## 10 - How to Eat an Elephant

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College





## Outline

- Complexity and Programming
- 2 Functions
- Thinking in Functions





## Outline

- Complexity and Programming
- 2 Functions
- Thinking in Functions





## A Bite of Wisdom

There is only one way to eat an elephant:





### A Bite of Wisdom

There is only one way to eat an elephant: a bite at a time.

- Desmond Tutu





### With Apologies to Douglas Adams





#### With Apologies to Douglas Adams

Useful software is big! You just won't believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think that program three is difficult, but that's just peanuts compared to real applications.

 A small useful application is usually around 5,000 lines of code.





#### With Apologies to Douglas Adams

- A small useful application is usually around 5,000 lines of code.
- Most real-world software applications have more than 100,000 lines of functional code.





#### With Apologies to Douglas Adams

- A small useful application is usually around 5,000 lines of code.
- Most real-world software applications have more than 100,000 lines of functional code.
- If you look at an entire software system, you can easily break the 1,000,000 line barrier!





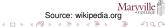
### With Apologies to Douglas Adams

- A small useful application is usually around 5,000 lines of code.
- Most real-world software applications have more than 100,000 lines of functional code.
- If you look at an entire software system, you can easily break the 1,000,000 line barrier!
- Each line of code is like a moving part in a machine.



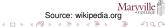
• Thus far, all known software is written by humans.





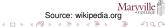
- Thus far, all known software is written by humans.
- The human race is a member of the hominidae family.





- Thus far, all known software is written by humans.
- The human race is a member of the hominidae family.
- We are apes.





- Thus far, all known software is written by humans.
- The human race is a member of the hominidae family.
- We are apes.
- We are the most successful ape.



- Thus far, all known software is written by humans.
- The human race is a member of the hominidae family.
- We are apes.
- We are the most successful ape.
- We are still apes, nonetheless.



- Thus far, all known software is written by humans.
- The human race is a member of the hominidae family.
- We are apes.
- We are the most successful ape.
- We are still apes, nonetheless.
- We can hold about seven ideas in our heads at once.





- Thus far, all known software is written by humans.
- The human race is a member of the hominidae family.
- We are apes.
- We are the most successful ape.
- We are still apes, nonetheless.
- We can hold about seven ideas in our heads at once.
- This is insufficient for almost all useful programming tasks.



• The most important skill in programming is decomposition.





- The most important skill in programming is decomposition.
- That is, the key skill is the ability to break a problem down into smaller chunks.





- The most important skill in programming is decomposition.
- That is, the key skill is the ability to break a problem down into smaller chunks.
- Because we are violent ape-brained creatures, we have to have mechanisms which allow us to focus on smaller parts of a programming function.





- The most important skill in programming is decomposition.
- That is, the key skill is the ability to break a problem down into smaller chunks.
- Because we are violent ape-brained creatures, we have to have mechanisms which allow us to focus on smaller parts of a programming function.
- C++ provides two such mechanisms:





- The most important skill in programming is decomposition.
- That is, the key skill is the ability to break a problem down into smaller chunks.
- Because we are violent ape-brained creatures, we have to have mechanisms which allow us to focus on smaller parts of a programming function.
- C++ provides two such mechanisms:
  - Modular Decomposition (functions)





- The most important skill in programming is decomposition.
- That is, the key skill is the ability to break a problem down into smaller chunks.
- Because we are violent ape-brained creatures, we have to have mechanisms which allow us to focus on smaller parts of a programming function.
- C++ provides two such mechanisms:
  - Modular Decomposition (functions)
  - Object Oriented Programming (classes and objects)





- The most important skill in programming is decomposition.
- That is, the key skill is the ability to break a problem down into smaller chunks.
- Because we are violent ape-brained creatures, we have to have mechanisms which allow us to focus on smaller parts of a programming function.
- C++ provides two such mechanisms:
  - Modular Decomposition (functions)
  - Object Oriented Programming (classes and objects)
- These allow us to create abstractions.



- The most important skill in programming is decomposition.
- That is, the key skill is the ability to break a problem down into smaller chunks.
- Because we are violent ape-brained creatures, we have to have mechanisms which allow us to focus on smaller parts of a programming function.
- C++ provides two such mechanisms:
  - Modular Decomposition (functions)
  - Object Oriented Programming (classes and objects)
- These allow us to create abstractions.
- We have to hide the other 4,393 parts of the problem so we can focus on the seven parts we are capable of.





## Outline

- Complexity and Programming
- 2 Functions
- 3 Thinking in Functions





```
Function Syntax

return_type name( parameters )
{
    //function body
}
```





```
Function Syntax

return_type name( parameters )
{
    //function body
}
```





```
Function Syntax
return_type name( parameters )
{
    //function body
}
```

 A function is a block of code that can be called multiple times.





```
Function Syntax
return_type name( parameters )
{
    //function body
}
```

- A function is a block of code that can be called multiple times.
- A function's signature consists of the following:





```
Function Syntax
return_type name( parameters )
{
    //function body
}
```

- A function is a block of code that can be called multiple times.
- A function's signature consists of the following: return type This is the type of value the function evaluates to when it is used in an expression.





```
Function Syntax
return_type name( parameters )
{
    //function body
}
```

- A function is a block of code that can be called multiple times.
- A function's signature consists of the following:
   return type This is the type of value the function evaluates to when it is used in an expression.
  - name The identifier which names the function.



```
Function Syntax
return_type name( parameters )
{
    //function body
}
```

- A function is a block of code that can be called multiple times.
- A function's signature consists of the following:
  - return type This is the type of value the function evaluates to when it is used in an expression.
  - name The identifier which names the function.

    parameters The local variables which receive the arguments of the function.

## **Example: The Main Function**

```
int main()
{
    cout << "Hello, world" << endl;
    return 0;
}</pre>
```

Every C++ program has at least one function.





# **Example: The Main Function**

```
int main()
{
    cout << "Hello, world" << endl;
    return 0;
}</pre>
```

- Every C++ program has at least one function.
- This function is named main.





## **Example: The Main Function**

```
int main()
{
    cout << "Hello, world" << endl;
    return 0;
}</pre>
```

- Every C++ program has at least one function.
- This function is named main.
- The main function above takes no arguments.





# **Example: The Main Function**

```
int main()
{
    cout << "Hello, world" << endl;
    return 0;
}</pre>
```

- Every C++ program has at least one function.
- This function is named main.
- The main function above takes no arguments.
- The main function returns an integer.





# **Example: The Main Function**

```
int main()
{
    cout << "Hello, world" << endl;
    return 0;
}</pre>
```

- Every C++ program has at least one function.
- This function is named main.
- The main function above takes no arguments.
- The main function returns an integer.
- We can explicitly return a value by using the return keyword.



 Sometimes, it is desirable to have a function do something, but return no value.





- Sometimes, it is desirable to have a function do something, but return no value.
- Such a function has a return type of void





- Sometimes, it is desirable to have a function do something, but return no value.
- Such a function has a return type of void
- Take a look at examples/10-Elephant/roman.cpp





- Sometimes, it is desirable to have a function do something, but return no value.
- Such a function has a return type of void
- Take a look at examples/10-Elephant/roman.cpp





- Sometimes, it is desirable to have a function do something, but return no value.
- Such a function has a return type of void
- Take a look at examples/10-Elephant/roman.cpp

```
//Print the roman numeral for the given value.
//This function can print values for 1,4,5,9, and 10
//All other values print "invalid"
void print roman numeral (int value)
    if(value == 1) {
        cout << "T":
    } else if(value == 4) {
        cout << "IV";
    } else if(value == 5) {
        cout << "V";
    } else if(value == 9) {
        cout << "IX";
    } else if(value == 10) {
        cout << "X";
    } else {
        cout << "Invalid";
```





 Functions are called by typing their name and putting their arguments in parenthesis.





- Functions are called by typing their name and putting their arguments in parenthesis.
- Take, for example, the main function from roman.cpp





- Functions are called by typing their name and putting their arguments in parenthesis.
- Take, for example, the main function from roman.cpp





- Functions are called by typing their name and putting their arguments in parenthesis.
- Take, for example, the main function from roman.cpp

```
int main()
    int x;
    //get the number
    cout << "Enter a number: ";</pre>
    cin >> x:
    //print it as a roman numeral
    print_roman_numeral(x);
    cout << endl;
```

• roman.cpp works, but the main function is at the end.





- roman.cpp works, but the main function is at the end.
- It would make more sense to have the main function be the first function in the file.





- roman.cpp works, but the main function is at the end.
- It would make more sense to have the main function be the first function in the file.
- Copy roman.cpp to labs/week6





- roman.cpp works, but the main function is at the end.
- It would make more sense to have the main function be the first function in the file.
- Copy roman.cpp to labs/week6
- Try moving the print\_roman\_numeral function definition to the end of the file.





- roman.cpp works, but the main function is at the end.
- It would make more sense to have the main function be the first function in the file.
- Copy roman.cpp to labs/week6
- Try moving the print\_roman\_numeral function definition to the end of the file.
- Compile and run.





- roman.cpp works, but the main function is at the end.
- It would make more sense to have the main function be the first function in the file.
- Copy roman.cpp to labs/week6
- Try moving the print\_roman\_numeral function definition to the end of the file.
- Compile and run.
- Why doesn't it work?





 Function prototypes allow you to declare a function before it is defined.





- Function prototypes allow you to declare a function before it is defined.
- This is a sort of "contract" between you and the compiler.





- Function prototypes allow you to declare a function before it is defined.
- This is a sort of "contract" between you and the compiler.
- This allows you to have functions in any order in the file.





- Function prototypes allow you to declare a function before it is defined.
- This is a sort of "contract" between you and the compiler.
- This allows you to have functions in any order in the file.
- Change the first few lines of roman.cpp so it reads as follows:

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

//function prototypes
void print_roman_numeral(int value);
```





 The main function should be the first function definition in the file.





- The main function should be the first function definition in the file.
- You should provide prototypes for every function other than the main function.





- The main function should be the first function definition in the file.
- You should provide prototypes for every function other than the main function.
- Your files should be ordered as follows:





- The main function should be the first function definition in the file.
- You should provide prototypes for every function other than the main function.
- Your files should be ordered as follows:
  - Opening comment, explaining the program.





- The main function should be the first function definition in the file.
- You should provide prototypes for every function other than the main function.
- Your files should be ordered as follows:
  - Opening comment, explaining the program.
  - All of your #include directives.





- The main function should be the first function definition in the file.
- You should provide prototypes for every function other than the main function.
- Your files should be ordered as follows:
  - Opening comment, explaining the program.
  - All of your #include directives.
  - A section for function prototypes. (labeled)





- The main function should be the first function definition in the file.
- You should provide prototypes for every function other than the main function.
- Your files should be ordered as follows:
  - Opening comment, explaining the program.
  - All of your #include directives.
  - A section for function prototypes. (labeled)
  - The main function.





- The main function should be the first function definition in the file.
- You should provide prototypes for every function other than the main function.
- Your files should be ordered as follows:
  - Opening comment, explaining the program.
  - All of your #include directives.
  - A section for function prototypes. (labeled)
  - The main function.
  - 6 All of the other functions.





- The main function should be the first function definition in the file.
- You should provide prototypes for every function other than the main function.
- Your files should be ordered as follows:
  - Opening comment, explaining the program.
  - All of your #include directives.
  - A section for function prototypes. (labeled)
  - The main function.
  - All of the other functions.
- Every function (other than main) should have a comment before their definition which explains what the function does.



## Lab Activity: Finish print\_roman\_numeral

Edit the print\_roman\_numeral function to include all other roman numerals.

I 1			
IV 4	XL 40	CD	400
	L 50	D	500
V 5	XC 90	CM	900
IX 9	C 10	 M	1000
Y 10	0 10	, o lvi	1000





#### Outline

- Complexity and Programming
- 2 Functions
- Thinking in Functions





 As we design a task, we often have tasks that will have many sub steps.





- As we design a task, we often have tasks that will have many sub steps.
- For example:





- As we design a task, we often have tasks that will have many sub steps.
- For example:
  - Read a number





- As we design a task, we often have tasks that will have many sub steps.
- For example:
  - Read a number
  - Translate into a roman numeral





#### Top-Down Design

- As we design a task, we often have tasks that will have many sub steps.
- For example:
  - Read a number
  - Translate into a roman numeral
- We could translate into the following (go ahead and change your main function to this):

```
int main()
{
    int num;

    //read number
    cout << "Enter a number: ";
    cin >> num;

    indian_to_roman(num);
```



#### Lab Activity: Roman Numeral Translator

 Go ahead and add a function prototype for our new function:

```
void indian_to_roman(int num);
```





#### Lab Activity: Roman Numeral Translator

 Go ahead and add a function prototype for our new function:

```
void indian_to_roman(int num);
```

 Now, at the bottom of your file, add an empty definition for the function:

```
void indian_to_roman(int num)
{
```





• We will translate to roman numerals as follows:



- We will translate to roman numerals as follows:
  - Start the value at 1000.





- We will translate to roman numerals as follows:
  - Start the value at 1000.
  - Divide the number by the value.





- We will translate to roman numerals as follows:
  - Start the value at 1000.
  - Divide the number by the value.
  - Print that many of print\_roman\_numeral (value)





- We will translate to roman numerals as follows:
  - Start the value at 1000.
  - Divide the number by the value.
  - Print that many of print\_roman\_numeral (value)
  - Set value to the next roman numeral value.





- We will translate to roman numerals as follows:
  - Start the value at 1000.
  - Divide the number by the value.
  - Print that many of print\_roman\_numeral (value)
  - Set value to the next roman numeral value.
  - Subtract what we have just printed from the number.





- We will translate to roman numerals as follows:
  - Start the value at 1000.
  - Divide the number by the value.
  - Print that many of print\_roman\_numeral (value)
  - Set value to the next roman numeral value.
  - Subtract what we have just printed from the number.
  - Repeat the process until the number is zero.





- We will translate to roman numerals as follows:
  - Start the value at 1000.
  - Divide the number by the value.
  - Print that many of print\_roman\_numeral (value)
  - Set value to the next roman numeral value.
  - Subtract what we have just printed from the number.
  - Repeat the process until the number is zero.
- Let's discuss. What functions are there in the above?





- We will translate to roman numerals as follows:
  - Start the value at 1000.
  - Divide the number by the value.
  - Print that many of print\_roman\_numeral (value)
  - Set value to the next roman numeral value.
  - Subtract what we have just printed from the number.
  - Repeat the process until the number is zero.
- Let's discuss. What functions are there in the above?
- How do we do each step?





- We will translate to roman numerals as follows:
  - Start the value at 1000.
  - ② Divide the number by the value.
  - Print that many of print\_roman\_numeral (value)
  - Set value to the next roman numeral value.
  - Subtract what we have just printed from the number.
  - 6 Repeat the process until the number is zero.
- Let's discuss. What functions are there in the above?
- How do we do each step?
- Let's build this thing!





## Lab Week 6 Requirements

You must have the following for full credit:

- count2.cpp (for loop version)
- fahrenheit.cpp (for loop version)
- double-count.cpp (fully corrected version)
- roman.cpp (able to translate to roman numerals)



