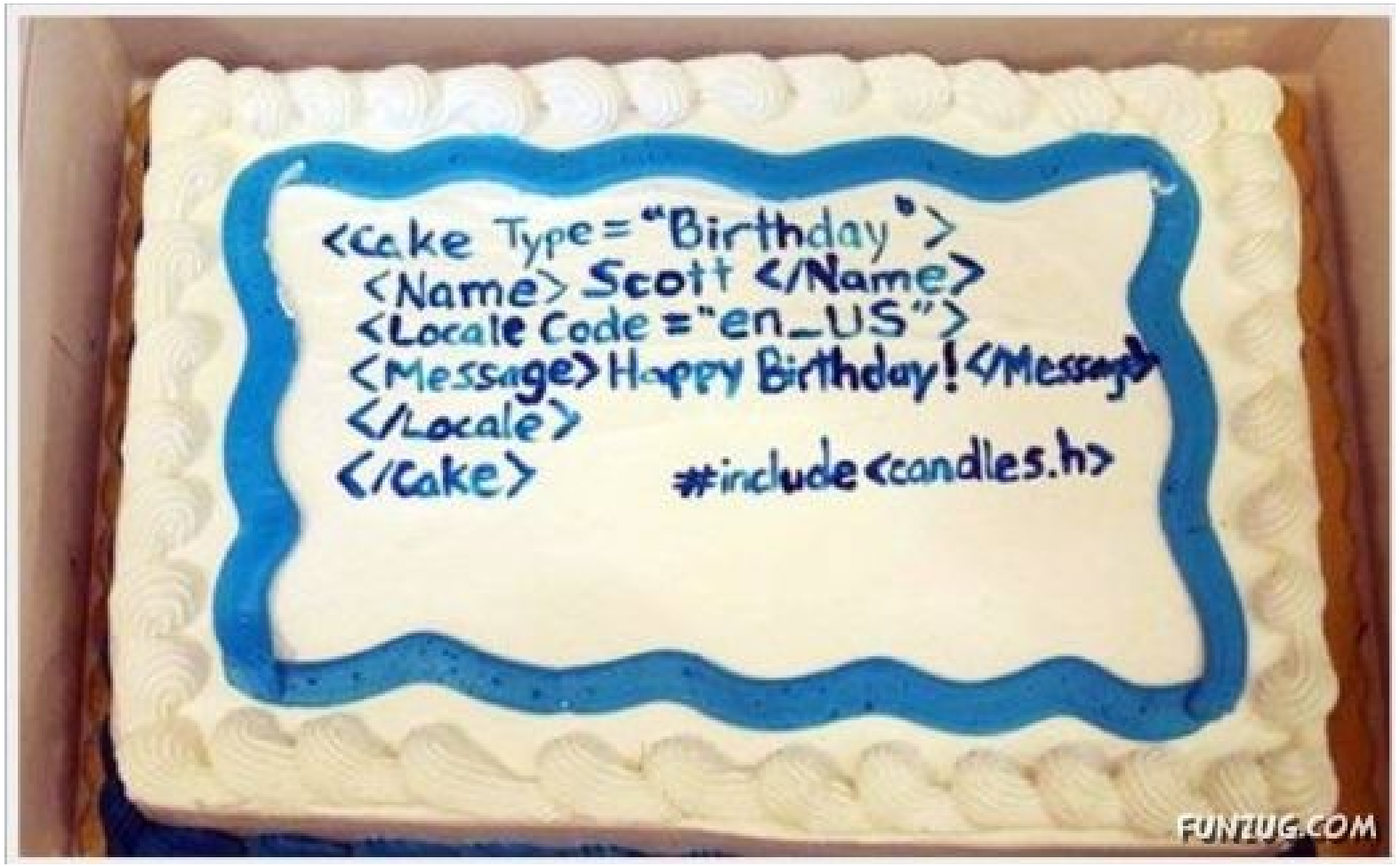


Algorithms & Pseudo Code

Due this week

- **Recitation**
  - Pseudocode practice
- **Syllabus Quiz** due last night! (you can turn it in late shhhhh….)
- **Homework 0**
  - Submit zip file on Canvas. (Late extension – turn in by 5:59 PM tonight)
- **Homework 1**
  - **Check due date on Canvas (Sunday)**
- 3-2-1 reflection
- Start going through the textbook readings and watch the videos
  - Take **Quiz 1** and **Quiz 2**. Check the due date!

Algorithms and Pseudo Code

Topics

1. What is programming?
2. Anatomy of a computer
3. Machine code and programming

Videos

4. Becoming familiar with your programming environment
5. Analyzing your first program
6. Errors

Today (briefly)

7. Problem solving: algorithm design → Today

Warm Up Activity

- Program me to eat the number of pistachios I’m hungry for
    - Write me instructions
- https://bit.ly/3XPCvef
- (Only open this when ready to copy/paste)
    - The last character is a lowercase “L”

Warm Up Activity

- What were the challenges of doing this activity?
- Where did you fail to communicate your intentions to the “computer” (me)? Why?

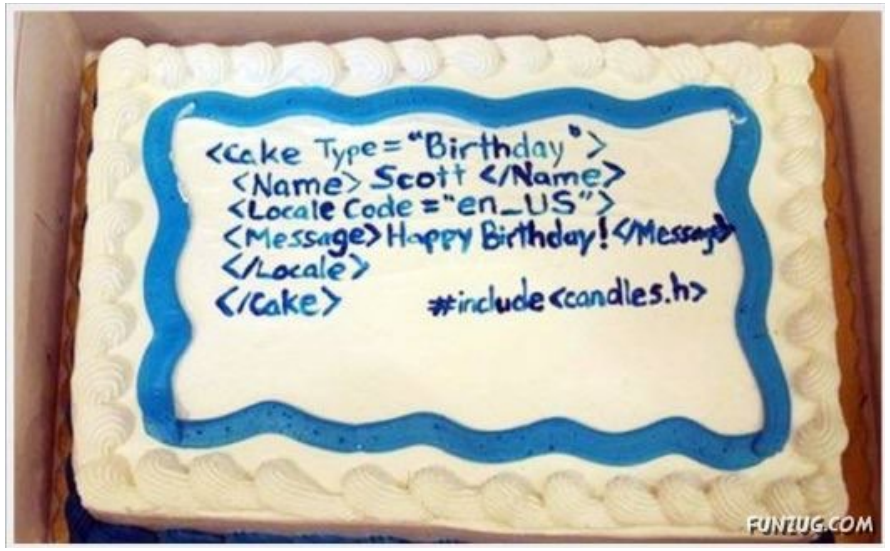
What am I trying to program?

- I was a human, and came with no documentation

- Fortunately, computers are very simple, and have documentation
  - Watch the videos!!

Algorithms

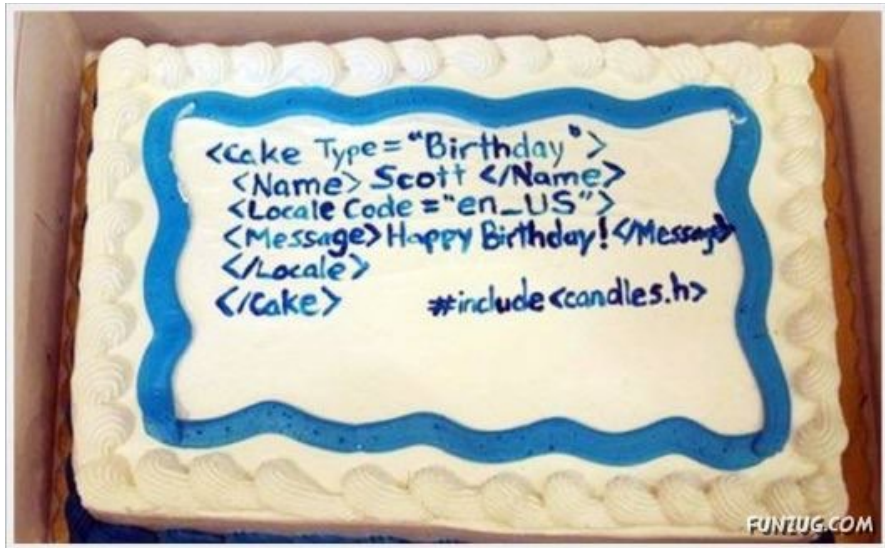
- Every program is based on an algorithm (or more)
- An algorithm is like a recipe for cooking
  - It tells the ingredients (*inputs*)
  - It tells the sequential steps for processing the inputs
  - It tells the serving size and style (*outputs*)
- The computer acts like a chef, exactly following the algorithm recipe



Algorithms

The computer acts like a chef, exactly following the algorithm recipe

- Chef Computer does not know the meaning of “a whole bunch of flour”
- Chef Computer also does not know that of course you don’t include the eggshells when the recipe calls for “two eggs”



Making a Cake

- What do we need to make the cake?
- Let’s assume you don’t have anything needed for making the cake.

High Level Abstraction for Making a Cake

Make a Cake:

- Drive to Store
- Buy Ingredients
- Drive Home
- Bake the Cake
- … and frost it like a nerd

This is the ***algorithm*** for making a cake.

It uses high level abstractions to make the algorithm easy to understand.

High Level Abstraction for Making a Cake

Make a Cake:

- Drive to Store
- Buy Ingredients
- Drive Home
- Bake the Cake
- … and frost it like a nerd

- Get into and Start the Car
- Drive to King Soopers
- Park the Car
- Turn the Car Off
- Get Out of Car

High Level Abstraction for Making a Cake

Make a Cake:

- Drive to Store
- Buy Ingredients
- Drive Home
- Bake the Cake
- … and frost it like a nerd

- Get into and Start the Car
- Drive to Home
- Park the Car
- Turn the Car Off
- Get Out of Car

High Level Abstraction for Making a Cake

Make a Cake:

- Drive to Store
- Buy Ingredients
- Drive Home
- Bake the Cake
- … and frost it like a nerd

→

- Get into and Start the Car
- Drive to DESTINATION
- Park the Car
- Turn the Car Off
- Get Out of Car

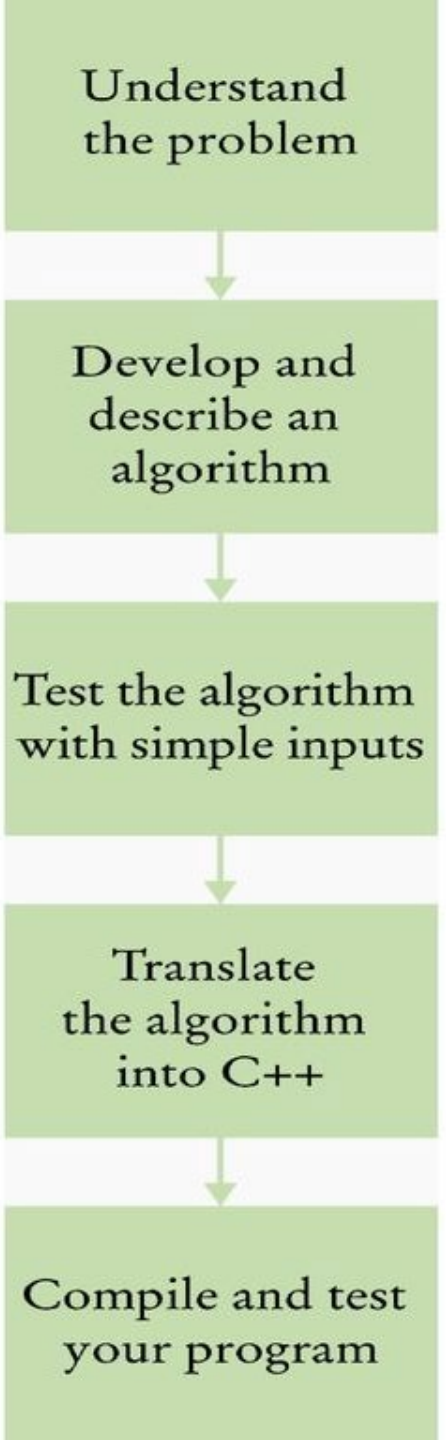
- we needed to drive to store and drive to home
- wouldn't it be nice to be able to solve the general problem of driving to … **destination?**

Input: initial location, destination

Output: sequence of steps

The Software Development Process

- For each problem the programmer goes through these steps
- ***You MUST write an algorithm in words, pictures, and/or equations before attempting to translate to C++***



6 Building Blocks for Computational Representations

1. Create a variable to store a value for later use
2. Modify the value of a variable
3. Get input or generate output
4. Check if a statement is True or False
5. Repeat a statement or collection of statements
6. *Encapsulating a collection of statements*

6 Building Blocks for Computational Representations

1. Create a variable to store a value for later use

What is a variable?

- Have you encountered variables before? Where?

Variables	Values or quantities that change over time
Range of a variable	What are all the possible values it could take?
Variable type	Numeric, text, other

Example story: Alexis is 18 y.o. and her grandma is approaching 80.

Inspired by the VELA project: http://csforall.sri.com/

6 Building Blocks for Computational Representations

1. Create a variable to store a value for later use

Examples:

lemons = 5

oranges = 4

fruit = lemons + oranges

celsius = 15

6 Building Blocks for Computational Representations

2. Modify the value of a variable

Examples:

```
lemons = 5

oranges = 4

fruit = lemons + oranges

fruit = fruit + bananas
```

```
celsius = 15

fahrenheit  = celsius * 9 / 5 + 32
```

6 Building Blocks for Computational Representations

3. Get input or generate output

Examples:

lemons = 5

oranges = 4

fruit = lemons + oranges

fruit = fruit + bananas

Print out the number of fruits

get the celsius value from user (and save the value entered by the user in variable *celsius)*

fahrenheit = *celsius* * 9 / 5 + 32

Print the fahrenheit value

6 Building Blocks for Computational Representations

4. Check if a statement is True or False

Examples:

lemons = 5
oranges = 4
fruit = lemons + oranges
fruit = fruit + bananas
Print out the number of fruits

if the number of fruits is larger than 10
    print “lets make a fruit salad”

get the celsius value from user
(and save the value entered by the user in variable *celsius)*

fahrenheit = *celsius* * 9 / 5 + 32
Print the fahrenheit value

If fahrenheit is less than or equal 32
    display “its freezing in here”

6 Building Blocks for Computational Representations

5. Repeat a statement or collection of statements

Examples:

```
lemons = 5
oranges = 4
fruit = lemons + oranges
fruit = fruit + bananas
Print out the number of fruits
```

```
if the number of fruits is larger than 10
    print “lets make a fruit salad”

for each fruit
    cut fruit into pieces

or

    While any piece of fruit is bigger than bite sized
        select largest piece of fruit
        cut selected piece of fruit into two pieces
```

Algorithms

- Step-by-step procedure for solving a problem or accomplishing some task

- When your algorithm has enough detail (it clearly informs how you will write your code), you are usually writing in *pseudo code*

Pseudo Code

pseu·do·code
ˈsōōdōˌkōd/

A notation resembling a simplified programming language for describing algorithms

- Intended for human readability, not a computer’s
- Does not need to be syntactically correct code
- Provides a language independent way to describe the steps of an algorithm

Algorithms

- Step-by-step procedure for solving a problem or accomplishing some task

- When your algorithm has enough detail (it clearly informs how you will write your code), you are usually writing in *pseudo code*

Describing an algorithm with Pseudocode (example 1)

Problem Statement:

You are asked to simulate a postage stamp vending machine. A customer inserts dollar bills into the vending machine, selects the number of stamps needed, and then pushes a “purchase” button. The vending machine gives out as many first-class stamps as the customer requested and can pay for, and returns the change in coins. A first-class stamp costs 55 cents. The machine is broken. The only available coins for change are dollar coins and pennies.

Describing an algorithm with Pseudocode

Step 1 Determine the **inputs** and **outputs**.

Inputs:

- *The amount of money* the customer inserts
- *The number of stamps* wished to purchase

Outputs:

- The number of stamps the machine returns
- The change:
  - The number of dollar coins
  - The number of pennies

Describing an algorithm with Pseudocode

Step 2 Break down the problem into smaller tasks

- Ask the user for input: how much money is inserted and how many stamps they wish to purchase
- Determine the total price
- Compute change value
  - Compute how many dollar coins and how many pennies

Describing an algorithm with Pseudocode

Step 3 Describe each subtask in pseudocode.

You will need to arrange the steps so that any intermediate values are computed before they are needed in other computations.

Describing an algorithm with Pseudocode

Step 3 Describe each subtask in pseudocode.

Ask user to input a whole number for the dollar amount inserted into the machine

Save in the variable *initial_money*

Ask user to input a whole number for the number of stamps wished to purchase

Save in the variable *num_stamps*

Compute total purchase price

*purchase_price_cents = num_stamps * 55*

Describing an algorithm with Pseudocode

Step 3 Describe each subtask in pseudocode.

Compute change needed

change = initial_money - purchase_price

Give change: … how do we give change?

Example:

initial_money = $5

num_stamps = 5

*purchase_price = 5 * $0.55 = $2.75*

change = $2.25

… which is $2 and 25 pennies

How can a C++ program come to the same conclusion?

Describing an algorithm with Pseudocode

Step 3 Describe each subtask in pseudocode.

Compute change needed
*change_cents = initial_money *100 - purchase_price_cents*

Example:

initial_money = $5
num_stamps = 5
*purchase_price_cents = 5 *55 = 275*
*change_cents = 5 * 100 – 275 = 225*

Describing an algorithm with Pseudocode

Step 3 Describe each subtask in pseudocode.

Give change:
change_dollars = change_cents / 100 (w/o remainder)

*change_pennies = change_cents – 100*change_dollars*

or

change_pennies = change_cents %100 (remainder),
where % is the **modulo** operator

Describing an algorithm with Pseudocode

Step 4 Test your pseudocode by working a problem.

Use these sample values:
Example 1:

initial_money = $5
num_stamps = 5
*purchase_price_cents = 5 * 55 = 275*
*change_cents = 5 * 100 – 275 = 225*
change_dollars = change_cents/100 = 2
change_pennies = change_cents%100 = 25

Pseudocode Reminders

Pseudocode

- Not a real programming language!
- Intended to:
  - Help you think through an algorithm before coding it
  - Communicate your ideas to other humans (it's "human readable")
- Algorithm is the abstract sequence of steps…
- Pseudocode is how we write it in human readable format
- Code is how we write it in computer understandable format

Time check

Your first program!

Taming your first program

Your first program

- The classic first program that everyone writes: Hello World!
  - (yes, everyone who is anyone started with this one)
- Its job is to write the words *Hello World!* on the screen.

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello, World!" << endl;
  return 0;
}
```

Your first program!

nobody:

beginner programmers:

the #include

- The first line tells the compiler to include a service for “stream input/output”. Later you will learn more about this but, for now, just know it is needed to write on the screen.

```
#include <iostream>

using namespace std;
int main()
{
	cout << "Hello, World!" << endl;
	return 0;
}
```

using namespace std

- The second line tells the compiler to use the “standard namespace”. This is used in conjunction with the <iostream> first line for controlling input and output.

```
#include <iostream>

using namespace std;
int main()
{
	cout << "Hello, World!" << endl;
	return 0;
}
```

int main()

- The next set of code *defines* a ***function,*** named **`main`.**
  - Every C++ program must contain its one `main` function.
  - All function names must be followed by parentheses. In `main`'s case, the parentheses are empty.
- Braces `{ }` must enclose all the code that belongs to `main`. The braces tell the compiler where to start reading the `main` code, and where to finish.

```
#include <iostream>

using namespace std;
int main()
{
	cout << "Hello, World!" << endl;
	return 0;
}
```

`cout` statement

- To show output on the screen, we use **`cout.`**
- What you want seen on the screen is “sent” to the **`cout`** entity using the **`<<`** operator (sometimes called the insertion operator): **`<< "Hello, World!"`**
- **The curious non-word `endl` means end-of-line, which tells the display to move the cursor down to the start of the next line.**

```
#include <iostream>

using namespace std;
int main()
{
	cout << "Hello, World!" << endl;
	return 0;
}
```

`return` statement

- The **`main`** function "returns" an "integer" (that is, a whole number without a fractional part, called **`int`** in C++) with value 0.
- This value indicates that the program finished successfully.

```
#include <iostream>

using namespace std;
int main()
{
	cout << "Hello, World!" << endl;
	return 0;
}
```

Semicolons are Required after Statements

- Each statement in C++ ends in a semicolon;
  - Note that not every line in a program is a statement, so there are no semicolons after the `<iostream>` line and the `main()` line
  - It is a strange idiosyncrasy, but you will get used to it

```
#include <iostream>

using namespace std;
int main()
{
	cout << "Hello, World!" << endl;
	return 0;
}
```

Output Statements and Streaming Operator <<

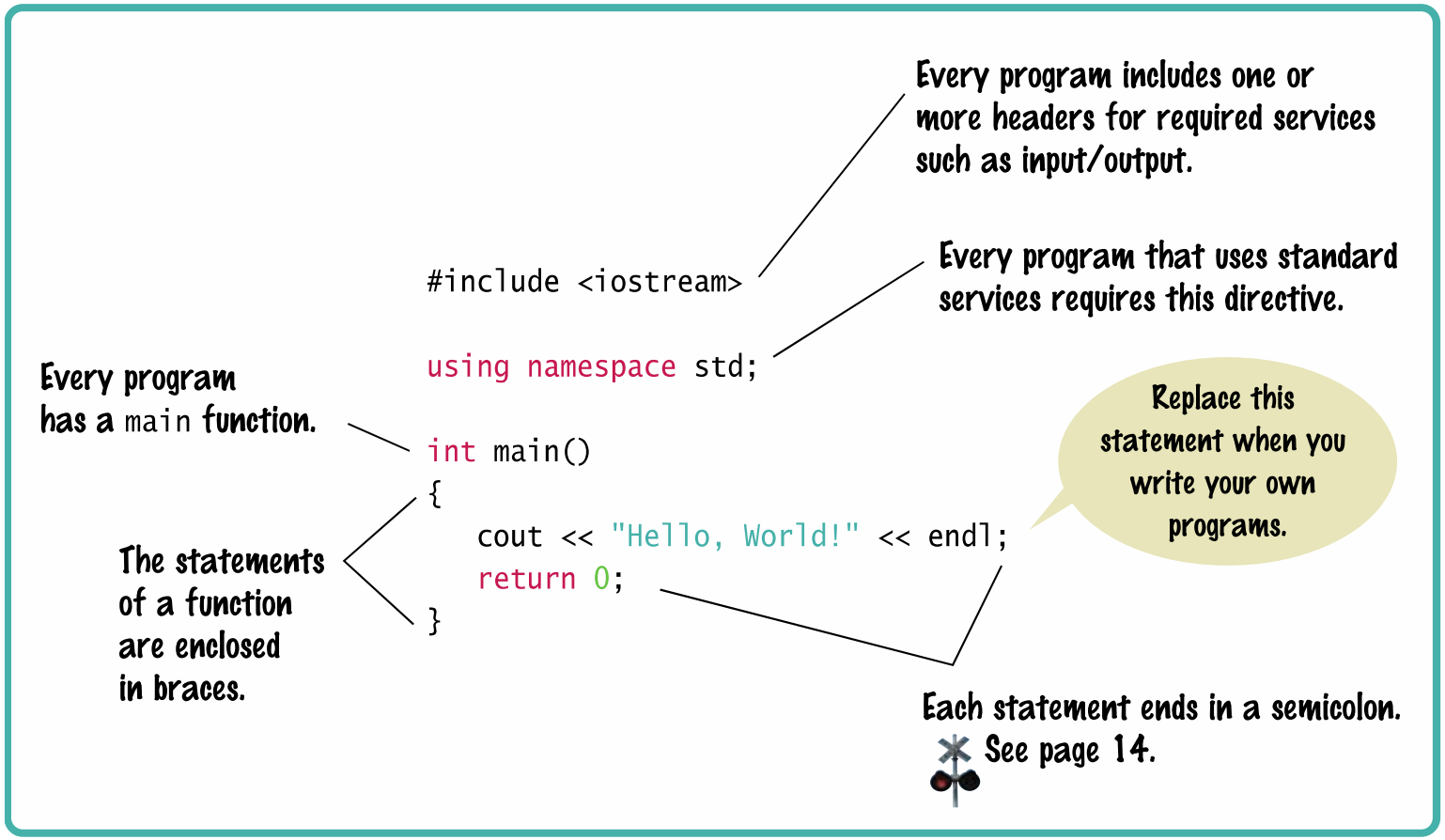
The statement

```
cout << "Hello World!" << endl;
```

is an *output statement*.

- To display values on the screen, you send them to an entity called cout.
  - Which stands for "character output" or "console output".

- The << operator denotes the “send to” command.



The reason why world never says hello back!!


EVERYONE SAYS
HELLO WORLD
NO ONE ASKS
HOW IS WORLD?

Errors!

Common Error – Omitting Semicolons errors

Omitting a semicolon (or two), in this case at the end of the `cout` statement

```
#include <iostream>

using namespace std;
int main()
{
	cout << "Hello, World!" << endl
	return 0;
}
```

Syntax errors

Without that semicolon you actually wrote:

```
cout << "Hello, World!" << endl return 0;
```

which thoroughly confuses the compiler with the `endl` immediately followed by the `return`!

- This is a *compile-time error* or *syntax error*.
- A syntax error is a part of a program that does not conform to the rules of the programming language.

Errors: Misspellings

- Suppose you (accidentally of course) wrote:

```
cot << "Hello World!" << endl;
```

- This will cause a compile-time error and the compiler will complain that it has no clue what you mean by cot.
- The exact wording of the error message is dependent on the compiler, but it might be something like

“Undefined symbol cot” or "Unknown identifier".

How many errors?

- The compiler will not stop compiling, and will most likely list lots and lots of errors that are caused by the first one it encountered.
- You should fix only those error messages that make sense to you, **starting with the first one**, and then recompile (after SAVING, of course!).

Logic Errors

Consider this:

```
cout << "Hollo, World!" << endl;
```

- *Logic errors* or *run-time errors* are errors in a program that compiles (the syntax is correct), but executes without performing the intended action.
- The programmer must thoroughly inspect and test the program to guard against logic errors.
  - *Testing and repairing a program usually takes more time than writing it in the first place, but is essential !*

Errors: Run-Time Exceptions

Some kinds of run-time errors are so severe that they generate an *exception*: a signal from the processor that aborts the program with an error message.

For example, if your program includes the statement

```
cout << 1 / 0;
```

Your program may terminate with a “divide by zero” exception.

Errors: extra or misspelled main() function

- Every C++ program must have one and only one **`main`** function.
- Most C++ programs contain other functions besides **`main`** (more about functions next week).

Errors: C++ is Case Sensitive

C++ is *case sensitive.* Typing:

```
int Main()
```

will compile but will not link.

A link-time error occurs here when the linker cannot find the **`main`** function – because you did not define a function named **`main`**. (**`Main`** is fine as a name but it is not the same as **`main`** and there has to be one **`main`** somewhere.)

If you want to learn more about the build process, read this. The content in this webpage is not a part of the syllabus and will not be on any course related assignments.

Making your Program Readable (by Humans)

C++ has *free-form layout*

```
int main(){cout<<"Hello, World!"<<endl;return 0;}
```

- *will* compile (but is practically impossible to read)

A good program is readable:

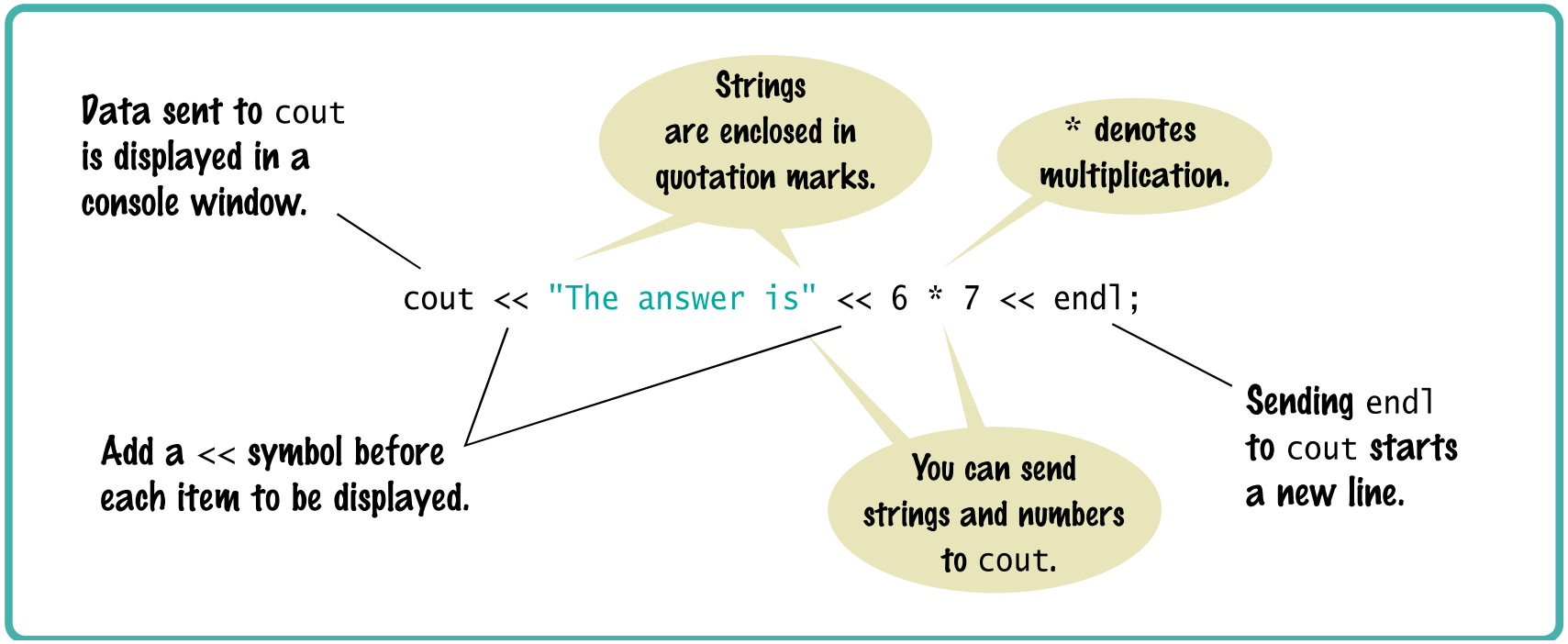
- code spaced across multiple lines, one statement per line
- follows indentation conventions

"Strings" and `endl`

```
cout << "Hello World!" << endl;
```

- **`"Hello World!"`** is called a *`string`*.
- You must put those double-quotes around strings.

- The **`endl`** symbol denotes an *end of line* marker which causes the cursor to move down to the next screen line.



Pseudocode Example Continued (optional slides)

Describing an algorithm with Pseudocode

Step 4 Test your pseudocode by working a problem.

Use these sample values:
Example 2:

initial_money = $5
num_stamps = 7
*purchase_price_cents = 7 * 55 = 385*
*change_cents = 5 * 100 – 385 = 115*
change_dollars = change_cents/100 = 1
change_pennies = change_cents%100 = 15

Describing an algorithm with Pseudocode

Step 4 Test your pseudocode by working a problem.

Are we ready to implement it into code? Have we thought of all possibilities?

Example 3:

initial_money = $5
num_stamps = 17
*purchase_price_cents = 17 * 55 = 935*
*change_cents = 5 * 100 – 935 = - 435*
change_dollars = change_cents/100 = ?
change_pennies = change_cents%100 = ?

Describing an algorithm with Pseudocode

Step 4 Test your pseudocode by working a problem.

Are we ready to implement it into code? Have we thought of all possibilities?

Example 4:

initial_money = $5
num_stamps = -3
purchase_price_cents = …
change_cents = …
change_dollars = change_cents/100 = …
change_pennies = change_cents%100 = …

Describing an algorithm with Pseudocode

Step 4 Test your pseudocode by working a problem.

Are we ready to implement it into code? Have we thought of all possibilities?

Example 5:

initial_money = $5
num_stamps = r
purchase_price_cents = …
change_cents = …
change_dollars = change_cents/100 = …
change_pennies = change_cents%100 = …

Describing an algorithm with Pseudocode

Step 3 Describe each subtask in pseudocode. **Make changes!**

Ask user to input a whole positive number for the dollar amount inserted into the machine

Save in the variable *initial_money*

Ask user to input a whole positive number for the number of stamps wished to purchase

Save in the variable *num_stamps*

Compute total purchase price

*purchase_price_cents = num_stamps * 55*

Describing an algorithm with Pseudocode

Step 3 Describe each subtask in pseudocode. **Make changes!**

If *purchase_price_cents <= initial_money*100*
    then, Compute change needed
    *change_cents = initial_money *100 - purchase_price_cents*
Otherwise
    print “Not enough money”

Give change:
change_dollars = change_in_cents / 100 (w/o remainder)
change_pennies = change_in_cents %100 (remainder)

Is this correct?

NO!

Describing an algorithm with Pseudocode

Step 3 Describe each subtask in pseudocode. **Make changes!**

If *purchase_price_cents <= initial_money*100*

then, Compute change needed

*change_cents = initial_money *100 - purchase_price_cents*

Give change:

change_dollars = change_in_cents / 100 (w/o remainder)

change_pennies = change_in_cents %100 (remainder)

Otherwise

print “Not enough money”

Questions?