns base^expo (base to the power expo)
{
    double result = 1.0;
    // invariant: result * (base^expo) = answer

- Is invariant true initially? Why?
- If we use return result; then what should loop test be?
  ➤ How will we make progress towards loop termination?
  ➤ What values will change in body of loop?
double Power(double base, int expo)
// precondition: expo >= 0
// postcondition: returns base^expo (base to the power expo)
{
    double result = 1.0;
    // invariant: result * (base^expo) = answer
    while (expo > 0)
    {
        if (expo % 2 == 0)
        {
            expo /= 2;         // divide by 2 how many times?
            // how does base change?
        }
        // more here for odd exponent
    }
    // more here for odd exponent
    return result;
}

● When exponent is even we divide it by two, what about when exponent is odd?
Code for odd exponents

double Power(double base, int expo)
   // precondition: expo >= 0
   // postcondition: returns base^expo (base to the power expo)
{
   double result = 1.0;
   // invariant: result * (base^expo) = answer
   while (expo > 0)
   {       if (expo % 2 == 0)      // code here from before
            else
               {
               
               }
   }

   return result;
}

● Use: result x base^{expo} = (result x base) x base^{expo/2} x base^{expo/2}
Factor out common code

double Power(double base, int expo)
   // precondition: expo >= 0
   // postcondition: returns base^expo (base to the power expo)
{
   double result = 1.0;
   // invariant: result * (base^expo) = answer
   while (expo > 0)
   {
      if (expo % 2 != 0) // exponent is odd
      {
         result *= base;
      }
      expo /= 2; // 4/2 == 2, 5/2 == 2
      base *= base; // (a*a)^(b/2) == a^b
   }
   return result;
}

Will this function work if base is a BigInt value? What must change?