

repeated computation

```
// lowhigh.cpp      Dave Reed      9/14/01
//
// This program converts low & high temperatures from Fahrenheit to Celsius.
//
//
#include <iostream>
using namespace std;

int main()
{
    double lowTemp, highTemp;
    cout << "Enter the forecasted low (in degrees Fahrenheit): ";
    cin >> lowTemp;
    cout << "Enter the forecasted high (in degrees Fahrenheit): ";
    cin >> highTemp;

    cout << endl
         << "The forecasted low (in Celsius): " << (5.0/9.0 * (lowTemp - 32)) << endl
         << "The forecasted high (in Celsius): " << (5.0/9.0 * (highTemp - 32)) << endl;

    return 0;
}
```

disadvantage: same formula is duplicated in the program

- requires extra typing (or cut-and-paste)
- must maintain consistency if make any changes

functions

ideal: encapsulate the computation of Fahrenheit → Celsius once, then reuse as needed

we have already seen practical examples of this

in mathematics, a *function* is a mapping from inputs to a single output
in programming, a *function* is a unit of computational abstraction

- cmath contained numerous useful functions
e.g., fabs sqrt pow ceil floor
- when you evaluate (a.k.a. *call*) a function in C++, specify the name with inputs (a.k.a. *arguments*) in parentheses
e.g., sqrt(9.0) pow(2, 10)
- the output (a.k.a. *return value*) of the function replaces the function call after evaluation

```
x1 = 0;  y1 = 0;  x2 = 3;  y2 = 4;
distance = sqrt(pow(x1-x2, 2) + pow(y1-y2, 2));
```

abstraction

in a complex world, we use many devices we don't fully understand

e.g., television set

QUESTION: could you build a TV? make major repairs?

QUESTION: how do you manage to watch TV?

ANSWER: you abstract away details, focus on simple interface

functions as units of abstraction

- functions encapsulate a computation that can be reused
- once defined & tested, the details can be abstracted away to perform the computation, only need to know how to call the function

QUESTION: would you know how to determine the square root of 1000?

ANSWER: you don't have to -- just know how to call `sqrt`

- for general-purpose functions, can even define in separate file and reuse

user-defined functions

to define a function, must specify

- name and return type: so compiler can recognize it later on
- parameters (variables that will store the inputs to the function)
- the computation and return value given the inputs

```
returnType functionName(type1 param1, type2 param2, . . .)
// Assumes: describe any assumptions about parameters
// Returns: describe return value of function
{
    POSSIBLY STATEMENTS PERFORMING COMPUTATION
}
return returnValue;
```

```
double FahrToCelsius(double tempInFahr)
// Assumes: tempInFahr is a temperature in degrees Fahrenheit
// Returns: equivalent temperature in degrees Celsius
{
    return (5.0/9.0 * (tempInFahr - 32));
}
```

function calling sequence

when a function is called, a specific sequence of events occurs:

1. arguments (inputs) in the function call are evaluated
2. space is allocated for the parameters in the function and the values of the arguments are stored there
3. statements in the function body are executed in order
4. when a return statement is encountered, the return value is evaluated and control returns to the calling point
5. the return value (output) replaces the function call in whatever expression

```
double FahrToCelsius(double tempInFahr)
// Assumes: tempInFahr is a temperature in degrees Fahrenheit
// Returns: equivalent temperature in degrees Celsius
{
    return (5.0/9.0 * (tempInFahr - 32));
}

.
.
.

cin >> lowTemp;
cout << FahrToCelsius(lowTemp) << endl;
```

repeated computation via a function

```
// lowhigh.cpp      Dave Reed      9/14/01
//
// This program converts low & high temperatures from Fahrenheit to Celsius.
////////////////////////////////////////////////////////////////////

#include <iostream>
using namespace std;

double FahrToCelsius(double tempInFahr)
// Assumes: tempInFahr is a temperature in Fahrenheit
// Returns: equivalent temperature in Celsius
{
    return (5.0/9.0 * (tempInFahr - 32));
}

int main()
{
    double lowTemp, highTemp;
    cout << "Enter the forecasted low (in degrees Fahrenheit): ";
    cin >> lowTemp;
    cout << "Enter the forecasted high (in degrees Fahrenheit): ";
    cin >> highTemp;

    cout << endl
         << "The forecasted low (in Celsius): " << FahrToCelsius(lowTemp) << endl
         << "The forecasted high (in Celsius): " << FahrToCelsius(highTemp) << endl;

    return 0;
}
```

define FahrToCelsius once

- encapsulates the computation

can call repeatedly in main

- specify diff. inputs (arguments) for diff. computations
- can ignore details

multiple inputs

for functions with multiple inputs, must have multiple parameters

- arguments are matched to parameters by order, i.e.,
 - 1st argument in 1st param,
 - 2nd argument in 2nd param,
 - ...

variables/parameters that appear in a function are distinct from variables in main

- each function, including main, defines its own environment
- can reuse variable names, but recognize that names are irrelevant to matching values

```
// distance.cpp      Dave Reed      9/14/01
//
// This program determines the distance between two points.
//
#include <iostream>
#include <cmath>
using namespace std;

double distance(double x1, double y1, double x2, double y2)
// Assumes: (x1,y1) and (x2,y2) are coordinates
// Returns: distance between the two coordinates
{
    return sqrt(pow(x1-x2, 2.0) + pow(y1-y2, 2.0));
}

int main()
{
    double x1, y1, x2, y2;
    cout << "Enter the x and y coordinates of the 1st point: ";
    cin >> x1 >> y1;
    cout << "Enter the x and y coordinates of the 2nd point: ";
    cin >> x2 >> y2;

    cout << endl << "The distance between (" << x1 << ", "
        << y1 << ") and (" << x2 << ", " << y2 << ") is "
        << distance(x1, y1, x2, y2) << endl;

    return 0;
}
```

function order

can have more than one function definition in a program

a function must be declared above any other function that calls it

- main must be the at the bottom of the program

this is far from ideal since it makes finding main hard

- would like main to be at the top

```
#include <iostream>
using namespace std;

const double PI = 3.14159;

double circleArea(double radius)
// Assumes: radius >= 0
// Returns: area of circle with specified radius
{
    return PI*radius*radius;
}

double circleCircumference(double radius)
// Assumes: radius >= 0
// Returns: circumference of circle with specified radius
{
    return 2*PI*radius;
}

int main()
{
    double pizzaDiameter, pizzaCost;
    cout << "Enter the diameter of the pizza (in inches): ";
    cin >> pizzaDiameter;
    cout << "Enter the price of the pizza (in dollars): ";
    cin >> pizzaCost;

    double pizzaArea = circleArea(pizzaDiameter/2);
    double costPerSqInch = pizzaCost/pizzaArea;

    cout << "Total area of the pizza: " << pizzaArea
        << " square inches" << endl;
    cout << "Price per square inch = $" << costPerSqInch << endl;
    cout << "Outer crust: " << circleCircumference(pizzaDiameter/2)
        << " inches" << endl;

    return 0;
}
```

function prototypes

can place function prototypes (headers) at top

- prototypes give compiler enough info to know that definition is coming
- given prototypes, the actual function definition can go anywhere

this is the preferred style:

1. comments
2. #includes
3. constants
4. function prototypes
5. main
6. function definitions

```
#include <iostream>
#include <iomanip>
using namespace std;

const double PI = 3.14159;

double circleArea(double radius);
double circleCircumference(double radius);

int main()
{
    // MAIN CODE AS BEFORE
}

////////////////////////////////////

double circleArea(double radius)
// Assumes: radius >= 0
// Returns: area of circle with specified radius
{
    return PI*radius*radius;
}

double circleCircumference(double radius)
// Assumes: radius >= 0
// Returns: circumference of circle with specified radius
{
    return 2*PI*radius;
}
```

functions calling functions

```
#include <iostream>
using namespace std;

double centimetersToInches(double cm);
double metersToFeet(double m);
double kilometersToMiles(double km);

int main()
{
    double distanceInKM;
    cout << "Enter a distance in kilometers: ";
    cin >> distanceInKM;

    cout << "That's equivalent to "
         << kilometersToMiles(distanceInKM)
         << " miles." << endl;

    return 0;
}

////////////////////////////////////
```

```
double centimetersToInches(double cm)
// Assumes: cm is a length in centimeters
// Returns: equivalent length in inches
{
    return cm/2.54;
}

double metersToFeet(double m)
// Assumes: m is a length in meters
// Returns: equivalent length in feet
{
    double cm = 100*m;
    double in = centimetersToInches(cm);
    return in/12;
}

double kilometersToMiles(double km)
// Assumes: km is a length in meters
// Returns: equivalent length in miles
{
    double m = 1000*km;
    double ft = metersToFeet(m);
    return ft/5280;
}
```

code libraries (first pass)

useful code can be placed in separate file

e.g., can place temperature & metric conversion code in convert.cpp

- can then #include in any file that needs it
note: user-defined files are indicated with quotes, not <>
- we will see later how to compile files separately, then link together

```
#include <iostream>
#include "convert.cpp"
using namespace std;

int main()
{
    double distanceInKM;
    cout << "Enter a distance in km: ";
    cin >> distanceInKM;

    cout << "That's equivalent to "
         << kilometersToMiles(distanceInKM)
         << " miles." << endl;

    return 0;
}
```

```
#include <iostream>
#include "convert.cpp"
using namespace std;

int main()
{
    double tempInFahr;
    cout << "Enter a temperature: ";
    cin >> tempInFahr;

    cout << "That's equivalent to "
         << FahrenheitToCelsius(tempInFahr)
         << " Celsius." << endl;

    return 0;
}
```

new example

suppose we wanted to print out verses of the song "Old MacDonald"

- could do it the hard way – cout statements to print each line

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl;
    cout << "And on that farm he had a cow, E-I-E-I-O." << endl;
    cout << "With a moo-moo here, and a moo-moo there," << endl;
    cout << " here a moo, there a moo, everywhere a moo-moo." << endl;
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl << endl;

    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl;
    cout << "And on that farm he had a horse, E-I-E-I-O." << endl;
    cout << "With a neigh-neigh here, and a neigh-neigh there," << endl;
    cout << " here a neigh, there a neigh, everywhere a neigh-neigh." << endl;
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl << endl;

    return 0;
}
```

DISADVANTAGES?

Old MacDonald (cont.)

DISADVANTAGES of version 1

- as add new verses, main gets longer and longer
- if want to change order of verses, lots of cut-and-paste

could define a function for each verse: no return value, just couts

- a function with no return value is assigned return type void
- function call is a stand-alone statement

```
void CowVerse()
// Results: displays the cow verse of Old MacDonald
{
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl;
    cout << "And on that farm he had a cow, E-I-E-I-O." << endl;
    cout << "With a moo-moo here, and a moo-moo there," << endl;
    cout << " here a moo, there a moo, everywhere a moo-moo." << endl;
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl << endl;
}
.
.
.
CowVerse();
```

version 2

main is much cleaner

if wanted to change order of verses, simply change order of calls in main

can easily repeat a verse

```
#include <iostream>
using namespace std;

void CowVerse();
void HorseVerse();

int main()
{
    CowVerse();
    HorseVerse();

    return 0;
}

////////////////////////////////////

void CowVerse()
// Results: displays the cow verse of Old MacDonald
{
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl;
    cout << "And on that farm he had a cow, E-I-E-I-O." << endl;
    cout << "With a moo-moo here, and a moo-moo there," << endl;
    cout << " here a moo, there a moo, everywhere a moo-moo." << endl;
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl << endl;
}

void HorseVerse()
// Results: displays the horse verse of Old MacDonald
{
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl;
    cout << "And on that farm he had a horse, E-I-E-I-O." << endl;
    cout << "With a neigh-neigh here, and a neigh-neigh there," << endl;
    cout << " here a neigh, there a neigh, everywhere a neigh-neigh." << endl;
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl << endl;
}
```

DISADVANTAGES?

Old MacDonald (cont.)

DISADVANTAGES of version 2

- as add new verses, program still gets longer and longer
- lots of redundancy (each verse is similar, must maintain consistency)

could generalize all verses using parameters

- verses differ only in animal and noise
- add two parameters for animal and noise, when call must specify argument

```
void Verse(string animal, string noise)
// Assumes: animal is an animal name, and noise is the noise it makes
// Results: displays the corresponding OldMacDonald verse
{
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl;
    cout << "And on that farm he had a " << animal << ", E-I-E-I-O." << endl;
    cout << "With a " << noise << "-" << noise << " here, and a "
        << noise << "-" << noise << " there," << endl;
    cout << " here a " << noise << ", there a " << noise << ", everywhere a "
        << noise << "-" << noise << "." << endl;
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl << endl;
}

. . .

Verse("cow", "moo");
```

version 3

combined all
verses into one
generalized
function

to add a new
verse, simply
make a new call
with the desired
argument

```
#include <iostream>
#include <string>
using namespace std;

void Verse(string animal, string noise);
void HorseVerse();

int main()
{
    Verse("cow", "moo");
    Verse("horse", "neigh");
    Verse("duck", "quack");

    return 0;
}

////////////////////////////////////

void Verse(string animal, string noise)
// Assumes: animal is an animal name, and noise is the noise it makes
// Results: displays the corresponding OldMacDonald verse
{
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl;
    cout << "And on that farm he had a " << animal << ", E-I-E-I-O." << endl;
    cout << "With a " << noise << "-" << noise << " here, and a "
        << noise << "-" << noise << " there," << endl;
    cout << " here a " << noise << ", there a " << noise << ", everywhere a "
        << noise << "-" << noise << "." << endl;
    cout << "Old MacDonald had a farm, E-I-E-I-O." << endl << endl;
}
```

QUESTION: what would `Verse("moo", "cow")` do?

function summary

C++ functions provide units of computational abstraction

- can define and test a function once, then reuse over and over
- don't need to remember the details, just need to know how to call it

function definitions can

- make `main` cleaner and clearer
- simplify repeated tasks (define once, call many times)
- generalize similar tasks using parameters (call function with diff. arguments)

can place useful functions in separate files, `#include` as needed

- later in course, we will see how to separately compile files
- more efficient since it avoids recompilation