Getting Ready for Canadian Computing Competition

Class 2
December 8, 2019

Coding Question 1

```
if (slipped on the drive walk) {
    if (ice on the drive walk) {
        twisted your ankle
    }
    else {
        nothing happens
    }
}
if (twisted your ankle) {
    if (you have a dog) {
        nothing happens
    }
    else if (your neighbour has a dog) {
        bitten by a dog
    }
    else {
        nothing happens
    }
}
if (bitten by a dog) {
    seriously injured
}
```

Coding Question 1 Solution

```
// solution 1
   if (slipped on the drive walk && ice on the drive walk && your neighbour has a
dog) {
          seriously injured
   }
   else if (!slipped on the drive walk) {
   }
   else {
      nothing happens
   }
```

Coding Question 1 Solution

```
// solution 2
  if (slipped on the drive walk) {
    if (ice on the drive walk && your neighbour has a dog) {
      seriously injured
    }
    else {
      nothing happens
    }
  }
```

Outline – Part 1

- Control flow statements
  - If .. Else
  - Ternary operator
  - Switch statement
  - While, do...while loops
  - For loop
  - Break, continue, counter, switch
- Arrays
  - Declaration
  - Initialization
  - Iterating through arrays
  - Reading an array from std::in
  - Multidimensional arrays
  - Static vs dynamic arrays (next class)

If ... Else

- Syntax:

```
if (Boolean_Expression_1) {
  Action1();
} else if (Boolean_Expression_2) {
  Action2();
} else {
  Action3();
}
```

- Braces can be omitted if there is only one statement (not recommended)
- “else” is optional
- Statements can be nested

Ternary Operator

- Syntax:

```
var = (Boolean_Expression) ? True_Case : False_Case;
```

- Equivalent to :

```
if (boolean_Expression) {
    True_case;
} else {
    False_case;
}
```

Example:
Declare an integer, maxVal, to be the max of two given integers, n1 and n2.

Ternary Operator

- Syntax:

```
var = (Boolean_Expression) ? True_Case : False_Case;
```

- Equivalent to :

```
 if (boolean_Expression) {
        True_case;
    } else {
        False_case;
    }
```

Example:
Declare an integer, maxVal, to be the max of two given integers, n1 and n2.

Solution:

```
int maxVal = (n1 > n2) ? n1 : n2;
```

Switch Statements

- Choice of branch is determined by a controlling expression that is one of: Bool, int, char, enum

- Syntax:

```
switch (Controlling_Expression) {
  case Constant_1:
    Statement_Sequence_1
    break;
  case Constant_2:
    Statement_Sequence_2
    break;
  default:  // default branch if other branches return false
    Default_Statement_Sequence
}
```

Switch Statements – continued …

Example:

Given a character in the table on the right, print its ASCII number. If the character does not exist in the table, print 0. Use a switch statement.

Character	ASCII
A	65
=	104
!	33
a	97

While Loop and Do-While Loop

- Syntax:

```
while (Boolean_Expression) {  // Loops Condition
  Statements
}


do {
  Statements
} while (Boolean_Expression);  // Loop Condition
```

For Loop

- Syntax:

```
for (Initialization_Action; Boolean_Expression; Update_Action){
  Statements
}
```

-
- `break` - forces the loop to exit immediately
- `continue` - skips rest of loop body

Example – while vs for loop

```
// printing from 0 to 9 using a while loop
   int i = 0;
   while (i < 10) {
       cout << i << endl;
       i++;
   }
```

Example – while vs for loop continued…

```
// printing from 0 to 9 using a while loop
   int i = 0; // initialization
   while (i < 10) {
       cout << i << endl;
       i++; // update
   }
```

Example – while vs for loop continued…

```
// printing from 0 to 9 using a while loop
   for (int i = 0; i < 10; i++) {
       cout << i << endl;
   }
```

Example 1 – Using break and continue

- What does the following print?

```
for (int i = 0; i < 10; i++) {
        if (i == 3) continue; // use of continue
        cout << i << endl;
    }
```

Example 1 – Using break and continue

- What does the following print?

```
for (int i = 0; i < 10; i++) {
        if (i == 3) continue; // use of continue
        cout << i << endl;
    }
```

Output:

0
1
2
4
5
6
7
8
9

Example 2 – Using break and continue

- What does the following print?

```
for (int i = 0; i < 10; i++) {
        switch (i){
            case 0:
                cout << "0!" << endl;
            default:
                break;

        }
    }
```

Example 2 – Using break and continue

- What does the following print?

```
for (int i = 0; i < 10; i++) {
        switch (i){
            case 0:
                cout << "0!" << endl; // missing break here
            default:
                break; // default does not need a break
// adding a break does not make a difference though, it only breaks out of the switch, not the for loop
        }
    }
```

Output:
0!

Example 3 – Using break and continue

- What does the following print?

```
for (int i = 0; i < 10; i++) {
        if (i == 3) continue;
        if (i == 3) break;
        if (i == 5) break;
        cout << i << endl;
    }
```

Example 3 – Using break and continue

- What does the following print?

```
for (int i = 0; i < 10; i++) {
        if (i == 3) continue;
        if (i == 3) break; // continue skips this line
        if (i == 5) break;
        cout << i << endl;
    }
```

Output:

0

1

2

4

Example 4

Write a program in C++ to find the perfect numbers between 1 to n.

A perfect number is a positive integer equal to the sum of its positive divisors, excluding the number itself. For example, 6 has divisors 1, 2 and 3 with 1 + 2 + 3 = 6, so 6 is a perfect number.

- Sample input:

500

- Sample output:

6
28
496

- Hints: nested loops, check divisibility using modulus division

Example 5 – Using a switch

Write a program in C++ to find prime number within a range. The input will contain 2 numbers, the lower bound and the upper bound, separated by a space. The output will contain all prime numbers in the range, one on each line.

Sample Input:

1 20

Sample Output:

2
3
5
7
11
13
17
19

Example 6 – Using a counter

Given a natural number, count the number of digits it has. The number is no more than 200000000, so using an int to store it will suffice.

Sample input:

123456

Sample output:

6

Practice 1

Create a checkerboard pattern with the words "black" and "white", given the dimension of the board as a natural number. For example, 3 means a 3 by 3 board.

Sample input:

3

Sample Output:

black-white-black
white-black-white
black-white-black

Practice 2

Modify the code above so that when given an nxn board with a few colours, print a number representing how many grids were wrong. The input will no longer contain ‘-‘ and will have one space separating the colours.

Sample input:

3
black black black
white black white
black white orange

Sample output:

2

Explanation: the first row should have “black white black”. Also, the last row should be “black” instead of “orange”. So there were a total of two mistakes.

Practice 3

Write a program that will print out all the integers it reads in. However, as soon as a string is entered, it terminates.

Review of I/O

Arrays

Arrays

- An array is a collection of data of the same type.

- To declare an array of size n:

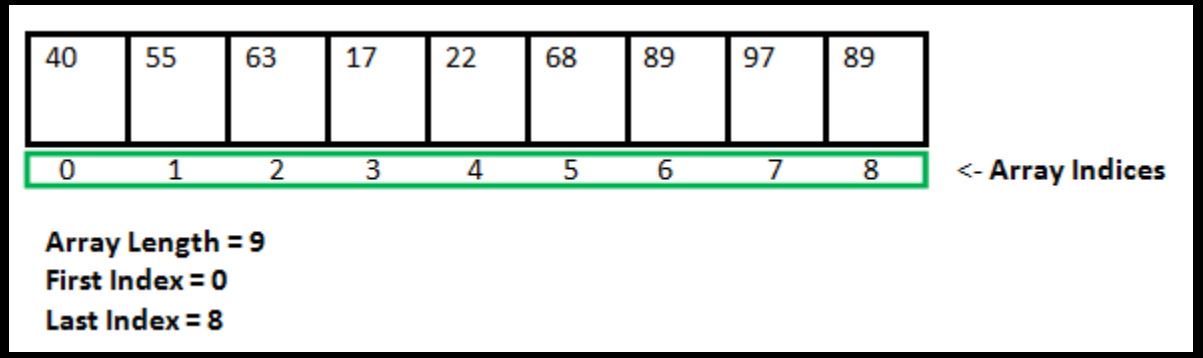
```
DataType arrayName[n];
```

```
ex). int myIntArray[10];
```

- Uninitialized arrays will have garbage values in them
  - If you write cout << myIntArray[0], it will print random numbers

Initializing Arrays

- The indices span from 0 to n – 1



- You can initialize an array:

```
int foo1 [3] = {16, 2, 77};

int foo2 [3] = {16, 2};  // {16, 2, 0};

int foo3 [3] = {};   // {0, 0, 0}
```

  - If fewer values than the array size is provided, the array is filled from the beginning. The rest are filled with the "zero" of the base array type

Array Base Type	"Zero" Value
Int	0
Bool	0 (false)
String	"" (empty)

Initializing Arrays Continued...

- If the array size is not specified, the size is determined by the number of initialization values.

Ex). `int foo5[] = {16, 2, 77};`

Accessing Array Elements

- To access the elements of the array, we use the [] operator with its index inside
  - The indices span from 0 to n – 1

Ex).

```
int foo1 [3] = {16, 2, 77};

foo1[0] = 16;

foo1[1] = 2;

foo1[2] = 77;

Foo1[3] = ??? // garbage value, unpredictable
```

Obtaining the Array Size

```
sizeof (x) // tells you the size of x in bytes

sizeof (arr) / sizeof (arr[0]) // gets array size
```

- By definition, arrays must contain data of the same type
- Size of entire array / size of first element
- You will not need this because you knew the size when you initialized it

Iterating through Arrays

- We use for loops to iterate from 0 to size – 1

```
for (int i = 0; i < size; i++) {…}
```

- You can only initialize it once:

```
int arr[3] = {4, 5, 6};
arr = {0, 1, 2}; // ERROR
arr [0] = 0;  // correct
arr [1]  1;  // correct
```

Iterating through Arrays Continued...

- Alternatively

```
for (int i = 0; i < 3; i++) {
        arr[i] = i;
}
```

Example 1

Given an integer n followed by n integers, read n integers into an array of length n.
(Almost every single CCC that involves a list of numbers will require this function)

```
int n = ***; // hidden implementation
```

Example 1

Given an integer n followed by n integers, read n integers into an array of length n.
(Almost every single CCC that involves a list of numbers will require this function)

Solution:

```
int n = ***; // hidden implementation
int arr[n];
int input;
for (int i = 0; i < n; i++) {
      cin >> input;
      arr[i] = input;
}
```

Example 2

Initialize an int array of size n to contain all 0's. The value of n has been defined.

Example 2

Initialize an int array of size n to contain all 0's. The value of n has been defined.

Solution:

```
int n = ...; // hidden
int arr[n];
for (int i = 0; i < n; i++) {
    arr[i] = 0;
}
```

Practice

Read in an array of size 10, print every other element in the array separated by spaces, where the first begins at index 1.

Sample input:

0 1 2 3 4 5 6 7 8 9

Sample Output:

1 3 5 7 9

Multi-Dimensional Arrays

```
int arr_2D [2][2] = {{1, 2}, {3, 4}};
```

- Accessing Elements:
  - Getting the number 3: arr_2D[1][0];

Exercise

```
int arr_3D [2][2][2] = {{{1}, {2, 3}}, {{4, 5}, {6}}};
```

Print each element using loops. What is the output?

Exercise

```
int arr_3D [2][2][2] = {{{1}, {2, 3}}, {{4, 5}, {6}}};
```

Print each element using loops. What is the output?

Use 3 for loops …

Output:

1
0
2
3
4
5
6
0

Practice 1 (hw)

Given an array of integers, a number n, and an integer sum, print “true” if there are at least n pairs of integers in the array that add up to the given sum. Print “false” otherwise.

Practice 2

Reverse an array by swapping its elements.

Outline – Part 2

- Vectors
  - Static vs dynamic arrays
  - 2D vectors
  - Vector operations
  - LIFO
  - Vector operations, erase(), iterators
- Functions
  - Function declarations
  - Functions calls
  - Passing by value
  - Scopes
  - Function overloading
  - Default values(if time permits)

Vectors

Vectors

- The arrays from last class are **static arrays**
  - They have a fixed size
- **Dynamic arrays** can grow and shrink in size
- Vectors have a base type and a collection of base type values

Vectors

- Declaration Syntax:

```
#include <vector> // at top of the file, use C++14

vector<Base_Type> vecName;


Ex). Vector of ints: vector <int> myIntVect;

Initialize the vector: vector <int> myIntVect = {1,2,3};
```

n-Dimensional Vectors

- Declaration:

```
vector <vector<int>> my2dVect = {1,2,3};
```

- Adding a row:

```
my2dVect.emplace_back (vector<int>()); // note the parenthesis
```

Vector Operations

- vec[pos] – gets the element at index pos
  - if pos is out of range, it will cause a run time error
  - Same as an array

- vec.push_back(x) or vec.emplace_back(x) for C++14 and later
  - x must be the base type, otherwise it will cause a compilation error
  - adds x to the end of vec

Vector Operations

- vec.back() – gets the last element in the vector
- vec.front() – gest the front of the vect, same as vec[0]
- Vec.pop_back() – removes the last element in the vector

Example

What will the following produce?

```
vector <int> board;

board.emplace_back(1);
board.emplace_back(3);
board.emplace_back(5);

while (!board.empty()) {
    cout << board.back() << endl;
    board.pop_back();
    cout << board.front() << endl;
}
```

Example

What will the following produce?

```
vector <int> board;

board.emplace_back(1);
board.emplace_back(3);
board.emplace_back(5);

while (!board.empty()) {
    cout << board.back() << endl;
    board.pop_back();
    cout << board.front() << endl;
}
```

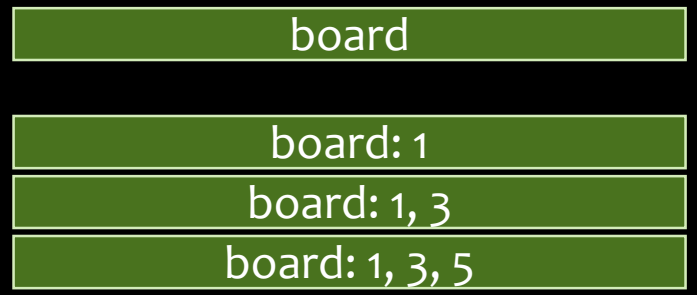
Example

What will the following produce?

```
vector <int> board;

board.emplace_back(1);
board.emplace_back(3);
board.emplace_back(5);

while (!board.empty()) {
    cout << board.back() << endl;
    board.pop_back();
    cout << board.front() << endl;
}
```



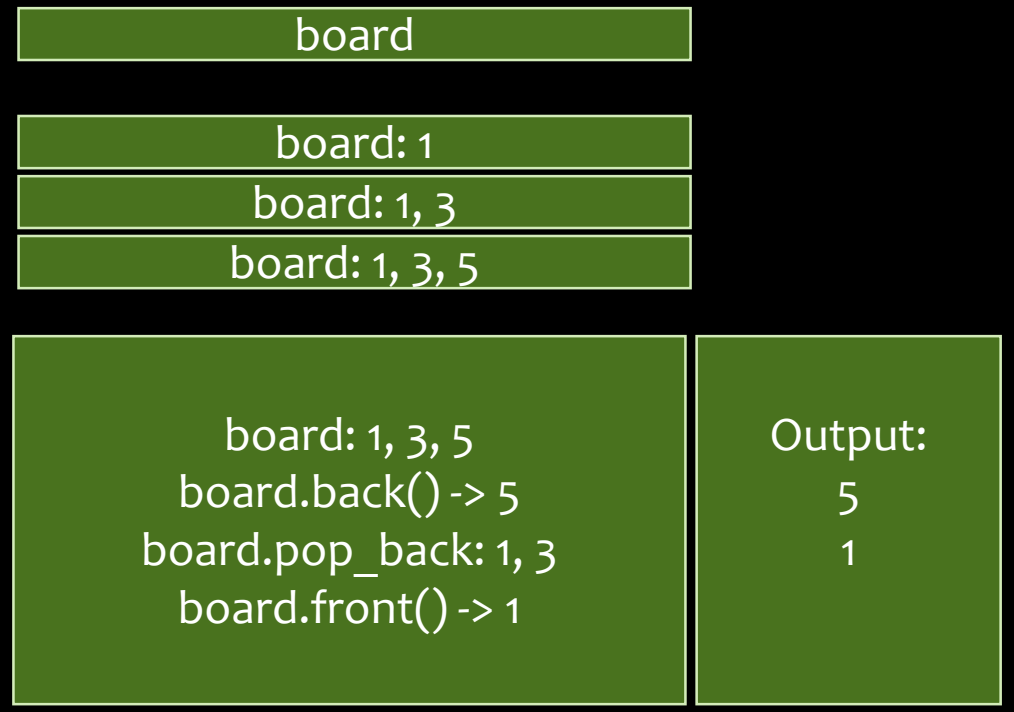
Example

What will the following produce?

```
vector <int> board;

board.emplace_back(1);
board.emplace_back(3);
board.emplace_back(5);

while (!board.empty()) {
    cout << board.back() << endl;
    board.pop_back();
    cout << board.front() << endl;
}
```



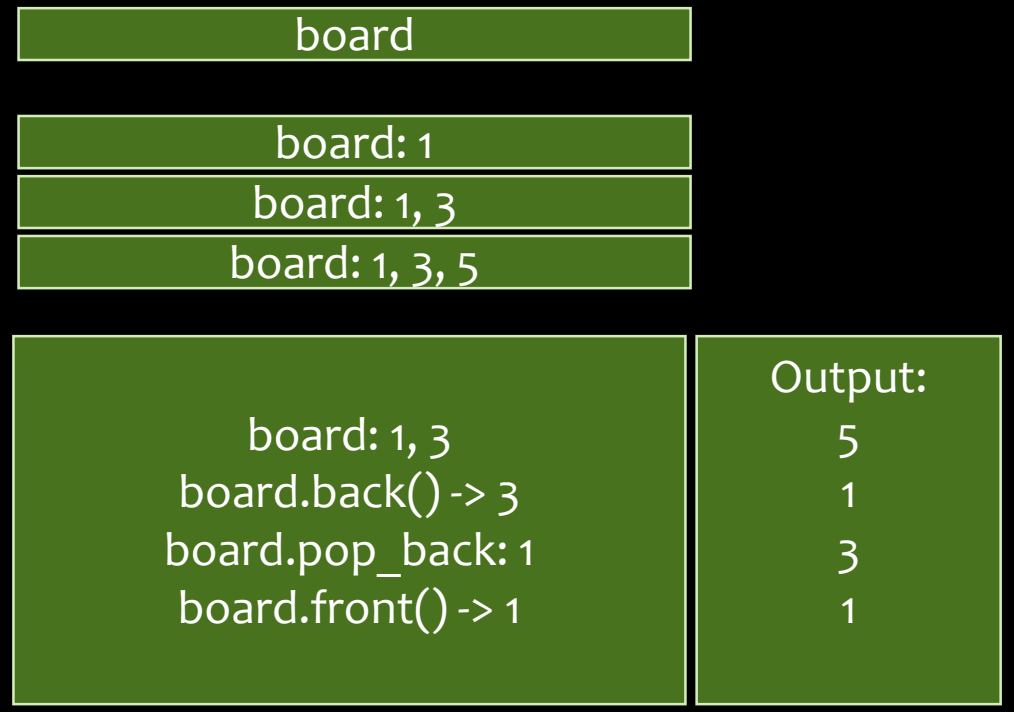
Example

What will the following produce?

```
vector <int> board;

board.emplace_back(1);
board.emplace_back(3);
board.emplace_back(5);

while (!board.empty()) {
    cout << board.back() << endl;
    board.pop_back();
    cout << board.front() << endl;
}
```



Example

What will the following produce?

```
vector <int> board;

board.emplace_back(1);
board.emplace_back(3);
board.emplace_back(5);

while (!board.empty()) {
    cout << board.back() << endl;
    board.pop_back();
    cout << board.front() << endl;
}
```

board

board: 1

board: 1, 3

board: 1, 3, 5

board: 1
board.back() -> 1
board.pop_back: empty
board.front() -> 1

Output:
5
1
3
1
1
1

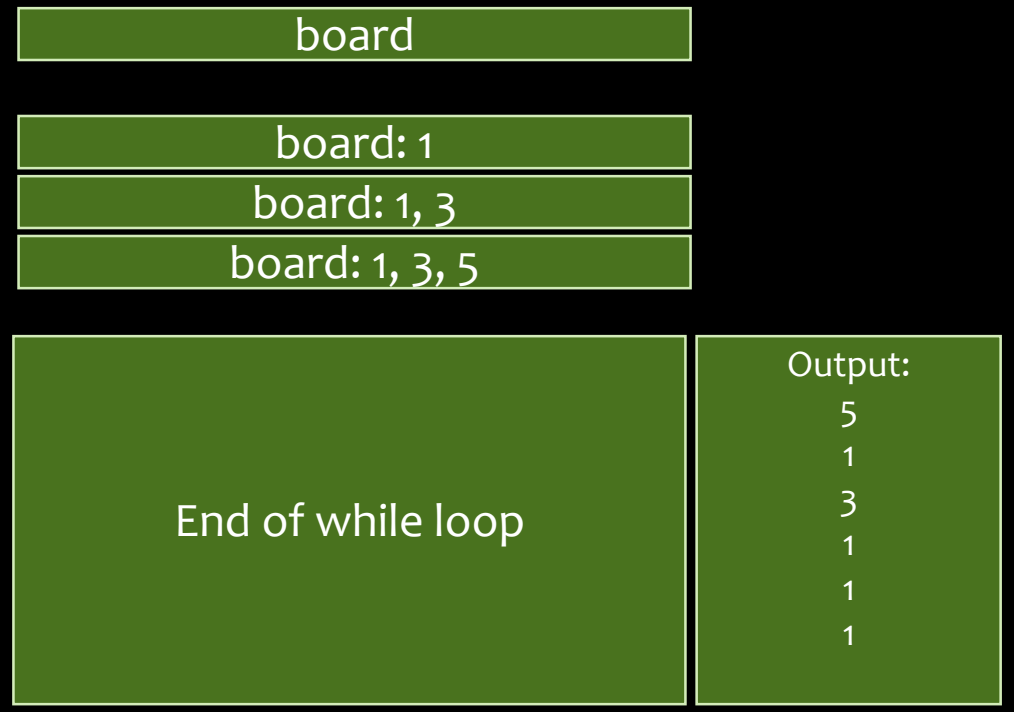
Example

What will the following produce?

```
vector <int> board;

board.emplace_back(1);
board.emplace_back(3);
board.emplace_back(5);

while (!board.empty()) {
    cout << board.back() << endl;
    board.pop_back();
    cout << board.front() << endl;
}
```



LIFO – Last in First Out

- Notice how in the last example, we had
board.emplace_back(1);
board.emplace_back(3);
board.emplace_back(5);
- When we used board.back() and board.pop_back() to get 5,3,1
- This is LIFO, last in first out. For example, 5 was placed in last but it was printed first
- *We can actually use a stack instead of a vector for this question, but we'll talk about that later

Exercise

- Given a string of length n, n is not 0, determine if n is a palindrome. A palindrome is a word, that reads the same backward as forward (ex. madam, level, kayak, racecar)

Sample Input 1:

7

racecar

Sample output 1:

true

Sample Input 2:

7

aacecar

Sample output 2:

false

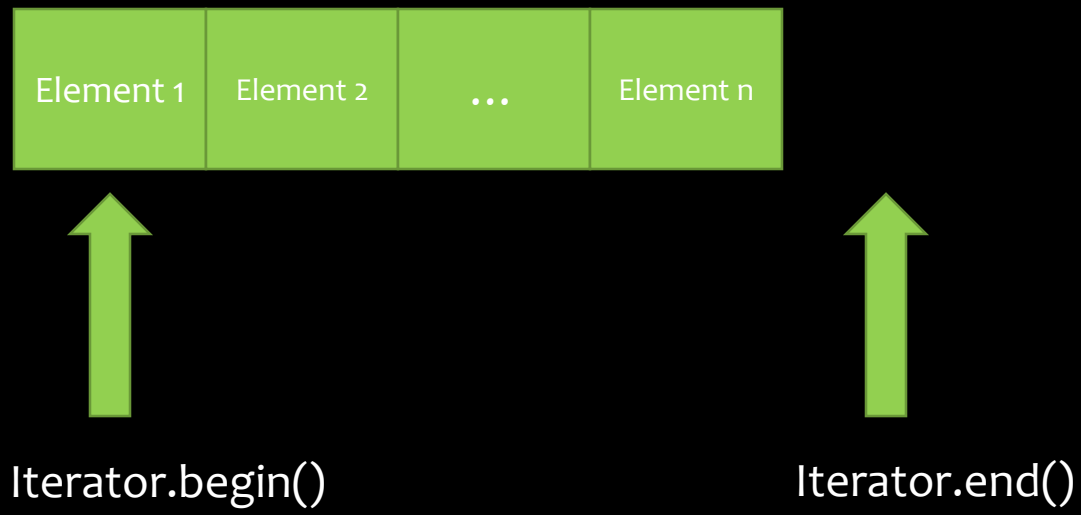
* don’t forget to consider edge cases like n = 1

More Vector Operations

- vec.size() - returns number of elements in vec
- vec.clear() - removes all elements from vector and leave the vector with a size of 0
- vec.erase(startPos, endPos) - removes element from startPos (inclusive) to endPos (exclusive)
  - must use iterator

Vector Iterators

- vec.begin() – iterator for beginning of vector (like a pointer)
- vec.end() – iterator for end of vector (like a pointer)
  - only use iterator for vec.erase(start, end) for now



Example 1

This will remove every element in myIntVect from index 1 to the end

```
vector <int> myIntVect = {1, 2, 3, 4, 5};

myIntVect.erase(myIntVect.begin() + 1, myIntVect.end());


// new myIntVect is {1}
```

Example 2

This will remove the second element

```
vector <int> myIntVect = {1, 2, 3, 4, 5};

myIntVect.erase(myIntVect.begin() + 1, myIntVect.begin() + 2);


// new myIntVect is {1, 3, 4, 5}
```

Exercise

Given a vector of an unknown size, remove all elements from the beginning to the middle. If there are an odd number of elements, keep the middle.

For example, if there were 5 elements, remove elements at index 0 and 1. If there were 6, remove elements at index 0, 1, 2.

Practice – 2D String Vectors

Create a 2d vector of size nxn storing a chess board. Each cell is black. Use “b” to denote black. Then print the board.

Sample Input:

3

Sample output:

bbb

bbb

bbb

Practice – 2D String Vectors

Correct the board to contain alternating “b” and “w”

Sample Input:

5

Sample Output:

bwbwb

wbwbw

bwbwb

wbwbw

bwbwb

Functions

Functions

- A block of code that runs when it is called
- Can be reused
- You can pass parameters into a function
- Functions can return values
- main() is a pre-defined function that is executed without being manually called
- Main returns 0 if execution is successful (they are omitted in the slides to save space, but it is highly recommended that you had them in your own programs)
- All functions must be declared before the main

Function Declaration

returnType *functionName*(arg1Type arg1Name, … , argnType argnName) {
// function definition

return … (type must match returnType)
}

Ex). This function gets an int and returns the int

```
int foo (int n) {

    return n;

}
```

Function Declaration

- If a function return nothing, we call it a void function:

```
void functionName(arg1Type arg1Name, …, argnType argnName ) {
          // function definition, no return statement
}
```

- A function can also have no arguments:

```
void functionName( ) {
          // function definition, no return statement
}
```

Function Calling

- Syntax

```
functionName (arguments);
```

- The number of **arguments** supplied must equal the number of **parameters** of the function

```
void myFunction(int n) {
  cout << "you passed me " << n << endl;
}

int main() {
  myFunction(5);

  return 0;
}
```

prints: you passed me 5

Function Calling

- Syntax

```
functionName (arguments);
```

- The number of **arguments** supplied must equal the number of **parameters** of the function

```
void myFunction(int n) {
  cout << "you passed me " << n << endl;
}

int main() {
  myFunction(5); // calling myFunction with arugment 5 for the parameter n

  return 0;
}
```

prints: you passed me 5

Examples

Example of a function that returns a value:

```
int myFunction(int n) {
  cout << "I am going to return " << n << endl;

  return n;
}

int main() {
  cout << myFunction(5) << endl;
  return 0;
}
```

What is printed?

Examples

Example of a function that returns a value:

```
int myFunction(int n) {
  cout << "I am going to return " << n << endl;

  return n;
}

int main() {
  cout << myFunction(5) << endl;
  return 0;
}
```

What is printed?

I am going to return 5
5

Examples

Example of a function with no parameters

```
int myFunction() {

  int n = 5;

  return n;
}

int main() {
  cout << myFunction();
  return 0;
}
```

What is printed?

Examples

Example of a function with no parameters

```
int myFunction() {

  int n = 5;

  return n;
}

int main() {
  cout << myFunction(); // call the function with no arguments
  return 0;
}
```

What is printed?

5

Exercise

```
void myFunction() {
}

int main() {
  cout << myFunction();
  return 0;
}
```

What happens if you run this code?

Exercise

```
void myFunction() {
}

int main() {
  cout << myFunction();
  return 0;
}
```

What happens if you run this code?

ERROR, cout cannot handle the value of myFunction()

Repeated Function Calls

A function can be called multiple times:

```
void myFunction() {
  cout << "I just got called" << endl;
}

int main() {
  for (int i = 0; i < 3; ++i) {
    myFunction();
  }
  return 0;
}
```

Repeated Function Calls

A function can be called multiple times:

```
void myFunction() {
  cout << "I just got called" << endl;
}

int main() {
  for (int i = 0; i < 3; ++i) {
    myFunction();
  }
  return 0;
}
```

Prints:
I just got called
I just got called
I just got called

Practice

Recall the vector chess board problem:

Create a 2d vector of size n by n storing a chess board.

Part 1: Each cell is black. Use “b” to denote black.

Part 2: print the board

Part 3: Now correct the board to contain alternating “b” and “w”.

Place parts 1 and 3 in a function called initialize_board, with one parameter, n.

Place part 2 in a function print_board, with a 2d vector argument that represents the board.

Write the main function reads in n. Then it will call initialize_board and print_board.

Passing by Value and Scopes

What does the following print?

```
int add1(int x) {
    return x + 1;
}
int main() {
    int x = 10;
    cout << add1 (x) << endl;
    cout << x << endl;
}
```

Passing by Value and Scopes

What does the following print?

```
int add1(int x) {
    return x + 1;
}
int main() {
    int x = 10;
    cout << add1 (x) << endl;
    cout << x << endl;
}
```

Print:

10

11

Passing by Value and Scopes

What does the following print?

```
int add1(int x) {
    return x + 1;
}
int main() {
    int x = 10;
    cout << add1 (x) << endl; // we are passing the value 10 to add1
    cout << x << endl; // the value of x remains constant
}
```

Print:

10

11

Passing by Value and Scopes

- Add1 got the value 10 when main called it
- The parameter x is a variable name within the **scope** of the add1 function, we say x **binds** to the function add1

Add1:
Input: x = 10
x = x + 1 => 11
return x (return the value 11)

Main:
x = 10
add1(x) -> add1 (10)

Passing by Value and Scopes

- Add1 got the value 10 when main called it
- The parameter x is a variable name within the **scope** of the add1 function, we say x **binds** to the function add1

- Calling add1(x) passes the value of x to add1, same as add1(10)
- The variable x is a name that binds to the main function

Add1:
Input: x = 10
x = x + 1 => 11
return x (return the value 11)

Main:
x = 10
add1(x) -> add1 (10)

Passing by Value and Scopes

- Add1 got the value 10 when main called it
- The parameter x is a variable name within the **scope** of the add1 function, we say x **binds** to the function add1

Add1:
Input: x = 10
x = x + 1 => 11
return x (return the value 11)

- Calling add1(x) passes the value of x to add1, same as add1(10)
- The variable x is a name that binds to the main function

Main:
x = 10
add1(x) -> add1 (10)

- We say that we passed the value 10 to add1, so this is a function call with arguments passed **by value**
- We will learn how to **pass by reference** later

Passing by Value and Scopes

- Add1 got the value 10 when main called it
- The parameter x is a variable name within the **scope** of the add1 function, we say x **binds** to the function add1

Add1:
Input: x = 10
x = x + 1 => 11
return x (return the value 11)

* The same applies for all values, including strings, chars, bools, arrays, and vectors

- Calling add1(x) passes the value of x to add1, same as add1(10)
- The variable x is a name that binds to the main function

Main:
x = 10
add1(x) -> add1 (10)

- We say that we passed the value 10 to add1, so this is a function call with arguments passed **by value**
- We will learn how to **pass by reference** later

Exercise

```
void foo(int x){
    cout << "x = " << x << endl;
    x = 6;
    cout << "x = " << x << endl;
}

int main() {
    int x = 5;
    cout << "x = " << x << endl;
    foo(x);
    cout << "x = " << x << endl;
    return 0;
}
```

What prints?

Exercise

```
void foo(int x){
    cout << "x = " << x << endl;
    x = 6;
    cout << "x = " << x << endl;
}

int main() {
    int x = 5;
    cout << "x = " << x << endl;
    foo(x);
    cout << "x = " << x << endl;
    return 0;
}
```

What prints?

```
x = 5
x = 5
x = 6
x = 5
```

Function Overloading

- You are overloading a function if the new function share the same name as an excising function
- These two functions must have different function signatures
  - Function signature – number , order, and type of parameters
- The compiler is smart enough to figure out which function you are calling as long as you have different function signatures
- Otherwise it will not let you compile

Example

```
void foo() {
    cout << "You called foo()" << endl;
}
void foo (int n) {
    cout << "You called foo(int n)" << endl;
}
int main() {
    foo ();
    foo (1);
}
```

Output:
You called foo
You called foo(int n)

Exercise

- Write a function weird_math that consumes 1,2, or 3 integers.
  - If there is only one integer, it will return the integer
  - If there are two, it will return their products
  - If there are three, it will return the weird_math of the first and the third integers plus the second integer

Given the main function:

```
int main() {
    cout << weird_math (1, 2, 3) << endl;
    cout <<weird_math (1) << endl;
    cout <<weird_math (1, 2) << endl;
}
```

Produce:

```
5
1
2
```

Exercise

Which of the following are valid function to declare given that

int foo (int a) {…}

has already been defined.

1. int foo (int a) {…}
2. bool foo (int a) {…}
3. int foo (int b) {…}
4. int foo (int a, int b) {…}

Exercise

Which of the following are valid function to declare given that

int foo (int a) {…} and

int foo (int a, int b) {…}

has already been defined.

1. int foo (int a, bool b) {…}
2. int foo (int b, int a) {…}

Exercise

Which of the following are valid function to declare given that

int foo (int a) {…} and

int foo (int a, int b) {…}

int foo (int a, bool b) {…}

has already been defined.

1. int foo (bool a, int b) {…}