10 - How to Eat an Elephant

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College

Outline

Outline

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*

Software and Complexity

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.

Software and Complexity

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.

Software and Complexity

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.
- Most real-world software applications have more than 100,000 lines of functional code.

Software and Complexity

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.
- 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!

Software and Complexity

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.
- 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.

The Problem

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

Human[1]

Temporal range: 0.35–0 Ma

Middle Pleistocene – Recent

An adult human male (left) and female (right) from the Akha tribe in Northern Thailand.

Conservation status

Least Concern (IUCN 3.1)[2]

Source: wikipedia.org

The Problem

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

Human[1]
Temporal range: 0.35–0 Ma

Middle Pleistocene – Recent

An adult human male (left) and female (right) from the Akha tribe in Northern Thailand.

Conservation status

Least Concern (IUCN 3.1)[2]

Source: wikipedia.org

The Problem

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

Human[1]
Temporal range: 0.35–0 Ma

Middle Pleistocene – Recent

An adult human male (left) and female (right) from the Akha tribe in Northern Thailand.

Conservation status

Least Concern (IUCN 3.1)[2]

Source: wikipedia.org

The Problem

- 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.

Human[1]
Temporal range: 0.35–0 Ma
PreЄ Є O S D C P T J K PgN
Middle Pleistocene – Recent

An adult human male (left) and female (right) from the Akha tribe in Northern Thailand.

Conservation status

Least Concern (IUCN 3.1)[2]

Source: wikipedia.org

The Problem

- 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.

Human[1]
Temporal range: 0.35–0 Ma
PreЄ Є O S D C P T J K PgN
Middle Pleistocene – Recent

An adult human male (left) and female (right) from the Akha tribe in Northern Thailand.

Conservation status

Least Concern (IUCN 3.1)[2]

Source: wikipedia.org

The Problem

- 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.

Human[1]
Temporal range: 0.35–0 Ma
PreЄ Є O S D C P T J K PgN
Middle Pleistocene – Recent

An adult human male (left) and female (right) from the Akha tribe in Northern Thailand.

Conservation status

Least Concern (IUCN 3.1)[2]

Source: wikipedia.org

The Problem

- 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.

Human[1]
Temporal range: 0.35–0 Ma
PreЄ Є O S D C P T J K PgN
Middle Pleistocene – Recent

An adult human male (left) and female (right) from the Akha tribe in Northern Thailand.

Conservation status

Least Concern (IUCN 3.1)[2]

Source: wikipedia.org

The Solution

- The most important skill in programming is decomposition.

The Solution

- 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 Solution

- 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 Solution

- 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 Solution

- 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 Solution

- 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 Solution

- 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 Solution

- 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

Function Definition

Function Syntax

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

Function Definition

Function Syntax

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

Function Definition

Function Syntax

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

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

Function Definition

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 Definition

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 Definition

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 Definition

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;
}
```

- Every C++ program has at least one function.

Example: The Main Function

```
int main()
{
    cout << "Hello, world" << endl;

    return 0;
}
```

- 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;
}
```

- 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;
}
```

- 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;
}
```

- 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.

Void Functions

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

Void Functions

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

Void Functions

- 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`

Void Functions

- 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`

Void Functions

- 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 << "I";
    } 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";
    }
}
```

Calling Functions

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

Calling Functions

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

Calling Functions

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

Calling Functions

- 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: ";
    cin >> x;

    //print it as a roman numeral
    print_roman_numeral(x);
    cout << endl;
}
```

The Structure of `roman.cpp`

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

The Structure of `roman.cpp`

- `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.

The Structure of `roman.cpp`

- `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`

The Structure of `roman.cpp`

- `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.

The Structure of `roman.cpp`

- `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.

The Structure of `roman.cpp`

- `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

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

Function Prototypes

- 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

- 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

- 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);
```

Best Practice for Files With Functions

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

Best Practice for Files With 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.

Best Practice for Files With 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:

Best Practice for Files With 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:
  1. Opening comment, explaining the program.

Best Practice for Files With 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:
  1. Opening comment, explaining the program.
  2. All of your `#include` directives.

Best Practice for Files With 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:
  1. Opening comment, explaining the program.
  2. All of your `#include` directives.
  3. A section for function prototypes. (labeled)

Best Practice for Files With 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:
  1. Opening comment, explaining the program.
  2. All of your `#include` directives.
  3. A section for function prototypes. (labeled)
  4. The `main` function.

Best Practice for Files With 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:
    1. Opening comment, explaining the program.
    2. All of your `#include` directives.
    3. A section for function prototypes. (labeled)
    4. The `main` function.
    5. All of the other functions.

Best Practice for Files With 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:
  1. Opening comment, explaining the program.
  2. All of your `#include` directives.
  3. A section for function prototypes. (labeled)
  4. The `main` function.
  5. 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	XL	40	CD	400
IV	4	L	50	D	500
V	5	XC	90	CM	900
IX	9	C	100	M	1000
X	10				

Outline

Top-Down Design

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

Top-Down Design

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

Top-Down Design

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

Top-Down Design

- As we design a task, we often have tasks that will have many sub steps.
- For example:
  1. Read a number
  2. 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:
  1. Read a number
  2. 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)
  {


  }
  ```

Roman Numeral Translation Steps

- We will translate to roman numerals as follows:

Roman Numeral Translation Steps

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

Roman Numeral Translation Steps

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

Roman Numeral Translation Steps

- We will translate to roman numerals as follows:
  1. Start the value at 1000.
  2. Divide the number by the value.
  3. Print that many of `print_roman_numeral(value)`

Roman Numeral Translation Steps

- We will translate to roman numerals as follows:
    1. Start the value at 1000.
    2. Divide the number by the value.
    3. Print that many of `print_roman_numeral(value)`
    4. Set value to the next roman numeral value.

Roman Numeral Translation Steps

- We will translate to roman numerals as follows:
  1. Start the value at 1000.
  2. Divide the number by the value.
  3. Print that many of `print_roman_numeral(value)`
  4. Set value to the next roman numeral value.
  5. Subtract what we have just printed from the number.

Roman Numeral Translation Steps

- We will translate to roman numerals as follows:
  1. Start the value at 1000.
  2. Divide the number by the value.
  3. Print that many of `print_roman_numeral(value)`
  4. Set value to the next roman numeral value.
  5. Subtract what we have just printed from the number.
  6. Repeat the process until the number is zero.

Roman Numeral Translation Steps

- We will translate to roman numerals as follows:
  1. Start the value at 1000.
  2. Divide the number by the value.
  3. Print that many of `print_roman_numeral(value)`
  4. Set value to the next roman numeral value.
  5. 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?

Roman Numeral Translation Steps

- We will translate to roman numerals as follows:
  1. Start the value at 1000.
  2. Divide the number by the value.
  3. Print that many of `print_roman_numeral(value)`
  4. Set value to the next roman numeral value.
  5. 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?

Roman Numeral Translation Steps

- We will translate to roman numerals as follows:
  1. Start the value at 1000.
  2. Divide the number by the value.
  3. Print that many of `print_roman_numeral(value)`
  4. Set value to the next roman numeral value.
  5. 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)