11 - Functions and Topdown Design

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College

Maryville COLLEGE

Outline

Outline

Roman Numeral Conversion Functions

After decomposing the problem, one possible set of functions for convert Indian numbers to Roman Numerals is:

Roman Numeral Conversion Functions

After decomposing the problem, one possible set of functions for convert Indian numbers to Roman Numerals is:

- `print_roman_numeral(value)`

Roman Numeral Conversion Functions

After decomposing the problem, one possible set of functions for convert Indian numbers to Roman Numerals is:

- `print_roman_numeral(value)`
- `indian_to_roman(num)`

Roman Numeral Conversion Functions

After decomposing the problem, one possible set of functions for convert Indian numbers to Roman Numerals is:

- `print_roman_numeral(value)`
- `indian_to_roman(num)`
- `repeat_roman(n, value)`

Roman Numeral Conversion Functions

After decomposing the problem, one possible set of functions for convert Indian numbers to Roman Numerals is:

- `print_roman_numeral(value)`
- `indian_to_roman(num)`
- `repeat_roman(n, value)`
- `next_roman(value)`

repeat_roman(n, value)

The design for `repeat_roman` goes something like this:

repeat_roman(n, value)

The design for `repeat_roman` goes something like this:

`repeat_roman(n, value)`

The design for `repeat_roman` goes something like this:

1. Call `print_roman_numeral(value)` `n` times.

next_roman(value)

next_roman(value)

- Of course, the hard part is going through the possible Roman values.

next_roman(value)

- Of course, the hard part is going through the possible Roman values.
- This function gives us the next number in the sequence: 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1

next_roman(value)

- Of course, the hard part is going through the possible Roman values.
- This function gives us the next number in the sequence: 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1
- We should make sure that if we give it a 1, we return 1. Why?

next_roman(value)

- Of course, the hard part is going through the possible Roman values.
- This function gives us the next number in the sequence: 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1
- We should make sure that if we give it a 1, we return 1. Why?
- We could implement this using a big chain of if .. else if statements.

next_roman(value)

- Of course, the hard part is going through the possible Roman values.
- This function gives us the next number in the sequence: 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1
- We should make sure that if we give it a 1, we return 1. Why?
- We could implement this using a big chain of if .. else if statements.
- Can we come up with a more clever way to pull off the function?

Roman Recap

Roman Recap

- Roman numeral conversions would have been a nightmare if we tried to do this all in one function!

Roman Recap

- Roman numeral conversions would have been a nightmare if we tried to do this all in one function!
- Thanks to the ability to subdivide the problem into different functions, it was only a slightly uncomfortable dream.

Roman Recap

- Roman numeral conversions would have been a nightmare if we tried to do this all in one function!
- Thanks to the ability to subdivide the problem into different functions, it was only a slightly uncomfortable dream.
- Modular decomposition is the crucial skill in any large scale program.

Roman Recap

- Roman numeral conversions would have been a nightmare if we tried to do this all in one function!
- Thanks to the ability to subdivide the problem into different functions, it was only a slightly uncomfortable dream.
- Modular decomposition is the crucial skill in any large scale program.
- By our simple ape-brained standards, almost all programs are large scale problems.

Outline

Scope

Scope

Scope

A **scope** is a region of program text. Identifiers are declared within a scope and are only available within their scope.

Scope

Scope

A **scope** is a region of program text. Identifiers are declared within a scope and are only available within their scope.

- Identifiers within a scope must be unique.

Scope

Scope

A **scope** is a region of program text. Identifiers are declared within a scope and are only available within their scope.

- Identifiers within a scope must be unique.
- The C++ scopes are:

Scope

Scope

A **scope** is a region of program text. Identifiers are declared within a scope and are only available within their scope.

- Identifiers within a scope must be unique.
- The C++ scopes are:
  - **global scope** The entire program text.

Scope

Scope

A **scope** is a region of program text. Identifiers are declared within a scope and are only available within their scope.

- Identifiers within a scope must be unique.
- The C++ scopes are:
  - **global scope** The entire program text.
  - **namespace scope** A named scope (more about this much later).

Scope

Scope

A **scope** is a region of program text. Identifiers are declared within a scope and are only available within their scope.

- Identifiers within a scope must be unique.
- The C++ scopes are:
  - **global scope** The entire program text.
  - **namespace scope** A named scope (more about this much later).
  - **class scope** The region within a class. (more about this later this semester).

Scope

Scope

A **scope** is a region of program text. Identifiers are declared within a scope and are only available within their scope.

- Identifiers within a scope must be unique.
- The C++ scopes are:
  - **global scope** The entire program text.
  - **namespace scope** A named scope (more about this much later).
  - **class scope** The region within a class. (more about this later this semester).
  - **local scope** The region within a function body or the function's argument list.

Scope

Scope

A **scope** is a region of program text. Identifiers are declared within a scope and are only available within their scope.

- Identifiers within a scope must be unique.
- The C++ scopes are:
  - **global scope** The entire program text.
  - **namespace scope** A named scope (more about this much later).
  - **class scope** The region within a class. (more about this later this semester).
  - **local scope** The region within a function body or the function's argument list.
  - **block/statement scope** The region within a statement or between two curly braces { }.

Scope Nesting: 11-Functions/scope.cpp

```
#include<iostream>

using namespace std;

//Global Scope

void count(int start, int stop, int increment)
{
    for(int i=start; i<=stop; i+=increment) {
        cout << i << endl;
    }
}

int main()
{
    int start, stop, increment;

    //read in start stop and increment
    cout << "Enter start stop and increment: ";
    cin >> start >> stop >> increment;

    //count
    count(start, stop, increment);
}
```

A Scope Puzzle: 11-Functions/puzzle.cpp

```
// What will the following program display?
#include<iostream>

using namespace std;

void scope_test(int x)
{
    x += 10;
}


int main()
{
    int x = 32;

    scope_test(x);

    cout << x << endl;
}
```

Argument Passing in C++

Argument Passing in C++

Argument Passing in C++

- In C++, the default behavior for arguments is **pass by value**.

Argument Passing in C++

- In C++, the default behavior for arguments is **pass by value**.
  - The value of the argument is copied into the local scoped variable named in the parameter list.

Argument Passing in C++

- In C++, the default behavior for arguments is **pass by value**.
  - The value of the argument is copied into the local scoped variable named in the parameter list.
  - In `puzzle.cpp`, even though they have the same name, `x` in `main` is a different variable than `x` in `scope_test`.

Argument Passing in C++

- In C++, the default behavior for arguments is **pass by value**.
  - The value of the argument is copied into the local scoped variable named in the parameter list.
  - In `puzzle.cpp`, even though they have the same name, `x` in `main` is a different variable than `x` in `scope_test`.
  - The two `x`'s are in a different scope.

Argument Passing in C++

- In C++, the default behavior for arguments is **pass by value**.
  - The value of the argument is copied into the local scoped variable named in the parameter list.
  - In `puzzle.cpp`, even though they have the same name, `x` in `main` is a different variable than `x` in `scope_test`.
  - The two `x`'s are in a different scope.
- We can also **pass by reference**.

Argument Passing in C++

- In C++, the default behavior for arguments is **pass by value**.
  - The value of the argument is copied into the local scoped variable named in the parameter list.
  - In `puzzle.cpp`, even though they have the same name, `x` in `main` is a different variable than `x` in `scope_test`.
  - The two `x`'s are in a different scope.
- We can also **pass by reference**.
  - A reference parameter is declared by placing an `&` between the type and parameter name.

Argument Passing in C++

- In C++, the default behavior for arguments is **pass by value**.
  - The value of the argument is copied into the local scoped variable named in the parameter list.
  - In `puzzle.cpp`, even though they have the same name, `x` in `main` is a different variable than `x` in `scope_test`.
  - The two `x`'s are in a different scope.
- We can also **pass by reference**.
  - A reference parameter is declared by placing an `&` between the type and parameter name.
  - For example: `scope_test(int &x)`

Argument Passing in C++

- In C++, the default behavior for arguments is **pass by value**.
  - The value of the argument is copied into the local scoped variable named in the parameter list.
  - In `puzzle.cpp`, even though they have the same name, `x` in `main` is a different variable than `x` in `scope_test`.
  - The two `x`'s are in a different scope.
- We can also **pass by reference**.
  - A reference parameter is declared by placing an `&` between the type and parameter name.
  - For example: `scope_test(int &x)`
  - This binds the parameter to the argument variable, so that both names refer to the same actual variable.

Another Scope Puzzle: 11-Functions/ref.cpp

```
// We can also pass by reference!  What will this display?
#include<iostream>

using namespace std;

void scope_test(int &x)
{
    x += 10;
}


int main()
{
    int x = 32;

    scope_test(x);

    cout << x << endl;
}
```

Outline

Why Reuse Code?

Why Reuse Code?

- Software is complex and expensive to produce.

Why Reuse Code?

- Software is complex and expensive to produce.
- If we had to constantly rewrite everything, we would never be able to get any work done.

Why Reuse Code?

- Software is complex and expensive to produce.
- If we had to constantly rewrite everything, we would never be able to get any work done.
- We need some way to separate generic code from a specific application.

Why Reuse Code?

- Software is complex and expensive to produce.
- If we had to constantly rewrite everything, we would never be able to get any work done.
- We need some way to separate generic code from a specific application.
- This allows us to reuse code in future projects!

Working With Multiple Files

- One easy way to reuse function is to put them into separate files.

Working With Multiple Files

- One easy way to reuse function is to put them into separate files.
- Often, we put a main function into a file by itself, and then the functions that it calls go into one or more additional files.

Working With Multiple Files

- One easy way to reuse function is to put them into separate files.
- Often, we put a main function into a file by itself, and then the functions that it calls go into one or more additional files.
- In this way, we can copy files between programs, or even write several programs that use the same functions in the same directory.

Separating `roman.cpp`

1. Create the `labs/week7` directory.

Separating `roman.cpp`

1. Create the `labs/week7` directory.
2. Copy `labs/week6/roman.cpp` to `labs/week7`

Separating `roman.cpp`

1. Create the `labs/week7` directory.
2. Copy `labs/week6/roman.cpp` to `labs/week7`
3. Now, create a new file `roman-converter.cpp` and move your main function over to this file. (Be sure to delete your old main function and to include all the iostream stuff that it needs.)

Separating `roman.cpp`

1. Create the `labs/week7` directory.
2. Copy `labs/week6/roman.cpp` to `labs/week7`
3. Now, create a new file `roman-converter.cpp` and move your main function over to this file. (Be sure to delete your old main function and to include all the iostream stuff that it needs.)
4. Try to compile your program:

```
g++ -o roman-converter roman.cpp roman-converter.cpp
```

Separating `roman.cpp`

1. Create the `labs/week7` directory.
2. Copy `labs/week6/roman.cpp` to `labs/week7`
3. Now, create a new file `roman-converter.cpp` and move your main function over to this file. (Be sure to delete your old main function and to include all the iostream stuff that it needs.)
4. Try to compile your program:

   ```
   g++ -o roman-converter roman.cpp roman-converter.cpp
   ```

5. Did it work? Why or why not?

Gluing it Together With Header Files

- A function must be declared before it can be used.

Gluing it Together With Header Files

- A function must be declared before it can be used.
- Because the definitions are in a separate file, they are not declared in the file that contains our main function.

Gluing it Together With Header Files

- A function must be declared before it can be used.
- Because the definitions are in a separate file, they are not declared in the file that contains our main function.
- We can solve this with prototypes.

Gluing it Together With Header Files

- A function must be declared before it can be used.
- Because the definitions are in a separate file, they are not declared in the file that contains our main function.
- We can solve this with prototypes.
- Repeating prototypes in every file is painful.

Gluing it Together With Header Files

- A function must be declared before it can be used.
- Because the definitions are in a separate file, they are not declared in the file that contains our main function.
- We can solve this with prototypes.
- Repeating prototypes in every file is painful.
- Enter the header file!

Gluing it Together With Header Files

- A function must be declared before it can be used.
- Because the definitions are in a separate file, they are not declared in the file that contains our main function.
- We can solve this with prototypes.
- Repeating prototypes in every file is painful.
- Enter the header file!
- A header file usually has a `.h` extension and contains:

Gluing it Together With Header Files

- A function must be declared before it can be used.
- Because the definitions are in a separate file, they are not declared in the file that contains our main function.
- We can solve this with prototypes.
- Repeating prototypes in every file is painful.
- Enter the header file!
- A header file usually has a `.h` extension and contains:
  - Function Prototypes

Gluing it Together With Header Files

- A function must be declared before it can be used.
- Because the definitions are in a separate file, they are not declared in the file that contains our main function.
- We can solve this with prototypes.
- Repeating prototypes in every file is painful.
- Enter the header file!
- A header file usually has a `.h` extension and contains:
  - Function Prototypes
  - Constants

Gluing it Together With Header Files

- A function must be declared before it can be used.
- Because the definitions are in a separate file, they are not declared in the file that contains our main function.
- We can solve this with prototypes.
- Repeating prototypes in every file is painful.
- Enter the header file!
- A header file usually has a `.h` extension and contains:
  - Function Prototypes
  - Constants
  - Type Definitions

Gluing it Together With Header Files

- A function must be declared before it can be used.
- Because the definitions are in a separate file, they are not declared in the file that contains our main function.
- We can solve this with prototypes.
- Repeating prototypes in every file is painful.
- Enter the header file!
- A header file usually has a `.h` extension and contains:
  - Function Prototypes
  - Constants
  - Type Definitions
- Let's create `roman.h`

roman.h

The contents of `roman.h`:

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

roman.h

The contents of `roman.h`:

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

- We only include prototypes for functions we want to call from outside of this file.

roman.h

The contents of `roman.h`:

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

- We only include prototypes for functions we want to call from outside of this file.
- This is the public interface for the roman module.

roman.h

The contents of `roman.h`:

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

- We only include prototypes for functions we want to call from outside of this file.
- This is the public interface for the roman module.
- The other functions are sometimes referred to as **helper functions**.

roman.h

The contents of `roman.h`:

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

- We only include prototypes for functions we want to call from outside of this file.
- This is the public interface for the roman module.
- The other functions are sometimes referred to as **helper functions**.
- Technically, what we have does not conform to best practices, but we will omit some other details for now.

roman.h

The contents of `roman.h`:

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

- We only include prototypes for functions we want to call from outside of this file.
- This is the public interface for the roman module.
- The other functions are sometimes referred to as **helper functions**.
- Technically, what we have does not conform to best practices, but we will omit some other details for now.
- Aren't I a merciful fellow?

Including Header Files

- Header files are included using the `#include` preprocessor directive.

Including Header Files

- Header files are included using the `#include` preprocessor directive.
- When you use angle brackets: `< >`, the preprocessor searches the system's include directories for the file.

Including Header Files

- Header files are included using the `#include` preprocessor directive.
- When you use angle brackets: `< >`, the preprocessor searches the system's include directories for the file.
- When you use quotes: `" "`, the preprocessor searches the local directory for the file.

Including Header Files

- Header files are included using the `#include` preprocessor directive.
- When you use angle brackets: `< >`, the preprocessor searches the system's include directories for the file.
- When you use quotes: `" "`, the preprocessor searches the local directory for the file.
- So when you do `#include<iostream>`, you are searching for the system file called `iostream`.

Including Header Files

- Header files are included using the `#include` preprocessor directive.
- When you use angle brackets: `< >`, the preprocessor searches the system's include directories for the file.
- When you use quotes: `" "`, the preprocessor searches the local directory for the file.
- So when you do `#include<iostream>`, you are searching for the system file called `iostream`.
- To include your header file, change the top of `roman-converter.cpp` to read as follows:

```
#include <iostream>

#include "roman.h"
```

Including Header Files

- Header files are included using the `#include` preprocessor directive.
- When you use angle brackets: `< >`, the preprocessor searches the system's include directories for the file.
- When you use quotes: `" "`, the preprocessor searches the local directory for the file.
- So when you do `#include<iostream>`, you are searching for the system file called `iostream`.
- To include your header file, change the top of `roman-converter.cpp` to read as follows:

```
#include <iostream>

#include "roman.h"
```

- Now we can compile this multi-part program:

```
g++ -o roman-converter roman.cpp roman-converter.cpp
```

Some Best Practices Regarding Header Files

- Always name your headers with the `.h` extension.

Some Best Practices Regarding Header Files

- Always name your headers with the `.h` extension.
- This makes them distinguishable from the C++ system library.

Some Best Practices Regarding Header Files

- Always name your headers with the `.h` extension.
- This makes them distinguishable from the C++ system library.
- When including files, start with the system-wide includes, and then list your own:

```
#include <iostream>
#include <cmath>
#include <time.h>
#include "fun_functions.h"
#include "easy_functions.h"
#include "awful_functions.h"
```

Some Best Practices Regarding Header Files

- Always name your headers with the `.h` extension.
- This makes them distinguishable from the C++ system library.
- When including files, start with the system-wide includes, and then list your own:

```
#include <iostream>
#include <cmath>
#include <time.h>
#include "fun_functions.h"
#include "easy_functions.h"
#include "awful_functions.h"
```

- Keep all of your include directives together at the top of your files.

Some Best Practices Regarding Header Files

- Always name your headers with the `.h` extension.
- This makes them distinguishable from the C++ system library.
- When including files, start with the system-wide includes, and then list your own:

```
#include <iostream>
#include <cmath>
#include <time.h>
#include "fun_functions.h"
#include "easy_functions.h"
#include "awful_functions.h"
```

- Keep all of your include directives together at the top of your files.
- Header files may include other files, but do not put a `using namespace` in a header file.