```
//===========================
// Section 11 Functions
//===========================

// What is a function
C++ programs
- stuff in the C++ standard libraries
- Third party libraries
- our own functions and classes

Functions allow us to modularize things
- Separate code into logical self-contained units
- These units can be reused

Boss\worker analogy - The things you need to understand:
- Write the code to the function specifications
- Understand what it does
- Understand what it needs
- Understand what it sends back
- Understand any errors
- Understand any performance constraints

Don't worry about how it works internally unless you wrote it

This is called 'information hiding';

// Simple examples
#include <cmath>
sqrt(400);
pow(2.0, 3.0);

- we don't need to know how this works. We just get to use it

int addNumbers(int a, int b){
    return a + b;
}

C++ standard library header files - documentation for librarie

// <cmath>
ceil()
floor()
round()
pow()
sqrt()

// <cstdlib>
rand()  //pseudo-random number
srand() // seeds pseudo-random number
```

```cpp
//===========================
// Random number generator example
//===========================
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>

int main(){
    int randomNumner{};
    size_t count{ 10 }; // number of random numbers to generate
    int min{ 1 };       // lower bound (inclusive)
    int max{ 6 };       // upper bound (inclusive)

    // seed the random number generator
    // if you don't seed it you will get the same sequence every time
    std::cout << "RAND_MAX_ on my system is : " << RAND_MAX << std::endl;
    srand(time(nullptr));

    for (size_t i{ 1 }; i <= count; i++) {
        randomNumner = rand() % max + min; // generate a random number[min, max]
        std::cout << randomNumner << std::endl;
    }

    return 0;
}
//===========================
// Random number generator example
//===========================

// Function definitions
name
- the name of the function
- same rules are for variables
- should be meaningful
- usually a verb or verb phrase

parameter list
- variables passed into the function
- types must be specified

return type
- the type of data that is returned from this function

body
- the statements that are executed when the function is called
- contained within {}

int functionName(){
    // statements
    return 0;
}

int functionName(int a){
    // statements
    return 0;
}

void functionName(){
    // returns no data, but just does something
    return; // optional
}

void functionName(int a, std::String b){
    // arguments must come in the same order as listed
}
```

```
functions can call other functions
compiler must know about the function first

// Function prototypes
Define functions before caling them
- ok for small programs
- not practical for larger ones

Use function prototypes
- tells the compuler about the function
- also called forward declarations
- placed at the beginning of the program
- also used in header files (.h)

// Example
int functionName(int a); // This is the prototype - the name of the variable coming in is optional


int functionName(int a){
    //stuff
}

Make sure your function calls match the prototype

// Function parameters and the return statement
When we call a function we can pass data in
They are called arguments
in the definition they are parameters
they must match in number, order and type

NOTE: If the compiler knows how to convert one type to another it will try to do so
be careful with this

// Pass by value
This is the default
It means a copy of the data is passed into the function
Whatever changes you make to the parameter does not affect the original object

formal vs actual
Formal Parameters: the parameters defined in the function header
Actual parameters: the parameter used in the function call; the arguments.

// Example of formal\actual parameters
void ParamTest(int formal){     // Copy of the actual parameters
    cout << formal << endl;     // 50
    formal = 100;               // Only changes the local copy
    cout << formal << endl;     // 100
}

int main){
    int actual{50};
    cout << actual << endl;     // 50
    paramTest(actual);          // Pass in 50 to param test
    cout << actual << endl;     // 50 - did not change
    return 0;
}

// Return statement
If a function returns a value then it must use a return statement
If it does not return a value (void) then the return statement is optional
It can occur anywhere in the function body
It immdiately exist the function
A function can have multiple return statements
- try to avoid having a whole lot
The return value is the result of the function call
```

```
// Default argument values
When a function s called, arguments must be supplied
Soemtimes some of the arguments have the same values most of the time
We can tell the compiler to use default values if the arguments are not supplied
The default values can be in the prototype or the definition, but NOT both
- best practice is to put in the prototype
- Must appear at the tail end of the parameter list

Can have multiple default values
These must appear consecutively at the tail end of the parameter list

// Example:
double calcCost(double baseCost, double taxRate = 0.06, double shipping = 3.50); // Prototype with a default value

double calcCost(double baseCost, double taxRate, double shipping){
    return basecost += (baseCost * taxRate);
}

int main(){
    double cost{0}
    cost = calcCost(200.00, 0.08, 4.25);        // Will not use any defaults
    cost = calcCost(100.0, 0.08 );              // Will not use the tax default and wll use default shipping
    cost = calcCost(200.00);                    // Will use all defaults
    return 0;
}

// Overloading functions
Different parameter lists but same function name
abstraction mechanism so we can just think of the function we want
It's a type of polymorphism
same name work with different data types to execute similar behavior
Compiler must be able to tell the functions apart based on the parameter list and arguments supplied

int addNumbers(int a, int b);
double addNumbers(double a, double b);

NOTE: You must implement all the functions you prototype
One restriction: The return type is not considered when the compiler decides which one to call
Compiler will not guess which one we want

NOTE: Characters are promoted to integers if used as an argument that gets sent into a function that requires an int
the same thing will happen if we happen to send a float into a double - function(12.4F) will be promoted to a double
Compiler will also convert a C-style string to a C+ string object

this is one case where the compiler does try to guess what you actually wanted

// Passing arrays to functions
We can pass one in by including the [] in the formal parameter description
void printArray(int numbers[], size_t size); // pass in the size of the array as well

The array elements are NOT copied
The array name evaluates to the location of the array in memory, this address is what is copied

The function has no idea how many elements are in the array since all it knows is the location of the first element
We have to also pass in the size
NOTE: The function can modify the actual array

// Const parameters
we can tell the compiler that function parameters are read only
void printArray(const int numbers[], size_t size){} // Now the compiler won't let us do any changes
```

```
// Pass by reference
Sometimes we want to change the actual parameter from within the function body
We need the actual location
We use reference parameters to tell the compiler to pass in a reference to the actual parameter
The formal parameter is now an alias for the actual parameter

// Example
void scaleNumber(int &num); // prototype

void scaleNumber(int &num){ // function also gets the &num notation
    if (num > 100){
        num = 100;
    }
}

int main(){
    int number{1000};
    scaleNumber(number); // pass in the actual variable, don't use the & here
    cout <<  number << endl;
}

NOTE: If you are putting things into a function that isn't meant to change data, make it a const reference

void printSomething(const vector<string> &v); // const reference - now we can't accidentally change something

// Scope rules
Determines where an identifier can e used
uses static or lexical scoping (the same way you read a program)
local or block scope
Global scope

Local or block scope:
- identifiers in a {}
- function parameters have block scope
- Only visible in the block it was declared
- Function local variables are only active while the function is executing
- Local variables are NOT preserved between function calls
- With nested blocks, inner blocks can see out but outer blocks cannot see in

// Static local variables
Its lifetime is the lifetime of the program
Declared with the static qualifier

static int value{10};

Only initialized the first time the function is called
Still only visible to the function it's inside

// Global scope
Identifier declared outside any function or class
Visible to all parts of the program after it's been declared
Global constants are ok
Best practice - don't use global variables

// How do function calls work?
Functions use the 'function call stack'
- Analogous to a stack of books
- Last in, First Out. You cannot jump into the middle of the stack
- Push and pop into and off the stack

Stack frame or activation record
- Functions must return control to the function that called it
- Each time a function is called we create a new activation record and push it on the stack
- When a function terminates we pop the activation record and return
- Local varialbes and function parameters are allocated on the stack
```

```

NOTE: Stack size is finite - we can (and sometimes do) overflow it


        Memory
//==========================//
//      Heap                //
//      Free Store          //
//                          //
//                          //
//==========================//
//              ^           //
//              |           //
//  Stack - push and pop    //
//      |                   //
//     \ /                  //
//==========================//
//  Static variables        //
//==========================//
//                          //
//      Code Area           //
//                          //
//==========================//

// Inline functions
Function calls have a certain amount of overhead
Some simple functions can be compiled inline
- avoids that overhead
generates inline assembly code
faster
but could cause code bloat

compiler optiizations are very sophisticated
It will probably inline what it can without your suggestion

// To tell the compiler
inline int addNumbers(int a, int b){
    // definition
}

// Recursive functions
This is a function that calls itself either directly or indirectly through another function

Recursive problem solving:
You need a base case to test for
Divide the rest of the problem into subproblems and do recursive calls

There are many problems we can solve this way - like factorials, fibonacci, fractals

Searching and sorting through binary searches, seach trees. towers of hanoi

// Example - factorial
Base case: factorial(0) = 1 // The recusion will stop when it hits this
Recusive case: factorial(n)=n * factorial(n-1) // As long as there is work to do, the function will continue

unsigned long long factorial(unsigned long long n){
    if (n == 0){
        return 1; // the base case - when we have hit 0 we can get out of this recursion
    } else {
        return n * factorial(n-1); // This calls the function again passing n back in
    }
}

int main(){
    cout << factorial(8) << endl;
}

```

```
// Fibonacci numbers (definition)
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n - 2)

Two base cases in this example
fib(0) = 0
fib(1) = 1

Recursive case
fib(n) = fib(n-1) + fib(n - 2)

unsigned long long fibonacci(unsigned long long n){
    if (n <= 1 ) {
        return n; // Handles the base cases
    } else {
        return fibonacci(n-1) + fibonacci(n -2) // recursive calls
    }
}


int main(){
    cout << fibonacci(30) << endl;
}

Important notes
If recursion doesn't stop you will have infinite recursion - always have a base case!
Recursion can be resource intensive
Only use them when it makes sense
Anything that can be done with recursion can also be done with iteration

We go up one side of the stack, building function calls with the data until we hit the base case
When we hit the base case, we start returning the values calculated by those functions to each
previous caller

// Version the first time I did this
//===========================
#include <iostream>
#include <iomanip>
using namespace std;

int function_activation_count{ 0 };

double a_penny_doubled_everyday(int numberOfDays, double startingAmount = 0.01);

void amount_accumulated() {

    double total_amount{ a_penny_doubled_everyday(25) };
    cout << "If I start with a penny and doubled it every day for 25 days, I will have $" << setprecision(10) << total_amount;
}
-

double a_penny_doubled_everyday(int numberOfDays, double startingAmount) {
    function_activation_count++;

    if (numberOfDays == 1) {
        return startingAmount;
    } else {
        return a_penny_doubled_everyday((numberOfDays - 1), (startingAmount + startingAmount));
    }
}

int test_function_activation_count() {
    return function_activation_count;
}
```

```cpp
// Version the second time I did this
//===========================
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>

double aPennyDoubledEveryDay(int numberOfDays, double startingAmount = 0.01);

int main(){

    // Return the total amount accumulated if a penny is doubled every day
    // penny += penny;
    std::cout << aPennyDoubledEveryDay(18, 0.15) << std::endl; //  go with the defaults for 5 days
    return 0;
}

double aPennyDoubledEveryDay(int numberOfDays, double startingAmount) {
    if (numberOfDays == 1) {
        return startingAmount;
    } else {
        startingAmount += startingAmount;
        return aPennyDoubledEveryDay(numberOfDays - 1, startingAmount);
    }
}

//===========================
// Section 11 Challenge - refactor into functions
//===========================
#include <iostream>
#include <vector>
#include <string>

// Prototypes
void clearScreen();
void printMenu();
char getSelection();
void printNoNumbersInList();
void printAllNumbersInList(const std::vector<double>& numbersList);
void addANumberToTheVector(std::vector<double>& numbersList);
void calculateMeanOfNumbers(const std::vector<double>& numbersList);
void calculateSmallestNumber(const std::vector<double>& numbersList);
void calculateLargestNumber(const std::vector<double>& numbersList);
void quitProgram();
void invalidInput();

int main() {

    char selection{};
    std::vector<double> numbersList{};
    do {
        printMenu();
        selection = getSelection();
        if (selection == 'P') {
            if (numbersList.size() == 0) {
                clearScreen();
                printNoNumbersInList();
            } else {
                clearScreen();
                printAllNumbersInList(numbersList);
            }
        } else if (selection == 'A') {
            addANumberToTheVector(numbersList);
            clearScreen();
        } else if (selection == 'M') {
            calculateMeanOfNumbers(numbersList);
        } else if (selection == 'S') {
            calculateSmallestNumber(numbersList);
        } else if (selection == 'L') {
            calculateLargestNumber(numbersList);
        } else if (selection == 'Q') {
            quitProgram();
        } else {
            invalidInput();
        }
    } while (selection != 'Q');
    return 0;
}

void clearScreen() {
    // This translates to a code that clears the console
    std::cout << "\033[2J\033[1;1H";
}

void printMenu() {
    std::cout << "Please make a selection: " << std::endl;
    std::cout << "P - Print numbers" << std::endl;
    std::cout << "A - Add a number" << std::endl;
    std::cout << "M - Display the mean of the numbers" << std::endl;
    std::cout << "S - Display the smallest" << std::endl;
    std::cout << "L - Display the largest" << std::endl;
    std::cout << "Q - Quit" << std::endl;
}

char getSelection() {
    char userInput{};
    std::cin >> userInput;
    return toupper(userInput);
}

void printNoNumbersInList() {
    std::cout << "There are no numbers in the list. " << std::endl;
    std::cout << "==================================" << std::endl;
}

void printAllNumbersInList(const std::vector<double>& numbersList) {
    for (auto item : numbersList) {
        std::cout << item << " ";
    }
    std::cout << std::endl;
    std::cout << "==================================" << std::endl;
}

void addANumberToTheVector(std::vector<double>& numbersList) {
    int numberBuffer{};
    std::cout << "Enter the number to add to the vector: ";
    std::cin >> numberBuffer; // Circle back to this to deal with input validation
    numbersList.push_back(numberBuffer);
}

void calculateMeanOfNumbers(const std::vector<double>& numbersList) {
    double average{};
    for (auto item : numbersList) {
        average += item;
    }
    average /= numbersList.size();
    clearScreen();
    std::cout << "The average is: " << average << std::endl;
    std::cout << "================" << std::endl;
}

void calculateSmallestNumber(const std::vector<double>& numbersList) {
    double swap{ numbersList[0] };
    for (auto item : numbersList) {
        if (swap > item) {
            swap = item;
```

```cpp
        }
    }
    clearScreen();
    std::cout << "The smallest number is: " << swap << std::endl;
    std::cout << "================" << std::endl;
}

void calculateLargestNumber(const std::vector<double>& numbersList) {
    double swap{ numbersList[0] };
    for (auto item : numbersList) {
        if (swap < item) {
            swap = item;
        }
    }
    clearScreen();
    std::cout << "The largest number is: " << swap << std::endl;
    std::cout << "================" << std::endl;
}

void quitProgram() {
    clearScreen();
    std::cout << "Thanks for using the program." << std::endl;
}

void invalidInput() {
    clearScreen();
    std::cout << "That is not a valid selection, please try again." << std::endl;
}
```