



## Lecture 4: Taming Your First Program



# Announcements and reminders

---

## Submissions:

- HW 1 (picobot) -- due Wednesday 24 Jan at 6 PM
- HW 2 (pseudocode) -- due Saturday 27 Jan at 6 PM

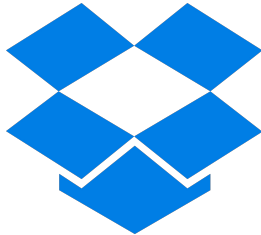
## Course reading to stay on track:

- 1.7-2.3 for today
- 2.3-2.4 for Friday
- Ch. 5: Functions for next week (see calendar)



# Back-up your work!

- GitHub (make it a **private** repository)
- Google Drive
- Dropbox
- You might want to revert to an earlier version
- Your computer might crash
- You might need to access your assignment from another computer
- **Use cloud storage - always save your work in more than one place!**

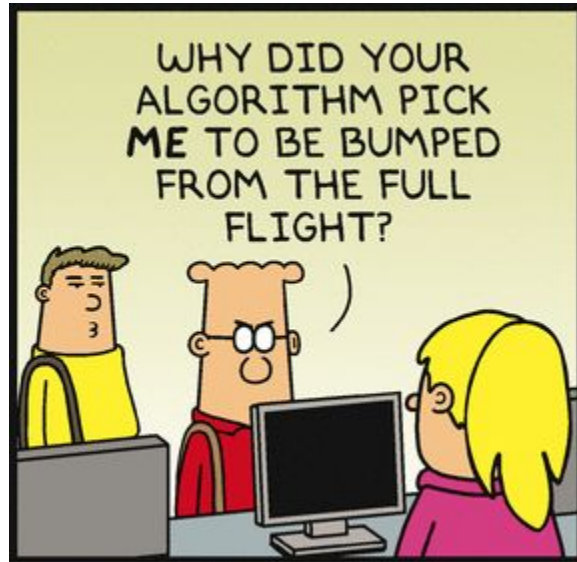


Don't care how. **No extensions** if you find that your assignment is lost to the ether the day before it is due.

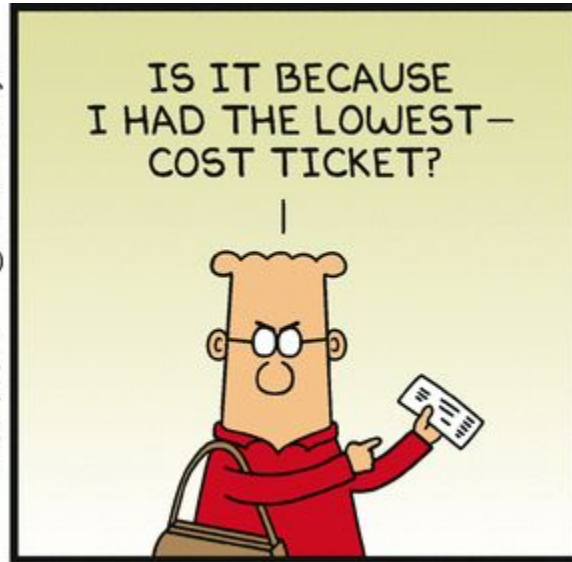


## Last time on *Starting Computing...*

- We talked **pseudocode**
- We learned what the heck an **algorithm** is



@ScottAdamsSays  
Dilbert.com



5-18-17 © 2017 Scott Adams, Inc./Dist. by Andrews McMeel



# Your first program

---

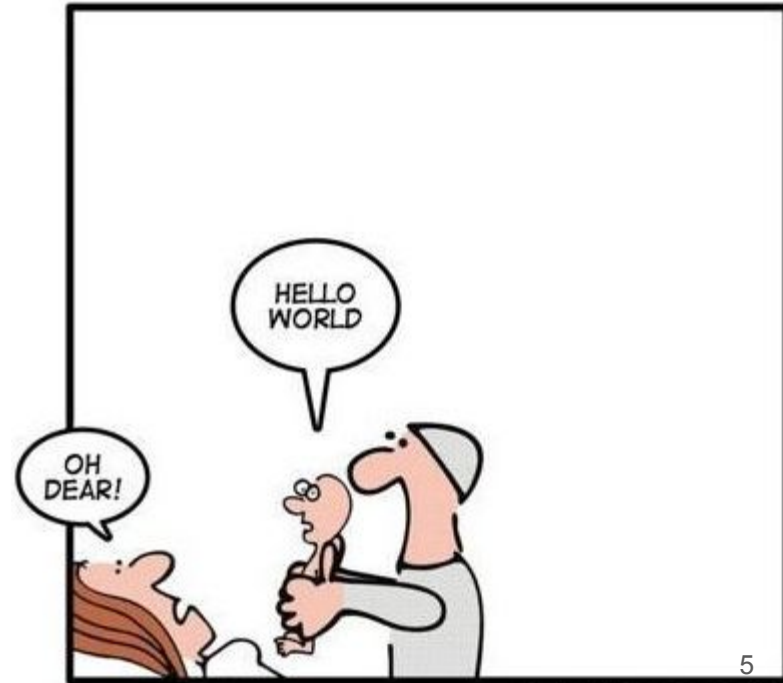
The classic first program everyone writes: Hello world!

Its job is to write the message “Hello world!” to the screen

Before we dive in, let’s all code it together!

You can code stuff however you want. Working in the Cloud9 IDE (integrated development environment)

- 1) <https://ide.c9.io/>
- 2) Sign into your Cloud9 account
- 3) Create a C++ workspace
  - a) share w/ TA, and make sure it is **private**
- 4) Create folder for lecture # \_\_\_\_ or HW # \_\_\_\_  
For example: lec04\_examples  
(to organize our work)



# Your first program

---

The classic first program everyone writes: Hello world!

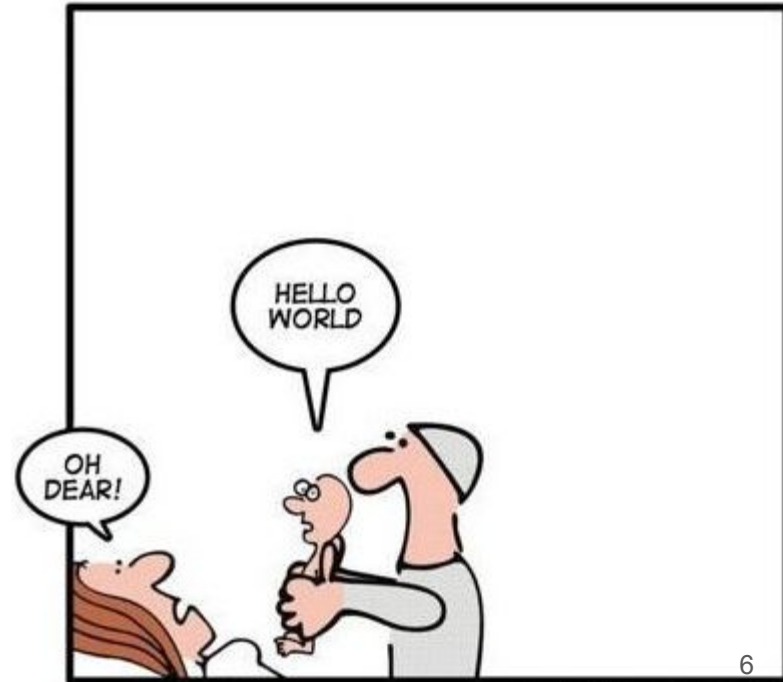
Its job is to write the message “Hello world!” to the screen

It looks like this:

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

Crystal clear right?

... Of course not. Let's digest this a bit more.



# Your first program

**#include <iostream>** -- tells compiler to include service for “stream input/output” (need for writing output to the screen)

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

# Your first program

**#include <iostream>** -- tells compiler to include service for “stream input/output” (need for writing output to the screen)

**using namespace std;** -- tells compiler to use the standard (“std”) namespace. Along with iostream, this lets us control input/output

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



# Your first program

**#include <iostream>** -- tells compiler to include service for “stream input/output” (need for writing output to the screen)

**using namespace std;** -- tells compiler to use the standard (“std”) namespace. Along with iostream, this lets us control input/output

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

**int main()** -- this is a **function** called main

- Every C++ program must contain one main function
- All function names are followed by parentheses. In main’s case, these parentheses are empty
- Curly braces { } surround the code that belongs to main. These tell the compiler where to start reading the main code and where to end.

# Your first program

**#include <iostream>** -- tells compiler to include service for “stream input/output” (need for writing output to the screen)

**using namespace std;** -- tells compiler to use the standard (“std”) namespace. Along with iostream, this lets us control input/output

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

**int main()** -- this is a **function** called main

- Every C++ program must contain one main function
- All function names are followed by parentheses. In main’s case, these parentheses are empty
- Curly braces { } surround the code that belongs to main. These tell the compiler where to start reading the main code and where to end.

**cout** statement -- shows output on the screen (“console output”)

- What you want to print to screen (“Hello world!”) is sent to cout entity using the << insertion operator
- endl tells the compiler “end-of-line”

# Your first program

**#include <iostream>** -- tells compiler to include service for “stream input/output” (need for writing output to the screen)

**using namespace std;** -- tells compiler to use the standard (“std”) namespace. Along with iostream, this lets us control input/output

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

**int main()** -- this is a **function** called main

- Every C++ program must contain one main function
- All function names are followed by parentheses. In main’s case, these parentheses are empty
- Curly braces { } surround the code that belongs to main. These tell the compiler where to start reading the main code and where to end.

**return** statement -- returns as fcn output an integer with value 0. Indicates program finished successfully

**cout** statement -- shows output on the screen (“console output”)

- What you want to print to screen (“Hello world!”) is sent to cout entity using the << insertion operator
- endl tells the compiler “end-of-line”

# Your first program

---

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

## Other notes:

- Every program includes one or more headers for required services (e.g., input/output)
- Every program using standard services needs the namespace std directive
- **Every** program has a main function
- The statements of a function are **always** enclosed in braces (curly brackets { } )
- This line is the “meat and cheese” of your program. To do other things, replace it with other codes!
- Every statement ends in a semicolon ;  
(So compiler knows where lines begin/end)

# Output statements -- the 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` (console/character output)
- The `<<` operator is called the ***streaming operator*** and is the computery way of saying "send to"
  - Can send **strings** and **numbers** to `cout`



# Output statements -- strings and endl

---

The statement `cout << "Hello world!" << endl;` is an **output statement**.

- To display values on the screen, you send them to an entity called `cout` (console/character output)
- The `<<` operator is called the **streaming operator** and is the computery way of saying "send to"
  - Can send **strings** and **numbers** to `cout`
- "Hello world!" is called a **string** (or **character string**).
  - Double quotes are needed to denote a char string
- `endl` symbol denotes the end of the line
  - Moves cursor down to next line (line break in output)




## Output statements -- printing multiple items

---

Can display more than one thing by chaining or *streaming* multiple copies of the << operator into the same statement.

**Example:** S'pose we want to print the message “Hello cruel science world!” to the screen, but only one word at a time.

```
#include <iostream>
using namespace std;
int main()
{
    
    return 0;
}
```

## Output statements -- printing multiple items

---

Can display more than one thing by chaining or *streaming* multiple copies of the << operator into the same statement.

**Example:** S'pose we want to print the message “Hello cruel science world!” to the screen, but only one word at a time.

```
#include <iostream>
using namespace std;
int main()
{
    cout << “Hello ” << “cruel ” << “science ” << “world!” << endl;
    return 0;
}
```

**Notice anything weird-looking?**




## Output statements -- printing multiple items

---

Can display more than one thing by chaining or *streaming* multiple copies of the << operator into the same statement.

**Example:** S'pose we want to print the message “A big number is [37\*41]” to the screen, where [37\*41] is replaced by us actually computing the product of 37 and 41.

```
#include <iostream>
using namespace std;
int main()
{
    
    return 0;
}
```

## Output statements -- printing multiple items

---

Can display more than one thing by chaining or *streaming* multiple copies of the << operator into the same statement.

**Example:** S'pose we want to print the message “A big number is [37\*41]” to the screen, where [37\*41] is replaced by us actually computing the product of 37 and 41.

```
#include <iostream>
using namespace std;
int main()
{
    cout << “A big number is ” << 37*41 << endl;
    return 0;
}
```

# Semicolons -- tough in English, easier in Comp Sci

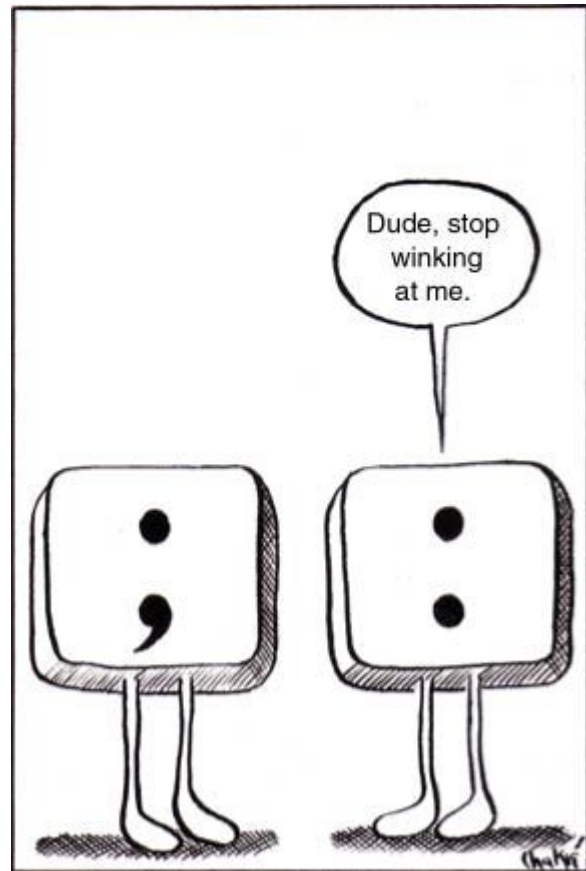
Each statement in C++ ends in a semicolon ;

Not every line in a program is a statement

(e.g., no semicolons after the `<iostream>` line and the `main()` line)

Kind of weird, but you'll get used to it. Your compiler will yell at you anyway if you're missing some.

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



## Common errors -- omitting semicolons

Each statement in C++ ends in a semicolon ;

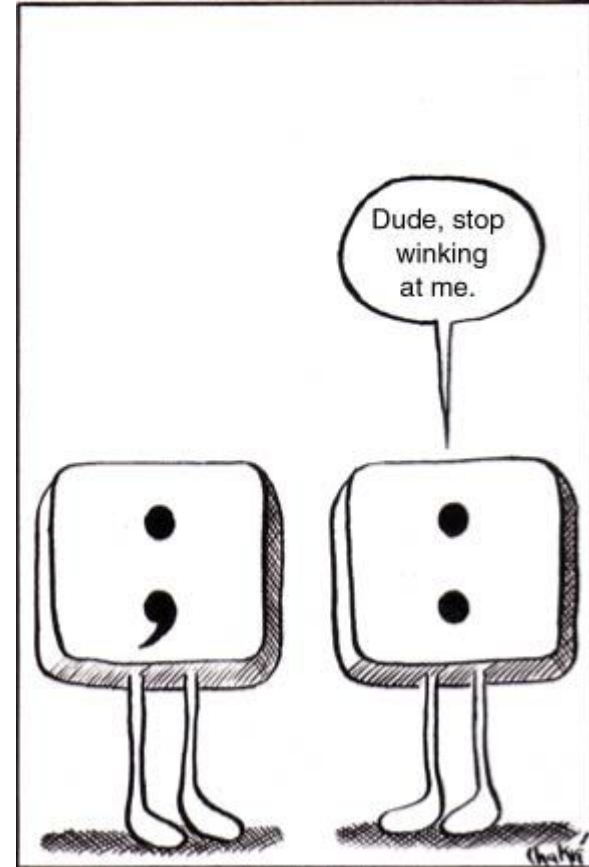
Not every line in a program is a statement

(e.g., no semicolons after the `<iostream>` line and the `main()` line)

Kind of weird, but you'll get used to it. Your compiler will yell at you anyway if you're missing some.

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

**What happens if we screw  
this up??**



## Common errors -- omitting semicolons

Without the semicolon we actually wrote:

```
cout << "Hello world!" << endl return 0;
```

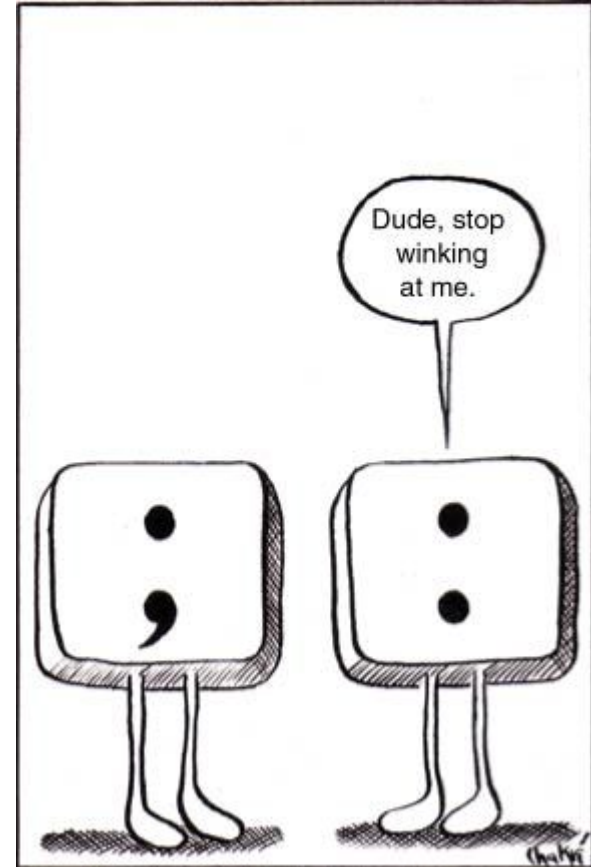
The "endl" immediately followed by "return" confuses the compiler

→ Gives an error:

```
helloworld.cpp:5:35: error: expected ';' after expression
    cout << "Hello world!" << endl
                                ^
                                ;
1 error generated.
```

→ This is a **compile-time/syntax error**

**Definition:** A **syntax error** is a part of a program that does not conform to the rules of the programming language.



## Common errors -- misspellings

---

S'pose you accidentally wrote:

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

The compiler will complain that it has no clue what you mean by “cot”

→ Gives a compile-time error:

```
helloworld.cpp:5:5: error: use of undeclared identifier 'cot'; did
    you mean 'cout'?
    cot << "Hello world!" << endl;
    ^~~
    cout
```

The exact message and how helpful it is will depend on your compiler. (I used g++ here.)

But it should say something like “Unknown identifier” or “Undefined symbol cot”,  
or “‘cot’ was not declared”

## Errors, errors, everywhere!

---

Compiler will not stop compiling, and probably will spit out lots and lots of errors that were caused by the first error.

Start debugging by **(1) starting from the beginning of the program** and  
**(2) starting with the most obvious/sensible errors.**

**SAVE** your work after each fix, just in case fixing your code actually breaks it worse.



## Logic errors

S'pose you accidentally wrote:

```
cout << "Hell world!" << endl;
```

Will compile and run just fine, but is probably not what you meant.



**Definition:** Logic/run-time errors are errors in a program that compiles (syntax is correct), but executes without performing the intended action.



## Logic errors

S'pose you accidentally wrote:

```
cout << "Hell world!" << endl;
```

Will compile and run just fine, but is probably not what you meant.



**Definition:** Logic/run-time errors are errors in a program that compiles (syntax is correct), but executes without performing the intended action.

- Your compiler will not protect you against logic errors!
- The programmer must thoroughly inspect and test the code to guard against logic errors
- Testing and debugging a program usually takes more time than writing it in the first place, but **it is essential.**

## Errors: run-time exceptions

**Definition:** Some kinds of 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 contains the statement:

```
cout << "A crazy number is " << 1 / 0 << endl;
```

then your program will generate a divide-by-zero exception:

```
helloworld.cpp:7:39: warning: division by zero is undefined
      [-Wdivision-by-zero]
      cout << "A crazy number is " << 1 / 0 << endl;
                                   ^ ~
1 warning generated.
```

## More errors

---

### Extra or misspelled `main()` function

- Every C++ program must have exactly one `main` function
- Many C++ programs contain other functions besides `main` (more on this next week!)

## More errors

---

### Extra or misspelled main() function

- Every C++ program must have exactly one main function
- Many C++ programs contain other functions besides main (more on this next week!)

### C++ is case-sensitive

`int Main()` will compile but it will not *link*

**Definition:** A link-time error occurs 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 function with *exactly* the name “main” somewhere.)

# Make your programs readable (by humans)

---

The program

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

*will* compile and run.

But it will make everyone hate you. Seriously. Just don't.

A good program is **readable by humans** too:

- Code spread across multiple lines, with one statement per line
- Follows indentation conventions (we'll see more later)
- Includes helpful **comments** (lines that begin with `//` or are surrounded by `/* [stuff] */` )

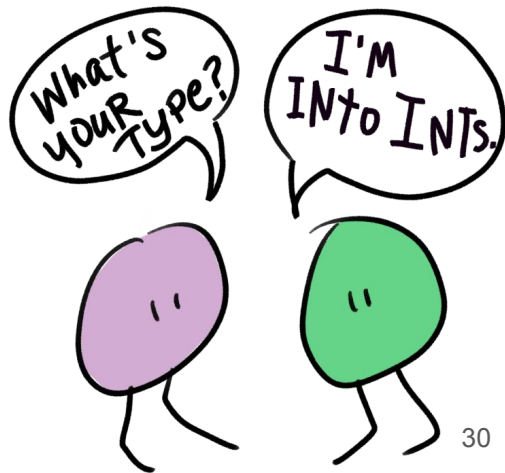


# Chapter 2: Fundamental Data Types

---

## Chapter goals:

- To understand the properties and limitations of integer and floating-point numbers
- To write arithmetic expressions and assignment statements in C++
- To appreciate the importance of comments and good code layout
- To create programs that read and process input, and display the results
- To process strings, using the standard C++ string type

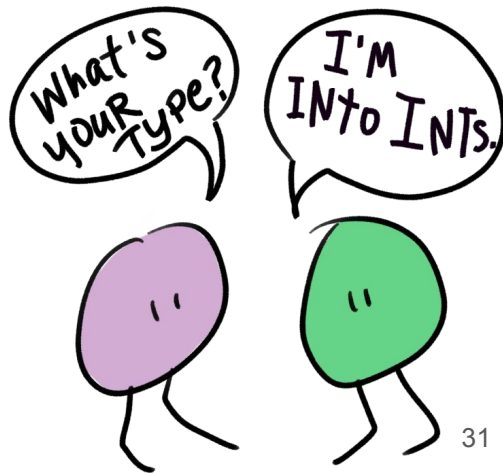


# Chapter 2: Fundamental Data Types

---

## Chapter topics:

- Variables
- Arithmetic
- Input and output
- Problem solving: first do it by hand
- Strings



# Variables

---

- Used to **store information**
  - Contents contain one piece of information at a time
- Has a **name** (its identifier)
  - A good name describes the contents of the variable or what it will be used for





# Variables

---

- Used to **store information**
  - Contents contain one piece of information at a time
- Has a **name** (its identifier)
  - A good name describes the contents of the variable or what it will be used for
- Is like a **parking garage**
  - Each parking space is identified (*like a variable's name*)
  - Each parking space contains a car (*like a variable's contents*)
  - Each parking space can contain only **one** car, and not trucks/buses, just cars



## Variable definitions

---

- When creating a variable, you must **declare** the type of information the variable will hold
  - (more on this later)
- We often give variables an **initial value**
  - This **initialization** puts a value into a variable when the variable is created
  - Initialization is not required.

**Example:**            `int cans_per_pack = 6;`

## Variable definitions

---

- When creating a variable, you must **declare** the type of information the variable will hold
  - (more on this later)
- We often give variables an **initial value**
  - This **initialization** puts a value into a variable when the variable is created
  - Initialization is not required.

**Example:**

```
int cans_per_pack = 6;
```

As always, end line  
with a semicolon ;

**int** = integer numbers  
**double** = decimal numbers  
(or “floating point” values)  
**string** = characters  
(see Table 2 and Sec 2.5)

Use a **descriptive** variable  
name. “cpp” would be a  
crappy name. (see Table 3  
for variable naming rules)

Supplying an initial value is  
option, but often a good idea.  
(see Common Error 2.2)

## Variable definitions: more examples

Declaration	Comment
<code>int cans = 6;</code>	
<code>int total = cans + bottles;</code>	
<code>int bottles = "10";</code>	
<code>int bottles;</code>	
<code>int cans, bottles;</code>	
<code>bottles = 1;</code>	

## Variable definitions: more examples

Declaration	Comment
<code>int cans = 6;</code>	Defines integer <code>cans</code> and initializes it with value of 6
<code>int total = cans + bottles;</code>	Defines integer <code>total</code> and initializes with whatever <code>cans+bottles</code> equals. <code>cans</code> and <code>bottles</code> must have been previously defined and set
<code>int bottles = "10";</code>	<b>Error!</b> You cannot initialize the <code>int bottles</code> with the string "10". You could change this to <code>int bottles = 10;</code>
<code>int bottles;</code>	Defines an integer <code>bottles</code> without initializing it. This can lead to errors
<code>int cans, bottles;</code>	Defines two integer variables in a single statement. Neat!
<code>bottles = 1;</code>	<b>Caution!</b> The type is missing. This is <i>not</i> a definition, but an assignment of a new value to an existing variable.

## Number types

**Definition:** A number written by a programmer is called a number literal.

There are some rules for writing literal values...

## Number literals (Table 2)

Number	Type	Comment
6	int	An integer has no fractional part
-6	int	Integers can be negative
0	int	Zero is totes an integer
0.5	double	A number with a fractional part has type double
1.0	double	Defines two integer variables in a single statement. Neat!
1E6	double	A number in <b>exponential notation</b> ( $1 \times 10^6$ , or 1000000) Numbers in exp. notation always have type double
2.96E-2	double	Negative exponent: $2.96 \times 10^{-2}$ , or 0.0296. Fractional #, so double
100,000		<b>Error!</b> You cannot use a comma as a decimal separator
3 1/2		<b>Error!</b> You cannot use fraction/mixed numbers. Use decimal: 3.5

## Number literals (Table 2)

Number	Type	Comment
6	int	An integer has no fractional part
-6	int	Integers can be negative
0	int	Zero is totes an integer
0.5	double	A number with a fractional part has type double
1.0	double	Defines two integer variables in a single statement. Neat!
1E6	double	A number in <b>exponential notation</b> ( $1 \times 10^6$ , or 1000000) Numbers in exp. notation always have type double
2.96E-2	double	Negative exponent: $2.96 \times 10^{-2}$ , or 0.0296. Fractional #, so double
100,000		<b>Error!</b> You cannot use a comma as a decimal separator
3 1/2		<b>Error!</b> You cannot use fraction/mixed numbers. Use decimal: 3.5



## Number literals (Table 2)

Number	Type	Comment
6	int	An integer has no fractional part
-6	int	Integers can be negative
0	int	Zero is totes an integer
0.5	double	A number with a fractional part has type double
1.0	double	Defines two integer variables in a single statement. Neat!
1E6	double	A number in <b>exponential notation</b> ( $1 \times 10^6$ , or 1000000) Numbers in exp. notation always have type double
2.96E-2	double	Negative exponent: $2.96 \times 10^{-2}$ , or 0.0296. Fractional #, so double
100,000		<b>Error!</b> You cannot use a comma as a decimal separator
3 1/2		<b>Error!</b> You cannot use fraction/mixed numbers. Use decimal: 3.5

## Variable names

**Life Lesson:** Pick a name that explains the variable's purpose

**Example:** `int cans_per_pack = 6;`

“cans\_per\_pack” tells people **exactly** what that variable does. No BS.

You *could* have been stingy with the characters and called it “cpp” instead. But then it is unclear what the variable does, and people reading your code will eat their laptops.



## Variable names -- some rules

---

- 1) Must start with a letter or the underscore character `_` and remaining characters must be letters, numbers, or underscores.
- 2) **NO** symbols like \$ or % or spaces.  
But you can use an underscore (like `cans_per_pack`)
- 3) Variable names are **case-sensitive**  
(so `cansperpack` and `cansPerPack` are different names)
- 4) You cannot use **reserved words** such as “double” or “return” as variable names.

*(They are reserved exclusively for their special C++ meanings; see Appendix B for more reserved names)*

## Variable names -- some examples

Variable name	Comment
can_volume1	Variable names consist of all letters, numbers and underscores
x	In math, you use short variable names like x or y. This is legal in C++ but is not good practice because it makes programs harder to understand
Can_volume	<b>Caution!</b> Variable names are case sensitive. This variable name might get confused with can_volume
6pack	<b>Error!</b> Variable names cannot start with a number
can volume	<b>Error!</b> Variable names cannot contain spaces
double	<b>Error!</b> Contains a reserved word
ltr/fl.oz	<b>Error!</b> Cannot use symbols like . or /

## Variable names -- some examples

Variable name	Comment
can_volume1	Variable names consist of all letters, numbers and underscores
x	In math, you use short variable names like x or y. This is legal in C++ but is not good practice because it makes programs harder to understand
Can_volume	<b>Caution!</b> Variable names are case sensitive. This variable name might get confused with can_volume
6pack	<b>Error!</b> Variable names cannot start with a number
can volume	<b>Error!</b> Variable names cannot contain spaces
double	<b>Error!</b> Contains a reserved word
ltr/fl.oz	<b>Error!</b> Cannot use symbols like . or /

## Variable names -- some examples

Variable name	Comment
can_volume1	Variable names consist of all letters, numbers and underscores
x	In math, you use short variable names like x or y. This is legal in C++ but is not good practice because it makes programs harder to understand
Can_volume	<b>Caution!</b> Variable names are case sensitive. This variable name might get confused with can_volume
6pack	<b>Error!</b> Variable names cannot start with a number
can volume	<b>Error!</b> Variable names cannot contain spaces
double	<b>Error!</b> Contains a reserved word
ltr/fl.oz	<b>Error!</b> Cannot use symbols like . or /

# The assignment statement

---

- Contents in variables can **vary** over time
- Variables can be changed by
  - Assigning to them: `cans_per_pack = 6;`
  - Using the increment or decrement operators: `++` and `--` respectively
  - Inputting into them: input statements (later!)



## Example:

```
int cans_per_pack = -1;  
  
cans_per_pack = 6;  
  
cans_per_pack = 12;  
  
cout << cans_per_pack << endl;
```

# The assignment statement

---

- Contents in variables can **vary** over time
- Variables can be changed by
  - Assigning to them: `cans_per_pack = 6;`
  - Using the increment or decrement operators: `++` and `--` respectively
  - Inputting into them: input statements (later!)



## Example:

```
int cans_per_pack = -1; // initializing cans_per_pack with value -1  
  
cans_per_pack = 6;      // at this point in the code, cans_per_pack has the value 6  
  
cans_per_pack = 12;     // this replaces cans_per_pack's value with 12  
  
cout << cans_per_pack << endl; // this will print a 12 to the screen
```



# The assignment statement

---

Important distinction: **defining** vs **assigning**

- **Definition statement:**      `int cans_per_pack = -1;`
  - Creates it, and possibly initializes it
- **Assignment statement:**      `cans_per_pack = 6;`
  - Modifies an ***already existing*** variable
  - Contents are replaced with the new stuff (was -1, is now 6)



# The *meaning* of the assignment statement

---

- We write:      `cans_per_pack = 6;`
  - But the = does **not** mean the two sides are equal like it does in math
  - The = is an **instruction** -- it tells the computer to do something:
    - take the value of the expression on the right...
    - ... and copy it into the variable on the left.

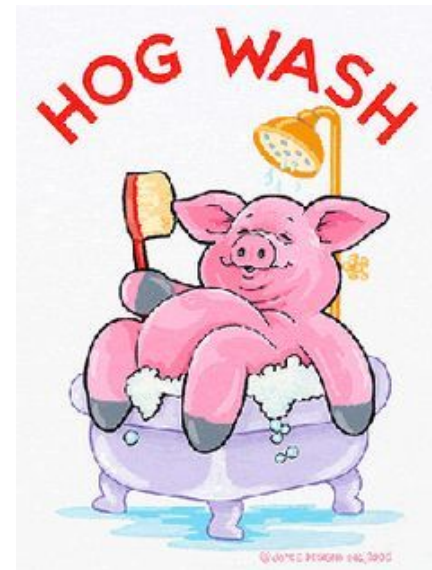


# The *meaning* of the assignment statement

- We write: `cans_per_pack = 6;`
  - But the `=` does **not** mean the two sides are equal like it does in math
  - The `=` is an **instruction** -- it tells the computer to do something:
    - take the value of the expression on the right...
    - ... and copy it into the variable on the left.
  - But what would it mean in math to state this?

`counter = counter + 2;`

- In math, that asks the question: does  $\text{counter} = \text{counter} + 2$ ?  
→ Hogwash!



## Assignment examples

---

```
counter = 1;           // set counter to 1  
counter = counter + 2; // increment counter by 2
```

- The first statement assigns the value 1 to counter
- The second statement looks up what is currently in counter (1)...
- ... then adds 2 to it and copies the result of this addition into the variable on the left
  - This changes the value of counter to 3



## Assignment examples

---

```
counter = 1;           // set counter to 1  
total = counter + 2;  // total is 2 more than whatever is in counter
```

- What about this one? What are the variables' values?
- And what statements must have appears earlier in the program?



# Constants

---

Sometimes we want to represent values that we know will not change during a run of our program.

- For this, we include the designation “const” in our variable declaration:

**Example:** `const double CAN_VOLUME = 12;`

- A const is a variable whose contents cannot be modified and must be set when created (most folks call these “constants” and not “variables” to avoid confusion)
- Constants are commonly written using capital letters to distinguish them visually from regular variables

# Constants

---

Constants are great when we want to be clear about the meaning of numbers in code!

## Example:

```
double volume = cans * 12;
```

- It might be unclear what the 12 represents
- But if we use a const then there is no question:

```
const CAN_VOLUME = 12;
```

```
double volume = cans * CAN_VOLUME;
```

# Constants

---

Constants are great when we want to be clear about the meaning of numbers in code!

## Another Example:

```
double volume_in_cans = cans * 12;
```

```
double eggs_total = cartons * 12;
```

- Do those 12's mean the same thing? Different things? WTF?
- It's bad programming practice to use these ***magic numbers***. Use constants instead.



# Constants

---

Constants are great when we want to be clear about the meaning of numbers in code!

## Nightmare Situation Example:

```
double volume_in_cans = cans * 12;
```

- Suppose we have thousands of lines of code, and the number 12 is used many times throughout the code.
- It might also be used in a ton of different contexts (cans, bottles, eggs, inches, ... )
- What if we upgraded to 16-oz cans for our nondescript beverages?
  - Need to change only the `CAN_VOLUME` instances to 16
- How would we change only the right instances of 12?



# Constants

---

Constants are great when we want to be clear about the meaning of numbers in code!

## Nightmare Situation Example:

Constants to the rescue!

```
const double CAN_VOLUME = 16;
```

```
const double EGGS_PER_CARTON = 12;
```

...

```
double volume_in_cans = cans * CAN_VOLUME;
```

```
double eggs_total = cartons * EGGS_PER_CARTON;
```



## Comments

---

**Comments** are brief explanations for human readers of your code (e.g., other programmers, your classmates, your instructors, your co-workers)

The compiler completely ignores any comments

A leading double slash `//` tells the compiler that the remainder of that line of code is a comment, so it should be ignored.

### Example:

```
const double CAN_VOLUME = 16;    // volume (fl oz) of a beverage can
```

# Comments

---

**Comments** come in two varieties:

1) Single line, using `//`

```
const double CAN_VOLUME = 16;    // volume (fl oz) of a beverage can
```

Compiler ignores everything from `//` to end of the line

2) Multi-line, using `/* [comment goes here] */`

```
/*
```

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

```
*/
```

Compiler ignores everything between `/*` and `*/`

## Complete program: volume1.cpp

---

```
#include <iostream>
```

```
using namespace std;
```

```
/*
```

This program computes the volume (in liters) of a six-pack of soda cans and the total volume of a six-pack and a two-liter bottle.

```
*/
```

```
int main()
```

```
{
```

```
    int cans_per_pack = 6;
```

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

```
    double total_volume = cans_per_pack * CAN_VOLUME;
```

```
    cout << "A six-pack of 12-ounce cans contains " << total_volume << " liters." << endl;
```

```
    const double BOTTLE_VOLUME = 2;           // Two-liter bottle
```

```
    total_volume = total_volume + BOTTLE_VOLUME;
```

```
    cout << "A six-pack and a two-liter bottle contain " << total_volume << " liters." << endl;
```

```
    return 0;
```

```
}
```

## Common error: using undefined variables

---

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

**Example:** The following sequence 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.

So when compiler reaches the first statement, it will freak out because it does not know what `liter_per_ounce` is.



## Common error: using uninitialized variables

---

Initializing a variable is not required, but is good practice.

But all variables have **some** value, even uninitialized ones.

→ Variables are just some place in the computer's memory.

→ **Something** will be there, the junk left over from some previous calculation or simply the random value there when the transistors in RAM were first turned on.

→ Using an uninitialized variable is like picking a parking spot in a garage without knowing what's there. **Unpredictable.**



### Examples:

```
int bottles;  
int bottle_volume = bottles * 2;  
cout << bottle_volume << endl;
```

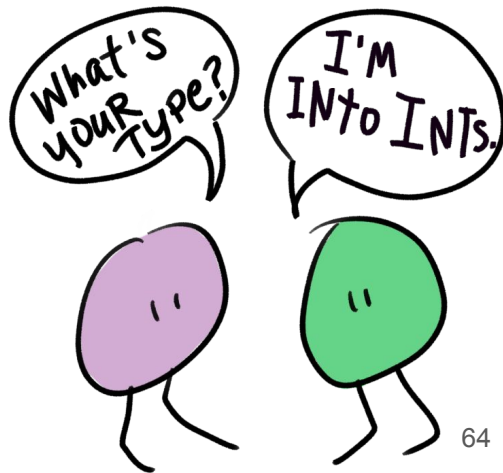
```
// Forgot to initialize  
// What will this be?!?  
// What will print to the screen?!?
```

# Chapter 2: Fundamental Data Types

---

## Chapter topics:

- Variables
- Arithmetic
- **Input and output**
- Problem solving: first do it by hand
- Strings





# Input

---

- Sometimes the programmer does not know what should be stored in a variable, but the user does
- The programmer must get the input value from the user
  - Users need to be prompted - how else would they know they need to type something?
  - Prompts are done in output (`cout`) statements
  - The keyboard needs to be read from (`cin`) statements

## Input -- using `cin >>`

---

- Sometimes the programmer does not know what should be stored in a variable, but the user does
- The programmer must get the input value from the user
  - Users need to be prompted - how else would they know they need to type something?
  - Prompts are done in output (`cout`) statements
  - The keyboard needs to be read from (`cin`) statements
- To read values from the keyboard, you input them from an object called `cin`
- The “double greater than” operator `>>` denotes the “send to” command

`cin >> bottles;` ← takes keyboard input and sends it into the `bottles` variable

## Input -- using cin >>

---

**Example:** Write some code to prompt the user to enter an integer number of bottles, and store this variable.

## Input -- using cin >>

---

**Example:** Write some code to prompt the user to enter an integer number of bottles, and store this variable.

Display a prompt to the user

Don't use endl here so the input is on the same line as the prompt.

**Solution:**

```
cout << "Enter the number of bottles: ";
```

```
int bottles;
```

```
cin >> bottles;
```

Define a variable to hold the input value (could be above the cout statement)

Program waits for user input, then places it into the variable bottles

## Input -- using cin >>

---

You can read more than one value in a single input statement:

```
cout << "Enter the number of bottles and cans: ";  
  
cin >> bottles >> cans;
```

The user can supply both inputs on the same line:

```
Enter the number of bottles and cans:  2  6
```

Or the user can press Enter or Tab after each input, because cin treats all blank spaces the same

## Formatted Output

---

- When you print an amount in dollars and cents, you generally want it to be *rounded* to two decimal places
- We saw earlier how to round off and store a value. But here, we only want to round for display purposes (and keep the more accurate number for calculations)
- A **manipulator** is something that is sent to `cout` to specify how values should be formatted
- To use manipulators, you must include the `iomanip` header and namespace `std`

```
#include <iomanip>
using namespace std;
```

## Formatted Output -- fixed << setprecision()

---

Which do you think the user prefers to see on her gas bill?

- Price per liter: \$1.22
- Price per liter: \$1.21997

**Example:** Including `fixed << setprecision(2)` (for example) will round to 2 decimal places:

```
price_per_liter = 1.21997
```

```
cout << fixed << setprecision(2) << "Price per liter: $" << price_per_liter << endl;
```

## Formatted Output -- setw()

---

Use the setw manipulator to set the **width** of the next output field

The width is the total number of characters, including digits, the decimal point and spaces

If you want aligned columns of certain widths, this is your jam

**Example:** To print numbers, right justified, in columns that are 8 characters wide, you use:

```
cout << setw(8) << ...
```

before **every** column's data

Let's look at an **example!** Suppose we wanted to show a **data table** of some kind.



## Formatted Output -- Examples

---

Given    `int quantity = 10;`  
         `double price = 19.95;`

What do the following statements print? (*represent leading spaces as underscores \_* )

```
cout << "Quantity:" << setw(4) << quantity;
```

```
cout << "Price:" << fixed << setw(8) << setprecision(2) << price;
```

```
cout << "Price:" << fixed << setprecision(2) << price;
```

```
cout << fixed << setprecision(3) << price;
```

```
cout << fixed << setprecision(1) << price;
```

## Persistence -- `setw()` vs `setprecision()`

---

There is a notable difference between `setprecision` and `setw` manipulators:

- Once you `setprecision`, that precision will be used for ***all*** floating-point numbers until the next time you set the precision
- `setw` only affects the ***next*** value

## Formatted Output -- Examples (Table 7)

Output statement	Output	Comment
<code>cout &lt;&lt; 12.345678;</code>	12.3457	
<code>cout &lt;&lt; fixed &lt;&lt; setprecision(2) &lt;&lt; 12.3;</code>	12.30	
<code>cout &lt;&lt; ":" &lt;&lt; setw(6) &lt;&lt; 12;</code>	: 12	
<code>cout &lt;&lt; ":" &lt;&lt; setw(2) &lt;&lt; 123;</code>	:123	
<code>cout &lt;&lt; setw(6) &lt;&lt; ":" &lt;&lt; 12;</code>	:12	

## Formatted Output -- Examples (Table 7)

Output statement	Output	Comment
<code>cout &lt;&lt; 12.345678;</code>	12.3457	By default, a number is printed with 6 significant digits (total, left+right of decimal pt)
<code>cout &lt;&lt; fixed &lt;&lt; setprecision(2) &lt;&lt; 12.3;</code>	12.30	The <code>fixed</code> and <code>setprecision</code> manipulators control the # of digits after decimal pt
<code>cout &lt;&lt; ":" &lt;&lt; setw(6) &lt;&lt; 12;</code>	: 12	4 spaces are printed before the number, total of 6 characters
<code>cout &lt;&lt; ":" &lt;&lt; setw(2) &lt;&lt; 123;</code>	:123	Width isn't sufficient, so <code>setw</code> is ignored
<code>cout &lt;&lt; setw(6) &lt;&lt; ":" &lt;&lt; 12;</code>	:12	Width only refers to <b>next item</b> , which is : Here, : is preceded by 5 spaces

## A complete program for volumes!

---

**Example:** Write a complete program to take in the price of a six-pack of cans, and the volume of each can (in fluid ounces), and output to the screen the price per fluid ounce, rounding to two decimal places.

## A complete program for volumes!

---

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // Read price per pack
    cout << "Please enter the price for a six-pack: ";
    double pack_price;
    cin >> pack_price;

    // Read can volume
    cout << "Please enter the volume for each can (in ounces): ";
    double can_volume;
    cin >> can_volume;
```

(continued on next slide)

## A complete program for volumes!

---

(continued from previous slide)

```
// Compute pack volume
const double CANS_PER_PACK = 6;
double pack_volume = can_volume * CANS_PER_PACK;

// Compute and print price per ounce
double price_per_ounce = pack_price / pack_volume;
cout << fixed << setprecision(2);
cout << "Price per ounce: " << price_per_ounce << endl;
return 0;
}
```

# What just happened...?

---

- We saw what a program looks like!
- We saw what are some common errors!
- We saw some debugging strategies!
- We saw some ways to write good, wholesome, human-readable code!
- We learned how to represent numerical variables!
- We learned how to input information into the computer, and get information back out!  
→ `cin >> & cout <<`
- We saw how to manage **input** from keyboard and **output** to screen!  
→ `fixed << setprecision(##) and setw(##)`







# Your first program

---

The classic first program everyone writes: Hello world!

Its job is to write the message “Hello world!” to the screen

Before we dive in, let’s all code it together!

You can code stuff however you want. Working off the CSEL server in JupyterLab is a nice alternative to Cloud9.

- 1) <https://coding.csel.io>
- 2) Sign in with CU Identikey credentials
- 3) “Start my server”
- 4) Open a Terminal and a Text Editor  
(might need to open a New Launcher by going to File → Open New Launcher)

