Program Structure

Dr. Robert Lowe

Department of Computer Information Technology Pellissippi State Community College

Outline

1. Program Structure

2. The Build Process

Outline

1 Program Structure

2 The Build Process

Modular Design

- We want to subdivide programs into manageable chunks.

Modular Design

- We want to subdivide programs into manageable chunks.
- Functions and Structures provide for top-down decompositions.

Modular Design

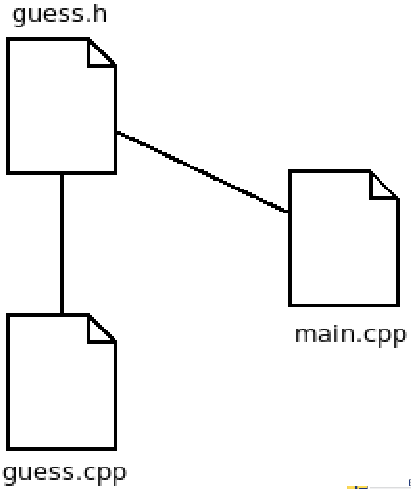
- We want to subdivide programs into manageable chunks.
- Functions and Structures provide for top-down decompositions.
- We can decompose further by grouping related functions into modules.

Modular Design

- We want to subdivide programs into manageable chunks.
- Functions and Structures provide for top-down decompositions.
- We can decompose further by grouping related functions into modules.
- In C++, there are no linguistic modules though we tend to follow the pattern of 1 module per file.

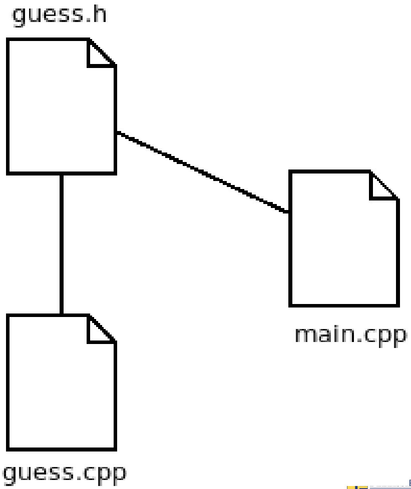
Multifile Programming

- Programs typically have multiple source files.



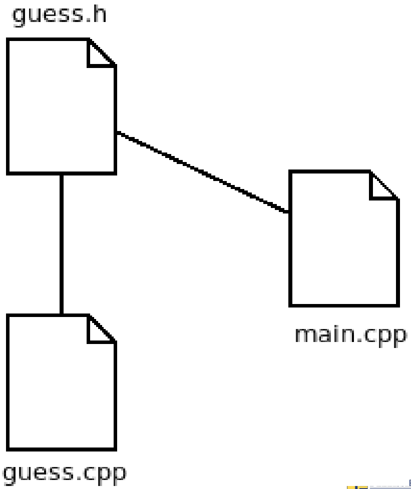
Multifile Programming

- Programs typically have multiple source files.
- Functions are implemented in `.cpp` or implementation files.



Multifile Programming

- Programs typically have multiple source files.
- Functions are implemented in `.cpp` or implementation files.
- Data types and prototypes are placed in `.h` or header files.



Header Files

- Type Definitions

```
//File: guess.h
//Purpose: Header file for the guessing game module

// Type Definitions

// Function Prototypes
```

Header Files

- Type Definitions
- Function Prototypes

```
//File: guess.h
//Purpose: Header file for the guessing game module

// Type Definitions

// Function Prototypes
```

Header Files

- Type Definitions
- Function Prototypes
- Every `.h` file typically has a corresponding `.cpp` file.

```
//File: guess.h
//Purpose: Header file for the guessing game module

// Type Definitions

// Function Prototypes
```

Conditional Compilation

- Prototypes can be repeated.

```
//File: guess.h
//Purpose: Header file for the guessing game module
#ifndef GUESS_H
#define GUESS_H

// Type Definitions

// Function Prototypes
#endif
```

Conditional Compilation

- Prototypes can be repeated.
- Type definitions can only appear once in a program.

```
//File: guess.h
//Purpose: Header file for the guessing game module
#ifndef GUESS_H
#define GUESS_H

// Type Definitions

// Function Prototypes
#endif
```

Conditional Compilation

- Prototypes can be repeated.
- Type definitions can only appear once in a program.
- We use preprocessor directives to protect against multiple inclusions.

```
//File: guess.h
//Purpose: Header file for the guessing game module
#ifndef GUESS_H
#define GUESS_H

// Type Definitions

// Function Prototypes
#endif
```

Implementation File

- The implementation files contain lots of C++ functions and code.

```
//File: guess.cpp
//Purpose: This is the implementation of the guessing game functions.
#include "guess.h"

//C++ Code for Functions Goes Here
```

Implementation File

- The implementation files contain lots of C++ functions and code.
- The main function typically gets its own file, which I like to name `main.cpp`.

```
//File: guess.cpp
//Purpose: This is the implementation of the guessing game functions.
#include "guess.h"

//C++ Code for Functions Goes Here
```

Implementation File

- The implementation files contain lots of C++ functions and code.
- The main function typically gets its own file, which I like to name `main.cpp`.
- I am a creative fellow, after all.

```
//File: guess.cpp
//Purpose: This is the implementation of the guessing game functions.
#include "guess.h"

//C++ Code for Functions Goes Here
```

Activity: Refactor Guessing Game

1. Make a directory to store the guessing game.
2. Copy the `guess.cpp` example into this directory.
3. Refactor the program into the following modules:
   - `guess.h, guess.cpp`
   - `score.h, score.cpp`
   - `main.cpp`
4. Add the following feature:
   After each game, ask the player if they want to play again.
   If they do, play again! (new number and all)

Compiling a Program with Multiple Files

```
g++ guess.cpp score.cpp main.cpp -o guess
```

Outline

1 Program Structure

2 The Build Process

Multi-Stage Compilation

- Compiling the entire source every time is quite time consuming.

Multi-Stage Compilation

- Compiling the entire source every time is quite time consuming.
- Instead we split the compilation into two parts:

Multi-Stage Compilation

- Compiling the entire source every time is quite time consuming.
- Instead we split the compilation into two parts:
  1. Compile `cpp` files.

Multi-Stage Compilation

- Compiling the entire source every time is quite time consuming.
- Instead we split the compilation into two parts:
  1. Compile `cpp` files.
  2. Link `cpp` files together.

Multi-Stage Compilation

- Compiling the entire source every time is quite time consuming.
- Instead we split the compilation into two parts:
  1. Compile `cpp` files.
  2. Link `cpp` files together.
- We can do this by adding the `-c` option to `g++`

Multi-Stage Compilation of the Guessing Game

Try the following sequence of commands:

```
g++ -c main.cpp
g++ -c guess.cpp
g++ -c score.cpp
g++ main.o guess.o score.o -o guess
```

Enter Make

- Linking object files is faster than compiling source files.

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.
- This is still a heavy workload!

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.
- This is still a heavy workload!
- This where the tool `make` comes in.

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.
- This is still a heavy workload!
- This where the tool `make` comes in.
- `make` lets us script the build process in an intelligent way.

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.
- This is still a heavy workload!
- This where the tool `make` comes in.
- `make` lets us script the build process in an intelligent way.
- `make` works by processing “recipes”.

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.
- This is still a heavy workload!
- This where the tool `make` comes in.
- `make` lets us script the build process in an intelligent way.
- `make` works by processing “recipes”.
- Recipes are either implicit or explicitly.

Implicit Recipes

- Make is scripted by creating a file named “`Makefile`”

Implicit Recipes

- Make is scripted by creating a file named “`Makefile`”
- In the `Makefile` we write a series of **recipes** in the following format:

  ```
  target:  ingredient list
  ```

Implicit Recipes

- Make is scripted by creating a file named “`Makefile`”
- In the `Makefile` we write a series of **recipes** in the following format:

  ```
  target:  ingredient list
  ```
- Make is “smart enough” to build some things without extra input.

Implicit Recipes

- Make is scripted by creating a file named “`Makefile`”
- In the `Makefile` we write a series of **recipes** in the following format:

  ```
  target:  ingredient list
  ```
- Make is “smart enough” to build some things without extra input.
- For instance, Create a new file called “`Makefile`” and enter the following:

  ```
  main.o: main.cpp guess.h score.h
  ```

Implicit Recipes

- Make is scripted by creating a file named "`Makefile`"
- In the `Makefile` we write a series of **recipes** in the following format:

  ```
  target:  ingredient list
  ```
- Make is "smart enough" to build some things without extra input.
- For instance, Create a new file called "`Makefile`" and enter the following:

  ```
  main.o: main.cpp guess.h score.h
  ```
- Now try the following commands:

  ```
      rm main.o
      make
  ```

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.
- For example, try the following:

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.
- For example, try the following:
  1. Modify your `Makefile` to read as follows:

```
guess: main.o guess.o score.o
    g++ main.o guess.o score.o -o guess
main.o: main.cpp guess.h score.h
guess.o: guess.cpp guess.h
score.o: score.cpp score.h
```

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.
- For example, try the following:
  1. Modify your `Makefile` to read as follows:
     ```
     guess: main.o guess.o score.o
         g++ main.o guess.o score.o -o guess
     main.o: main.cpp guess.h score.h
     guess.o: guess.cpp guess.h
     score.o: score.cpp score.h
     ```
- Remember that when indenting, you must use a literal tab character!

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.
- For example, try the following:
  1. Modify your `Makefile` to read as follows:

```
guess: main.o guess.o score.o
	g++ main.o guess.o score.o -o guess
main.o: main.cpp guess.h score.h
guess.o: guess.cpp guess.h
score.o: score.cpp score.h
```

- Remember that when indenting, you must use a literal tab character!
- Try running `make` now!

Some Predefined Variables

- The make syntax is itself a scripting language.

Some Predefined Variables

- The make syntax is itself a scripting language.
- Variables begin with dollar signs $.

Some Predefined Variables

- The make syntax is itself a scripting language.
- Variables begin with dollar signs $.
- There are several pre-defined variables, the two most commonly used ones are:

Some Predefined Variables

- The make syntax is itself a scripting language.
- Variables begin with dollar signs $.
- There are several pre-defined variables, the two most commonly used ones are:
  - `$@` – The name of the target

Some Predefined Variables

- The make syntax is itself a scripting language.
- Variables begin with dollar signs $.
- There are several pre-defined variables, the two most commonly used ones are:
  - `$@` – The name of the target
  - `$^` – The list of all ingredients

Some Predefined Variables

- The make syntax is itself a scripting language.
- Variables begin with dollar signs $.
- There are several pre-defined variables, the two most commonly used ones are:
  - `$@` – The name of the target
  - `$^` – The list of all ingredients
- We could simplify our `Makefile` like so:

```
guess: main.o guess.o score.o
    g++ $^ -o $@
main.o: main.cpp guess.h score.h
guess.o: guess.cpp guess.h
score.o: score.cpp score.h
```

User Defined Variables

- You can also define your own variables:

  ```
  TARGETS=guess
  ```

User Defined Variables

- You can also define your own variables:
  `TARGETS=guess`
- You refer to your own variables like this:
  `$(TARGETS)`

User Defined Variables

- You can also define your own variables:
  `TARGETS=guess`
- You refer to your own variables like this:
  `$(TARGETS)`
- This allows you to make compact makefiles.

Making The Program 5 Makefile

```
TARGETS=guess

#application builds
all: $(TARGETS)
guess: main.o guess.o score.o
    g++ $^ -o $@

#module builds
main.o: main.cpp guess.h score.h
guess.o: guess.cpp guess.h
score.o: score.cpp score.h

#delete all binaries
clean:
        rm -f *.o $(TARGETS)
```

Building With Make

- Run `make` to build the first recipe in the `Makefile`

Building With Make

- Run `make` to build the first recipe in the `Makefile`
- Run `make target` to build any other target.

Building With Make

- Run `make` to build the first recipe in the `Makefile`
- Run `make target` to build any other target.
- For example `make clean` runs the clean target.

Building With Make

- Run `make` to build the first recipe in the `Makefile`
- Run `make target` to build any other target.
- For example `make clean` runs the clean target.
- Each time you run `make`, it only does the minimal number of steps to complete the build!