# Chapter Two: 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 be able to define and initialize variables and constants
- To learn how to read user input and display program output
- To use the standard C++ `string` type to define and manipulate character strings
- To be able to write simple programs that read numbers and text, process the input, and display the results



## Defining Variables (2.1)

- A variable
  - is used to store information (values): the contents of the variable
    - A variable can contain one piece of 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



## Defining Variables

Parking garages store cars.



## Defining Variables

Each parking space is identified
– like a variable's identifier

A each parking space in a garage "contains" a car
– like a variable's current contents.



## Defining Variables

and
each space can contain only *one* car

and
*only* cars, not buses or trucks

## Defining Variables – Type and Initialization

- When creating variables, the programmer specifies the type of information to be stored.
  - (more on types later) e.g. integer or real or string
- 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.



## Defining Variables

The following statement defines a variable.

`cans_per_pack` is the variable's name. (identifier)

EX: `int cans_per_pack = 6;`

`int` (type) indicates that the variable `cans_per_pack` will be used to hold integers.

`= 6` (initialization) indicates that the variable `cans_per_pack` will initially contain the value 6.



## Defining Variables

**Syntax 2.1 Variable Definition**

```
int cans_per_pack = 6;
```

Types introduced in this chapter are the number types `int` and `double` and the `string` type

Must obey the rules for valid names

A variable definition ends with a semicolon.

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

Use a descriptive variable name.



## Defining Variables

**Table 1 Variable Definitions in C++**

| Variable Name | Comment |
|---|---|
| `int cans = 6;` | Defines an integer variable and initializes it with 6. |
| `int total = cans + bottles;` | The initial value need not be a constant. (Of course, `cans` and `bottles` must have been previously defined.) |
| ⊘ `bottles = 1;` | **Error:** The type is missing. This statement is not a definition but an assignment of a new value to an existing variable – see Section 2.2. |
| ⊘ `int bottles = "10";` | **Error:** You cannot initialize a number with a string. |
| `int bottles;` | Defines an integer variable without initializing it. This can be a cause for errors – see Common Error 2.2 on page 40. |
| `int cans, bottles;` | Defines two integer variables in a single statement. In this book, we will define each variable in a separate statement. |



## Choosing Variable Names

- When you define a variable, you should pick a name that explains its purpose.
- For example, it is better to use a descriptive name, such as `can_volume`, than a terse name, such as `cv`.



## Choosing Variable Names

In C++, there are a few simple rules for creating variable names:

## Rules for Variable Names

1. Variable names must start with a letter or the underscore (_) character, and the remaining characters must be letters numbers, or underscores. cannot start w/a #
2. You cannot use other symbols such as $ or %. Spaces are not permitted inside names; you can use an underscore instead of a space, as in `can_volume`.
3. Variable names are *case-sensitive*, that is, `CanVolume` and `canvolume` are different names.
   For that reason, it is a good idea to use only lowercase letters in variable names.
4. You cannot use *reserved words* such as `double` or `return` as names; these words are reserved exclusively for their special C++ meanings.



## Rules

| 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. |
| ⚠ CanVolume | **Caution:** Variable names are case-sensitive. This variable name is different from canvolume. |
| ⊘ 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 . |



## Number Literals

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

There are rules for writing literal values:

1. integers do not have decimal points
2. real #s (double) have decimal points OR are written in exponential notation
3. no commas in your #s
4. no fractions w/integers



## Number Literals

| Number | 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 $1 \times 10^6$ | 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 |



## Number Ranges – Not Standardized

The C++ Standard does not completely specify the number of bytes or ranges for numeric types.



## Number Ranges – Not Standardized

bit = 0 or 1 **Table 4 Number Types** 8 bits = 1 byte

| Type | Typical Range | Typical Size |
|---|---|---|
| ★ int | −2,147,483,648 . . . 2,147,483,647 (about 2 billion) | 4 bytes |
| unsigned | 0 . . . 4,294,967,295 | 4 bytes |
| short | −32,768 . . . 32,767 | 2 bytes |
| unsigned short | 0 . . . 65,535 | 2 bytes |
| ★ double | The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits | 8 bytes |
| float | The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits | 4 bytes |

## long long – Not Standard C++

Some compiler manufacturers have added other types like:

```
long long
```

integer type

| | | |
|---|---|---|
| long long | -9,223,372,036,854,775,808 . . . 9,223,372,036,854,775,807 | 8 bytes |

This type is not in the C++ standard as of this writing.



## Comments

- *Comments* are explanations for human readers of your code (other programmers).
- The compiler ignores comments completely.

Ex:

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

Comment



## Comments

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:

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



## Notice All the Issues Covered So Far in this Program

Ex:

ch02/volume1.cpp

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

/*
   This program computes the volume (in liters) of a
   six-pack of soda cans.
*/
int main()
{
   int cans_per_pack = 6;
   double can_volume = 0.355; // Liters in a 12-ounce can

   cout << "A sixpack of 12-ounce cans contains "
      << cans_per_pack * can_volume << " liters." << endl;

   return 0;
}
```

always indicate purpose as a comment

int * double → double



## Common Errors – 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 Errors – 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, the flotsam 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;// Result is unpredictable
```

What value would be output from the following statement?

```
cout << bottle_volume << endl; // Unpredictable
```

## Modifying Variables (2.2)

- 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



## Modifying Variables – The Assignment Statement

- An *assignment statement*

  stores a new value in a variable, replacing the previously stored value.

```
identifier = expression;
```



## Modifying Variables – The Assignment Statement

EX:

```
cans_per_pack = 8;
```

This assignment statement changes the value stored in `cans_per_pack` to be 8.

The previous value is replaced.



## Modifying Variables – The Assignment Statement

**Syntax 2.2 Assignment**

```
double total = 0;
.
.
total = bottles * BOTTLE_VOLUME;
.
.
.
total = total + cans * CAN_VOLUME;
```

This is an initialization of a new variable, NOT an assignment.

This is an assignment.

The name of a previously defined variable

The expression that replaces the previous value

The same name can occur on both sides.



## Modifying Variables – The Assignment Statement

- 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.



## Modifying Variables – The Assignment Statement

- 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 + 1;
```

counter EQUALS counter + 1 ?

assignment operator

$x = x + 1$
$-x \quad -x$
$0 = 1$ ∴ no solution

## Modifying Variables – The Assignment Statement

```
counter = 11; // set counter to 11
counter = counter + 1; // increment
```

by 1



## Modifying Variables – The Assignment Statement

```
counter = 11; // set counter to 11
counter = counter + 1; // increment
```

1. Look up what is currently in counter (11)



## Modifying Variables – The Assignment Statement

```
counter = 11; // set counter to 11
counter = counter + 1; // increment
```

1. Look up what is currently in counter (11)
2. Add 1 to that value (12)



## Modifying Variables – The Assignment Statement

```
counter = 11; // set counter to 11
counter = counter + 1; // increment
```

1. Look up what is currently in counter (11)
2. Add 1 to that value (12)
3. *copy* the result of the addition expression *into* the variable on the left, changing counter

```
cout << counter << endl;
```

12 is shown



## Modifying Variables – Increment and Decrement

- Changing a variable by adding or subtracting 1 is so common that there is a special shorthand for these:

add 1 subtract 1

The increment and decrement operators.

```
counter++; // add 1 to counter
counter--; // subtract 1 from counter
```

++ increment operator

-- decrement operator



## Modifying Variables – Increment and Decrement

C++ was based on C and so it's one better than C, right?

Guess how C++ got its name!

## Modifying Variables – Input Statements

- 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?
  - ① – Prompts are an output statements
- The keyboard needs to be read from
  - ② – This is done with an input statement

cin



## Modifying Variables – Input Statements

The input statement

- To read values from the keyboard, you input them from an object called `cin`.
- The << operator denotes the "send to" command.
- `cin >> bottles;`
  is an *input statement*.

Of course, `bottles` must be defined earlier.

we use cout << ...

but cin >> ....



## Modifying Variables – Input Statements

**Syntax 2.3 Input Statement**

Example:

Don't use endl here.

① Display a prompt in the console window.

② Define a variable to hold the input value.

③ The program waits for user input, then places the input into the variable.

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



Good programming practice is to

define variables when you need them

(not at the top of the program necessarily)

## Known Values – Constants for Known, Constant Values

- Sometimes the programmer knows certain values just from analyzing the problem, for this kind of information, programmers use the reserved word `const`.
- The reserved word `const` is used to define a constant.
- A `const` is a variable whose contents cannot be changed and must be set when created.
  (Most programmers just call them constants, not variables.)
- Constants are commonly written using capital letters to distinguish them visually from regular variables:

EX: `const double BOTTLE_VOLUME = 2;`

# liters in a 2 liter bottle



## Known Values – Constants for Known, Constant Values

Another good reason for using constants:

```
double volume = bottles * 2;
```

What does that 2 mean?

## Known Values – Constants for Known, Constant Values

If we use a constant there is no question:

```
double volume = bottles * BOTTLE_VOLUME;
```

Any questions?



## Known Values – Constants – No Magic Numbers!

And still another good reason for using constants:

```
double bottleVolume = bottles * 2;
double canVolume = cans * 2;
```

What does *that* 2 mean?

— *WHICH 2?*

That 2

is called a "***magic number***"

(so is that one)

because it would require magic to know what 2 means.

It is not good programming practice to use magic numbers.
Use constants.



## Known Values – Constants for Known, Constant Values

And it can get even worse ...

Suppose that the number 2 appears hundreds of times throughout a five-hundred-line program?

Now we need to change the BOTTLE_VOLUME to 2.23 (because we are now using a bottle with a different shape)

How to change ***only*** some of those magic numbers 2's?



## Known Values – Constants for Known, Constant Values

Constants to the rescue!

```
const double BOTTLE_VOLUME = 2.23;
const double CAN_VOLUME = 2;

...

double bottleVolume = bottles * BOTTLE_VOLUME;
double canVolume = cans * CAN_VOLUME;
```

**(Look no magic numbers!)**



## Ex: The Complete Program for Volumes

ch02/volume2.cpp

```
#include <iostream>

using namespace std;

int main()
{
   const double BOTTLE_VOLUME = 2;
   const double LITER_PER_OUNCE = 0.0296;
   const double CAN_VOLUME = 12 * LITER_PER_OUNCE;

   double total_volume = 0;

   // Read number of bottles ← comment

   // Display prompt and get user response ← comment
   cout << "Please enter the number of bottles: ";
   int bottles;
   cin >> bottles;
```



## The Complete Program for Volumes

```
   // Start the computation of the total volume ← comment

   total_volume = bottles * BOTTLE_VOLUME;

   // Read number of cans ← comment

   cout << "Please enter the number of cans: ";
   int cans;
   cin >> cans;

   // Update the total volume ← comment

   total_volume = total_volume + cans * CAN_VOLUME;

   cout << "Total volume: " << total_volume << endl;

   return 0;
}
```

## Compound Assignment Operators (Variable on left MUST occur on right in order to use!)

In C++, you can combine arithmetic and assignments.
For example, the statement

```
total += cans * CAN_VOLUME;
```

is a shortcut for

```
total = total + cans * CAN_VOLUME;
```

Similarly,

```
total *= 2;
```

is another way of writing

```
total = total * 2;
```

also available:
/=
-=

Many programmers *prefer* using this form of coding.



## Arithmetic Operators (2.3)

C++ has the same arithmetic operators as a calculator:

- * for multiplication: a * b
  (not a · b or ab as in math)
- / for division: a / b
  (not ÷ or a fraction bar as in math)
- + for addition: a + b
- - for subtraction: a - b

put a space before and a space after each operator



## Arithmetic Operators – Precedence

PEMDAS

Just as in regular algebraic notation,
* and / have higher precedence
than + and –.

In a + b / 2,
the b / 2 happens first.

parentheses are used to override precedence rules

(a+b) / 2 ← add parentheses if want addition done first



## Arithmetic Operators – Two Kinds of Division

- If both arguments of / are integers, the remainder is discarded:
  7 / 3 is 2, not 2.5 — you get an integer answer!
- but
  7.0 / 4.0
  7 / 4.0
  7.0 / 4
- all yield 1.75.

at least one argument must be real (decimal pt) to get a real answer

not considered efficient b/c the computer must coerce the integer argument into a real argument



## Arithmetic Operators – Getting the Remainder

- The % operator computes the remainder of an integer division.
- It is called the **modulus operator** (also modulo and mod)

EX: 5 % 3 would be 2 ← the remainder

- It has nothing to do with the % key on a calculator



## Arithmetic Operators – Getting the Remainder

EX: Time to break open the piggy bank. ← contains only pennies

You want to determine the value in dollars and cents stored in the piggy bank.
You obtain the dollars through an integer division by 100.
The integer division discards the remainder.
To obtain the remainder (cents), use the % operator:

```
int pennies = 1729;
int dollars = pennies / 100; // Sets dollars to 17
int cents = pennies % 100; // Sets cents to 29
```

(yes, 100 is a magic number)

## Arithmetic Operators – Getting the Remainder

```
dollars =    /  100;

cents =    % 100;
```

Don't worry, Penny wasn't broken or harmed in any way because she's on the right hand side of the = operator.



## Common Error – Mismatched Parentheses

Consider the expression

```
(-(b * b - 4 * a * c) / (2 * a)
```

?

What is wrong with it?

The parentheses are *unbalanced*.
This is very common with complicated expressions.



## Common Error – Mismatched Parentheses

Now consider this expression

```
-(b * b - (4 * a * c))) / 2 * a)
```

It is still is not correct.

There are too many closing parentheses.



## Common Error – Mismatched Parentheses – A Solution

The Muttering Method

Count (to yourself):
starting with 1 at the 1st parenthesis
add one for each (
subtract one for each )

```
- ( b * b - ( 4 * a * c ) )  ) / 2 * a )
  1         2             1 0  -1  OH NO!
```

If your count is not 0 when you finish, or if you ever drop to -1, STOP, something is wrong.



## Roundoff Errors

This program produces the wrong output:

```
#include <iostream>
using namespace std;
int main()
{
   double price = 4.35;
   int cents = 100 * price;
         // Should be 100 * 4.35 = 435
   cout << cents << endl;
         // Prints 434!
   return 0;
}
```

Why?



## Roundoff Errors

- In the processor hardware, numbers are represented in the binary number system, not in decimal.
- In the binary system, there is no exact representation for 4.35, just as there is no exact representation for ⅓ in the decimal system.
  The representation used by the computer is just a little less than 4.35, so 100 times that value is just a little less than 435.
- The remedy is to add 0.5 in order to round to the nearest integer:

```
int cents = 100 * price + 0.5;
```

usually only happens when dealing with integers

How to write arithmetic expressions in C++?

EX: Write $\frac{x+y}{2}$ in C++ syntax.

(x + y)/ 2.

## Math Functions

EX: What about this?

$$b+\left(1+\frac{r}{100}\right)^n$$

Inside the parentheses is easy:

`1. + (r / 100)` or `1. + r/100.`

But that raised to the *n*?



## Math Functions

- In C++, there are no symbols for powers and roots. To compute them, you must call *functions*.
- The C++ library defines many mathematical functions such as `sqrt` (square root) and `pow` (raising to a power).
- To use the functions in this library, called the `cmath` library, you must place the line:

```
#include <cmath>
```

at the top of your program file.

- It is also necessary to include

```
using namespace std;
```

at the top of your program file.



## Math Functions

Using the `pow` function:

```
b + pow(1. + r / 100, n)
```

base  exponent

pow ( base , exponent )



## Some Math Functions in cmath

| | |
|---|---|
| `pow(base, power)` | base raised to power |
| `sqrt(x)` | square root of x |
| `sin(x)` | sine of *x* (*x* in radians) |
| `cos(x)` | cosine of *x* |
| `tan(x)` | tangent of *x* |
| `log10(x)` | (decimal log) $\log_{10}(x)$, $x > 0$ |
| `fabs(x)` | absolute value $\lvert x \rvert$ |

for absolute value of a real #

abs(x) is used when arg. is an integer

in cstdlib



EX: Write $\sqrt{a^2+b^2}$ in C++ syntax.

```
#include <cmath>
using namespace std;
:
y = sqrt(pow(a,2) + pow(b,2))
:
```

a*a  b*b

more efficient when exp ≥ 3

a*a is more efficient than pow(a,2)

## Converting Floating-Point Numbers to Integers

- When a floating-point value is assigned to an integer variable, the fractional part is discarded:

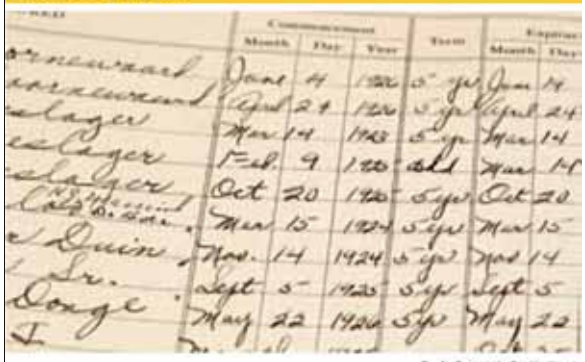
```
double price = 2.55;
int dollars = price;
      // Sets dollars to 2
```

- You probably want to round to the *nearest* integer. To round a positive floating-point value to the nearest integer, add 0.5 and then convert to an integer:

```
int dollars = price + 0.5;
      // Rounds to the nearest integer
```



## Formatting Output





## Formatting Output

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

```
Price per liter: $1.22
```

or

```
Price per liter: $1.21997
```

how do we make sure only 2 decimal places print out? and rounded



## Formatting Output

- When you print an amount in dollars and cents, you usually want it to be *rounded* to two significant digits.
- You learned how to actually round off and store a value but, for output, we want to round off *only* for display.
- 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 in your program:

```
#include <iomanip>
```

and

```
using namespace std;
```

is also needed



## Formatting Output (section 8.3 has more details)

**Table 8 Formatting Output**

| Output Statement | Output | Comment |
|---|---|---|
| `cout << 12.345678;` | 12.3457 | By default, a number is printed with 6 significant digits. |
| `cout << fixed << setprecision(2) << 12.3;` | 12.30 | Use the `fixed` and `setprecision` manipulators to control the number of digits after the decimal point. |
| `cout << ":" << setw(6) << 12;` | :  12 | Four spaces are printed before the number, for a total width of 6 characters. |
| `cout << ":" << setw(2) << 123;` | :123 | If the width not sufficient, it is ignored. |
| `cout << setw(8) << ":" << 12.3;` | ----:12.3 | The width only refers to the next item. Here, the : is preceded by five spaces. |

gets field width for next item only



## Formatting Output

- You can combine manipulators and values to be displayed into a single statement:

EX:

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

## Strings (2.4)

- Strings are sequences of characters:

  "hello world"

- If you include the string header, you can create variables to hold literal strings:

```
#include <string>
using namespace std;
...
string name = "Harry";
        // literal string "Harry" stored
```



## Strings – No Initialization Needed

- String variables are guaranteed to be initialized even if you don't initialize them:

```
string response;
        // literal string "" stored
```

- "" is called the empty or null string.

* String variables are automatically initialized to the empty or null string unless they are given a different value



## Strings – Concatenation

Use the + operator to *concatenate* strings; that is, put them together to yield a longer string.

EX:

```
string fname = "Harry";
string lname = "Morgan";
string name = fname + lname;
cout << name << endl;
name = fname + "_" + lname;
cout << name << endl;
```

The output will be

```
HarryMorgan
Harry Morgan
```



## Strings – Input

You can read a string from the console:

```
cout << "Please enter your name: ";
string name;
cin >> name;
```

When a string is read with the >> operator, only one word is placed into the `string` variable.

For example, suppose the user types

**Harry Morgan**

as the response to the prompt.
This input consists of two words.
Only the string "Harry" is placed into the variable name.



## Strings – Input The User Typed Harry Morgan

① You can use another input to read the second word.

```
cout << "Please enter your name: ";
string fname, lname;
cin >> fname >> lname;
```

gets Harry — gets Morgan

② getline (cin, string_variable);

EX: getline (cin, name);



## Errors with Strings

```
string greeting = "Hello, " + " World!";
                // will not compile
```

Literal strings cannot be concatenated.

* one argument of + must be a variable string in order to concatenate

## Strings Functions – Length

- The `length` *member function* yields the number of characters in a string.
- Unlike the `sqrt` or `pow` function, the `length` function is *invoked* with the *dot notation:*

```
int n = name.length();
```

will be an integer e.g. 12



## Strings Functions – Substring

- Once you have a string, you can extract substrings by using the substr member function.
- `s.substr(start, length)`
  returns a `string` that is made from the characters in the `string s`, starting at character `start`, and containing `length` characters. (`start` and `length` are integer values).

EX:

```
string greeting = "Hello, World!";
string sub = greeting.substr(0, 5);
    // sub contains "Hello"
```

string characters start counting at 0, NOT 1



EX: name = "Harry Morgan"

name.length() would yield 12

EX: name = "Harry Morgan"

name.substr(1,4) would yield "arry"

## Strings Functions – Numbering the Characters

```
string greeting = "Hello, World!";
string w = greeting.substr(7, 5);
    // w contains "World"
```

Why is 7 the position of the "W" in "World"?

- In most computer languages, the starting position 0 means "start at the beginning."
- The first position in a string is labeled 0, the second one 1, and so on. And don't forget to count the space character after the comma—and the quotation marks are stored.



## Strings Functions – Numbering the Characters

| H | e | l | l | o | , | | W | o | r | l | d | ! |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

The position number of the last character
(12 in "Hello, World!")
is always one less than the length of the string
(13 for "Hello, World").



## Strings Functions – Substring

```
string greeting = "Hello, World!";
string w = greeting.substr(7);
   // w contains "World!" - with the !
```

If you omit the length, you get all the characters from the given position to the end of the string.

s.substr(start)

## String Operations

| Statement | Result | Comment |
|---|---|---|
| `string str = "C";`<br>`str = str + "++";` | `str` is set to "C++" | When applied to strings, + denotes concatenation. |
| `string str = "C" + "++";` | Error | Error: You cannot concatenate two string literals. |
| `cout << "Enter name: ";`<br>`cin >> name;`<br>(User input: Harry Morgan) | `name` contains "Harry" | The >> operator places the next word into the `string` variable. |
| `cout << "Enter name: ";`<br>`cin >> name >> last_name;`<br>(User input: Harry Morgan) | `name` contains "Harry", `last_name` contains "Morgan" | Use multiple >> operators to read more than one word. |
| `string greeting = "H & S";`<br>`int n = greeting.length();` | `n` is set to 5 | Each space counts as one character. |



## String Operations

| Statement | Result | Comment |
|---|---|---|
| `string str = "Sally";`<br>`string str2 = str.substr(1, 3);` | `str2` is set to "all" | Extracts the substring of length 3 starting at position 1. (The initial position is 0.) |
| `string str = "Sally";`<br>`string str2 = str.substr(1);` | `str2` is set to "ally" | If you omit the length, all characters from the position until the end are included. |
| `string a = str.substr(0, 1);` | `a` is set to the initial letter in `str` | Extracts the substring of length 1 starting at position 0. |
| `string b = str.substr(`<br>`str.length() - 1);` | `b` is set to the last letter in `str` | The last letter has position `str.length() - 1`. We need not specify the length. |



## Strings

EX:

Write this code



## Strings

ch02/initials.cpp

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

int main()
{
   cout << "Enter your first name: ";
   string first;
   cin >> first;
   cout << "Enter your significant other's first name: ";
   string second;
   cin >> second;
   string initials = first.substr(0, 1)
      + "&" + second.substr(0, 1);
   cout << initials << endl;

   return 0;
}
```

← don't forget!



## Chapter Summary

1. A variable is a storage location with a name.
2. When defining a variable, you usually specify an initial value.
3. When defining a variable, you also specify the type of its values. (int, double, string)
4. Use the int type for numbers that cannot have a fractional part.
5. Use the double type for floating-point numbers.
6. Use comments to add explanations for humans who read your code. The compiler ignores comments.
7. An assignment statement stores a new value in a variable, replacing the previously stored value.
8. The assignment operator = does not denote mathematical equality.
9. The ++ operator adds 1 to a variable; the -- operator subtracts 1.



## Chapter Summary

10. Use the >> operator to read a value and place it in a variable.
11. You cannot change the value of a variable that is defined as const.
12. If both arguments of / are integers, the remainder is discarded. * 5/4 → 1
13. The % operator computes the remainder of an integer division. 5 % 4 → 1

14. The C++ library defines many mathematical functions such as sqrt (square root) and pow (raising to a power).

* include <cmath>

## Chapter Summary

15. You send manipulators to cout to specify how values should be formatted.

    fixed setprecision(n)
    setw(m)

16. Strings are sequences of characters.
17. Use the + operator to *concatenate* strings; that is, put them together to yield a longer string.
18. The length member function yields the number of characters in a string.
19. A member function is invoked using the dot notation.
20. Use the substr member function to extract a substring of a string.



Programming Assignment for Chp 2

p.75 P2.4
P2.9
P2.10

due: TBA

Also online: Chp 2 Quiz due Feb 5th @ 11pm

Reading Assignment due Jan 29th @ 11pm

this Friday!