



# Taming your first program

# Due this week

---

- **Syllabus Quiz** due tonight!
- **Homework 0**
  - Install VS Code
  - Tutorials and videos on Canvas, based on the operating system of your computer
  - Submit zip file on Canvas.
- Check the due dates on Canvas!

# Today

---

- Explore our IDE: VS Code
- Write our first program
- Pseudocode
- Variables in C++

# Let's look at our IDE!

---

# 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;
}
```

# 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 <<

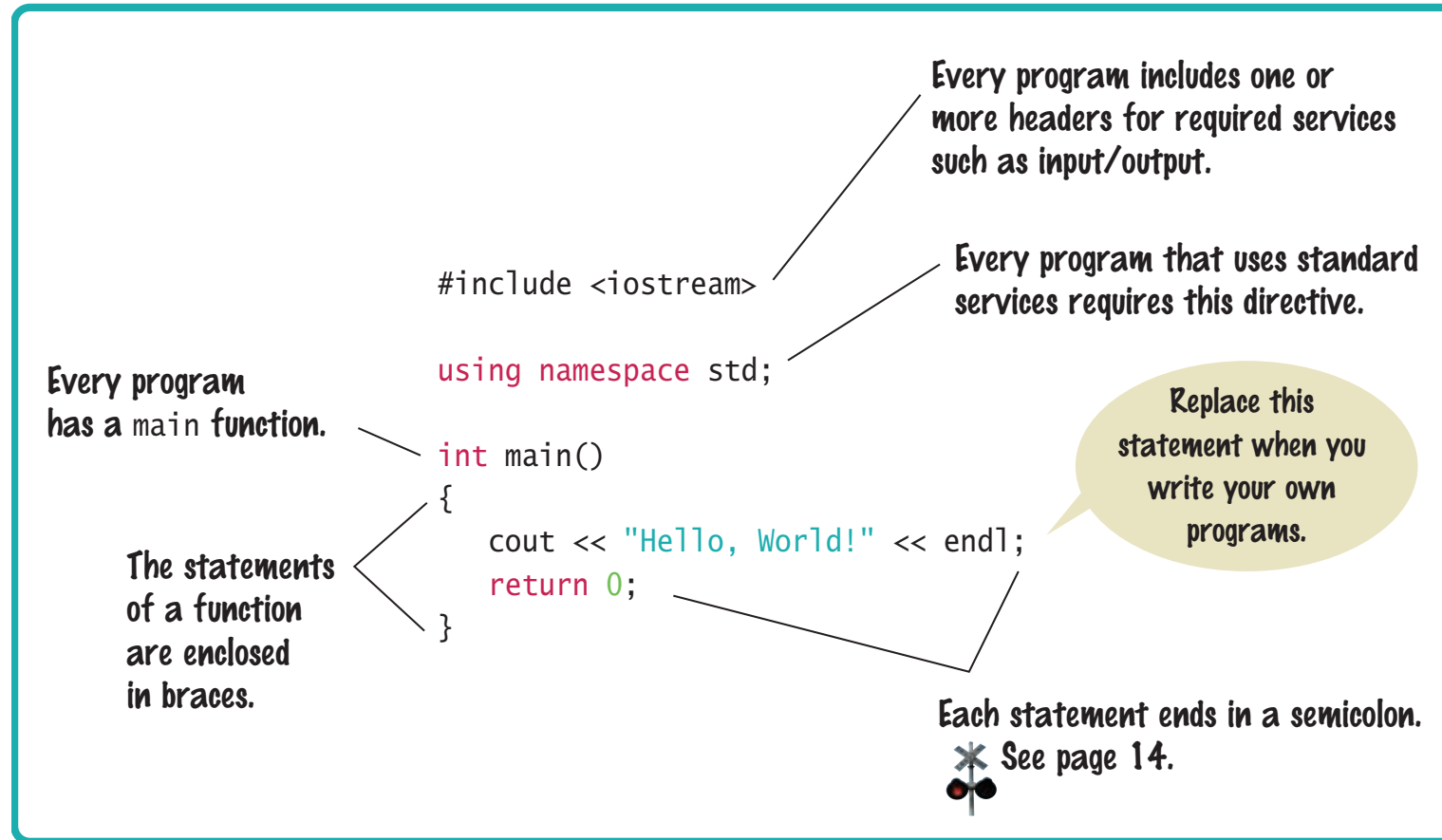
---

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.



# 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.

Data sent to `cout` is displayed in a console window.

Strings are enclosed in quotation marks.

\* denotes multiplication.

```
cout << "The answer is" << 6 * 7 << endl;
```

Add a `<<` symbol before each item to be displayed.

You can send strings and numbers to `cout`.

Sending `endl` to `cout` starts a new line.

# Variables

# 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  
'sōdō,kōd/

---

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

# 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.

# 6 Building Blocks for Computational Representations

---

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

Examples:

lemons = 5

celsius = 15

oranges = 4

fruit = lemons + oranges



# 6 Building Blocks for Computational Representations

---

## 2. Modify the value of a variable

Examples:

lemons = 5

celsius = 15

oranges = 4

fahrenheit = celsius \* 9 / 5 + 32

fruit = lemons + oranges

fruit = fruit + bananas

# 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

# Variables

---

## A **variable**:

- is used to **store** information (the **value/contents** of the variable)
  - can contain one piece of information at a time.
- has an **identifier** (the name of the variable)
- The programmer picks a good name
  - A good name describes the contents of the variable or what the variable will be used for
  - has a **type** (more about this very soon)

# Variables: Like a parking garage

---

- Parking garages store cars.
- Each parking space is identified
  - like a variable's identifier
- Each parking space “contains” a car
  - like a variable's current contents
- Each space can contain only one car
- and not trucks or buses, just a car



# Variable Definitions

---


- When creating variables, the programmer specifies the **type** of information to be stored.
- Unlike a parking space, a variable is often given an initial value.
  - ***Initialization*** is putting a value into a variable when the variable is created.
  - Initialization is **not required**.

Types introduced in this chapter are the number types `int` and `double` (page 32) and the string type (page 57).

```
int cans_per_pack = 6;
```

See page 33 for rules and examples of valid names.

A variable definition ends with a semicolon.

Use a descriptive variable name.  
 See page 38.

Supplying an initial value is optional, but it is usually a good idea.



See page 37.

## Variable Definitions

# Variable Definitions: example

---

The following statement defines a variable:

```
int cans_per_pack = 6;
```

**cans\_per\_pack** is the variable's name.

**int** indicates that the variable **cans\_per\_pack** will hold integers. Other variable types covered later will hold *strings* and *floating-point numbers*.

**= 6** indicates that the variable **cans\_per\_pack** will initially contain the value 6.

**Like all statements, it must end with a semicolon.**

# The Assignment Statement

---

- The contents in variables can “vary” over time (hence the name!).
- Variables can be changed by
  - assigning to them
    - **The assignment statement (“=”)**
  - using the increment or decrement operator (++ , --)
  - inputting into them
    - The input statement (“cin”)



# Assignment Statement Example

---

- An *assignment statement* stores a new value in a variable, replacing the previously stored value.

**cans\_per\_pack = 8;**

- This assignment statement changes the value stored in **cans\_per\_pack** to be 8.
- The previous value is replaced.

# The Meaning of the Assignment = Symbol

---

- The = in an assignment does ***not*** mean the left hand side is equal to the right hand side as it does in math.
- = is an instruction to do something:
  - copy*** the value of the expression on the right
  - into*** the variable on the left.
- Consider what it would mean, mathematically, to state:  
**counter = counter + 2;**

counter *EQUALS* counter + 2

# Assignment Statement: defining vs. assigning

---

- There is an important difference between a variable definition and an assignment statement:

```
int cans_per_pack = 6; // Variable definition
```

```
...
```

```
cans_per_pack = 8; // Assignment statement
```

- The first statement is the *definition* of `cans_per_pack`.
- The second statement is an *assignment statement*.
  - An *existing* variable's contents are replaced.
- A variable's definition must occur **only once** in a program. The same variable may be in several assignment statements in a program.

# Assignment Examples

---

```
counter = 11; // set counter to 11  
counter = counter + 2; // increment
```

1. First statement assigns 11 to counter
2. Second statement looks up what is currently in the variable counter (11)
3. Then it adds 2 and copies the result of the addition into the variable on the left, changing counter to 13

# Variable Definitions: more examples

---

Table 1: Variable Definitions in C++	
	Comment
<code>int cans = 6;</code>	Defines an integer variable and initializes it with 6.
<code>int total = cans + bottles;</code>	The initial value need not be a constant. (Of course, cans and bottles must have been previously defined.)
<code>int bottles = "10";</code>	Error: You cannot initialize an int variable with a string.
<code>int bottles;</code>	Defines an integer variable without initializing it. This can be a cause for errors—see Common Error 2.2.
<code>int cans, bottles;</code>	Defines two integer variables in a single statement. In this book, we will define each variable in a separate statement.
<code>bottles = 1;</code>	Caution: The type is missing. This statement is not a definition but an assignment of a new value to an existing variable—see Section 2.1.4.

Table 2: Number Literals		
	Type	Comment
6	int	An integer has no fractional part.
−6	int	Integers can be negative.
0	int	Zero is an integer.
0.5	double	A number with a fractional part has type double.
1.0	double	An integer with a fractional part .0 has type double.
1E6	double	A number in exponential notation: $1 \times 10^6$ or 1000000. Numbers in exponential notation always have type double.
2.96E-2	double	Negative exponent: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$
100,000		Error: Do not use a comma as a decimal separator.
3 1/2		Error: Do not use fractions; use decimal notation: 3.5.

Table 3: Variable Names	
Variable Name	Comment
can_volume1	Variable names consist of letters, numbers, and the underscore character.
x	In mathematics, you use short variable names such as x or y. This is legal in C++, but not very common, because it can make programs harder to understand (see Programming Tip 2.1)
Can_volume	Caution: Variable names are case sensitive. This variable name is different from can_volume.
6pack	Error: Variable names cannot start with a number.
can volume	Error: Variable names cannot contain spaces.
double	Error: You cannot use a reserved word as a variable name.
ltr/fl.oz	Error: You cannot use symbols such as . or /

# Common Error: Using Undefined Variables

---

You must define a variable before you use it for the first time.

For example, the following sequence of statements would not be legal:

```
double can_volume = 12 * liter_per_ounce;  
double liter_per_ounce = 0.0296;
```

Statements are compiled in top to bottom order.

When the compiler reaches the first statement, it does not know that **liter\_per\_ounce** will be defined in the next line, and it reports an error.



# Common Error: Using Uninitialized Variables

---

- Initializing a variable is not required, but there is always a value in every variable, even uninitialized ones.
- Some value will be there, left over from some previous calculation or simply the random value there when the transistors in RAM were first turned on.

```
int bottles; // Forgot to initialize  
int bottle_volume = bottles * 2;
```

What value would be output from the following statement?

```
cout << bottle_volume << endl;
```

# Comments

---

- *Comments* are explanations for human readers of your code (other programmers or your instructor).
- The compiler ignores comments completely.
- A leading double slash `//` tells the compiler the remainder of this line is a comment, to be ignored
- For example,

```
double can_volume = 0.355; // Liters in a 12-ounce can
```

# Comments: `//` or `/* multi-line */`

---

Comments can be written in two styles:

- Single line:

```
double can_volume = 0.355; // Liters in a 12-ounce can
```

The compiler ignores everything after `//` to the end of line

- Multiline for longer comments, where the compiler ignores everything between `/*` and `*/`

```
/*  
    This program computes the volume (in liters)  
    of a six-pack of soda cans.  
*/
```

# Next time

---

- Arithmetic
- Input statement

# Questions?