Chapter Two: Fundamental Data Types II

C++ for Everyone by Cay Horstmann

Slides by Evan Gallagher & Nikolay Kirov

Lecture Goals

- To create programs that read and process input, and display the results
- To process strings, using the standard C++ **`string`** type

Common Error – 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?

Common Error – 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;
```

Spaces in Expressions

It is easier to read

```
x1 = (-b + sqrt(b * b - 4 * a * c)) / (2 * a);
```

than

```
x1=(-b+sqrt(b*b-4*a*c))/(2*a);
```

Itreallyiseasiertoreadwithspaces!

So always use spaces around all operators: `+ - * / % =`

Spaces in Expressions

However, **don’t** put a space after a *unary* minus:
that’s a **–** used to negate a single quantity like this: **`-b`**

That way, it can be easily distinguished from a *binary* minus, as in **`a - b`**

It is customary **not** to put a space after a function name.

Write **`sqrt(x)`**
not **`sqrt (x)`**

Casts

Occasionally, you need to store a value
into a variable of a different type.

Whenever there is the risk of information loss,
the compiler generally issues a warning.

It’s not a compilation error to lose information.
But it may not be what a programmer intended.

Casts

For example, if you store a **`double`** value into an **`int`** variable, information is lost in two ways:

- The fractional part ***will*** be lost.

```
int n = 1.99999; // NO
```

1 is stored (the decimal part is truncated)

- The magnitude may be too large.

```
int n = 1.0E100; // NO
```

is not likely to work, because 10^{100} is larger than the largest representable integer.

Casts

A ***cast*** is a conversion from one type (such as **`double`**) to another type (such as **`int`**).

This is not safe in general, but if you know it to be safe in a particular circumstance, casting is the *only* way to do the conversion.

Casts

Nevertheless, sometimes you do want to convert a floating-point value into an integer value.

If you are prepared to lose the fractional part and you know that this particular floating point number is not larger than the largest possible integer, then you can turn off the warning by using a cast.

.

Casts

It’s not really about turning off warnings…

Sometimes you *need* to cast a value to a different type.

Consider money.
(A good choice of topic)
(and remember Penny?)

Casts

```
double change; // change owed
change = 999.89;
```

To annoy customers who actually want change when they pay with $10000 bills, we say:

“Sorry, we can only give change in pennies.”

(in a pleasantly lilting voice, of course)

How many pennies do we owe them?

We need to cast the change owed into the correct type for pennies.

Casts

A bank would round down to the nearest penny, of course, but we will do the right thing (even to this annoying customer)…

How to "round up" to the next whole penny?

Add 0.5 to the change and then ***cast*** that amount into an **`int`** value, storing the number of pennies into an **`int`** variable.

```
int cents; // pennies owed
```

Casts

You express a cast in C++ using a ***`static_cast`***

```
static_cast<   >(   )
```

Casts

```
change = 999.89; // change owed

int cents = static_cast<   >(                  );
```

Casts

```
change = 999.89; // change owed

int cents = static_cast<   >(                   );
```

You put the type you want to convert to inside the **<   >**

Casts

```
change = 999.89; // change owed

int cents = static_cast<int>(          );
```

You put the $\boxed{\text{type you want to convert to}}$ inside the **<   >**

Casts

```
change = 999.89; // change owed

int cents = static_cast<int>(          );
```

You put the type you want to convert to inside the `<     >`

You put the value you want to convert inside the `(     )`

Casts

```
change = 999.89; // change owed

int cents = static_cast<int>(100 * change + 0.5);
```

You put the type you want to convert to inside the `<    >`

You put the $\boxed{\text{value you want converted}}$ inside the `(    )`

Casts

```
change = 999.89; // change owed
```

```
int cents = static_cast<int>(100 * change + 0.5);
```

You put the type you want to convert to inside the `<      >`

You put the value you want to convert inside the `(      )`

and you get the value converted to the type: 1000 pennies

Casts

```
change = 999.89; // change owed
```

```
int cents = static_cast<int>(100 * change + 0.5);
```

You put the type you side the **<   >**

You put the value yo the **(   )**

and you get the value converted to the type

$\boxed{1000}$ will be stored into **`cents`**.

Combining Assignment and Arithmetic

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;
```

Many programmers *prefer* using this form of coding.

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 statements

- The keyboard needs to be read from
  - This is done with an input statement

Input

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.

Input

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
```

Input

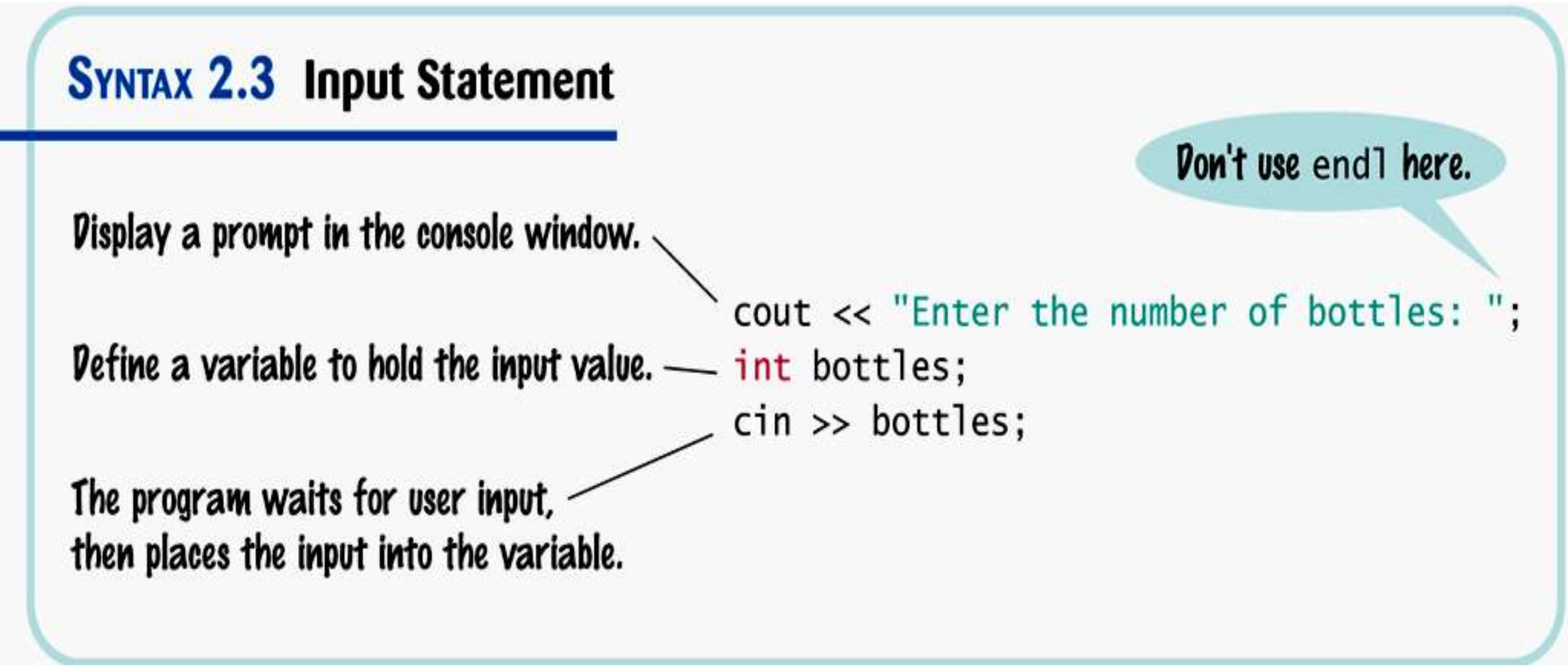
You can read more than one value in a single input statement:

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

Alternatively, the user can press the Enter key after each input:

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

Input Statement



Syntax 2.3 Input Statement

Don't use `endl` here.

Display a prompt in the console window.

Define a variable to hold the input value.

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

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

Formatted Output

Commencement			Term	Expirat	
Month	Day	Year		Month	Day
June	4	1926	5 yrs	June	14
April	24	1924	5 yrs	April	24
Mar	14	1923	5 yrs	Mar	14
Feb.	9	1923	old	Mar	14
Oct	20	1925	5 yrs	Oct	20
Mar	15	1924	5 yrs	Mar	15
Nov.	14	1924	5 yrs	Nov	14
Sept	5	1923	5 yrs	Sept	5
May	22	1926	5 yrs	May	22

Formatted 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

Formatted 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
```

Formatted Output

Table 7 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(6) << ":" << 12;`	`     :12.3`	The width only refers to the next item. Here, the `:` is preceded by five spaces.

Formatted Output

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

```
price_per_liter = 1.21997;
cout << fixed << setprecision(2)
   << "Price per liter: $"
   << price_per_liter << endl;
```

This code produces this output:

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

Formatted Output

You use the **`setw`** manipulator to set the *width* of the next output field.

The width is the total number of characters used for showing the value, including digits, the decimal point, and spaces.

Formatted Output

If you want columns of certain widths, use the **`setw`** manipulator.

For example, if you want a number to be printed, right justified, in a column that is eight characters wide, you use

```
<< setw(8)
```

Formatted Output

This code:

```
price_per_ounce_1 = 10.2372;
price_per_ounce_2 = 117.2;
price_per_ounce_3 = 6.9923435;
cout << setprecision(2);
cout << setw(8) << price_per_ounce_1;
cout << setw(8) << price_per_ounce_2;
cout << setw(8) << price_per_ounce_3;
cout << "--------" << endl;
```

produces this output:

```
   10.24
  117.20
    6.99
--------
```

Formatted Output

There is a notable difference between the **`setprecision`** and **`setw`** manipulators.

Once you set the precision, that width is used for all floating-point numbers until the next time you set the precision.

But **`setw`** affects only the *next* value.

Subsequent values are formatted without added spaces.

A Complete Program for Volumes

ch02/volume2.cpp

```
#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;
```

A Complete Program for Volumes

ch02/volume2.cpp

```
   // 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;
}
```

Strings

- 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

- 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.

Concatenation

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

```
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
```

Common Error – Concatenation of literal strings

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

Literal strings cannot be concatenated.

String 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.

String Input

You can use another input to read the second word.

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

$\underbrace{\texttt{>> fname}}_{\text{gets } \texttt{Harry}}$ $\underbrace{\texttt{>> lname}}_{\text{gets } \texttt{Morgan}}$

String Functions

- 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();
```

String Functions

- 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).

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

String Functions

0 ?

```
string sub = greeting.substr(0, 5);
```

String Functions

```
string greeting = "Hello, World!";
string w = greeting.substr(7, 5);
    // w contains "World" (not the !)
```

`"World"` is 5 characters long but…

why is 7 the position of the “W” in `"World"`?

0 ?

String Functions

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

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—but the quotation marks are ***not*** stored.

String Functions

`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
is always one less than the length of the string

The **`!`** is at position 12 in **`"Hello, World!"`**.
The length of **`"Hello, World!"`** is 13.

(C++ remembers to count the 0 as one of the
positions when counting characters in strings.)

String Functions

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

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

If you do not specify how many characters to take, you get all the rest.

String Functions

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

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

If you omit the starting position and the length, you get all the characters

(not much of *sub*string!)

String Operations

Statement	Result	Comment

`string str = "C";`
`str = str + "++";`
	`str` is set to `"C++"`	When applied to strings, + denotes concatenation.

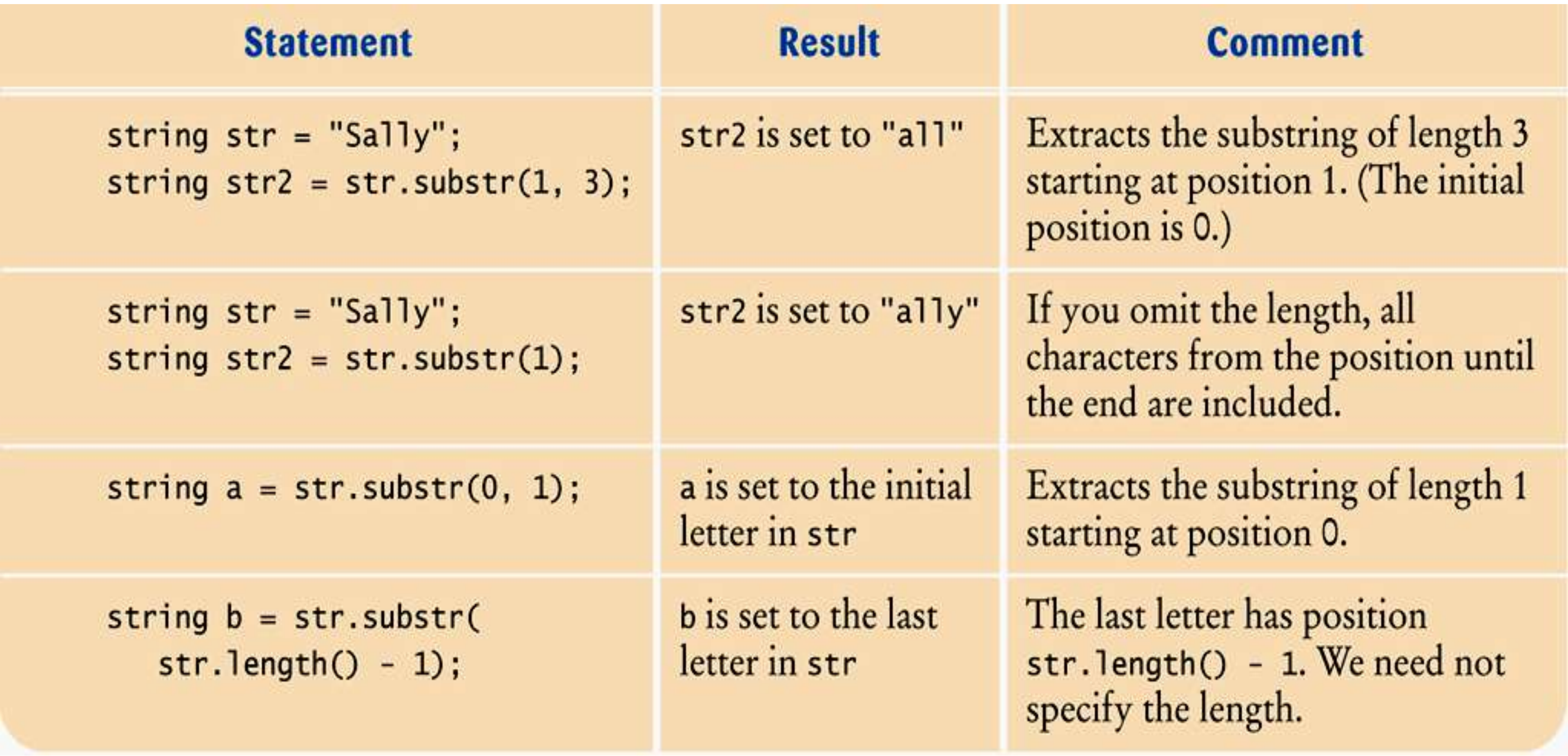
🚫 `string str = "C" + "++";`	**Error**	**Error:** You cannot concatenate two string literals.

`cout << "Enter name: ";`
`cin >> name;`
(User input: Harry Morgan)
	`name` contains `"Harry"`	The >> operator places the next word into the `string` variable.

`cout << "Enter name: ";`
`cin >> name >> last_name;`
(User input: Harry Morgan)
	`name` contains `"Harry"`, `last_name` contains `"Morgan"`	Use multiple >> operators to read more than one word.

`string greeting = "H & S";`
`int n = greeting.length();`
	`n` is set to 5	Each space counts as one character.

String Operations



Statement	Result	Comment

`string str = "Sally";`
`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";`
`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(`
`   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.

String Functions

Write this code

String Functions

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;
}
```

Chapter Summary

Write variable definitions in C++.

- A variable is a storage location with a name.
- When defining a variable, you usually specify an initial value.
- When defining a variable, you also specify the type of its values.
- Use the **`int`** type for numbers that cannot have a fractional part.
- Use the **`double`** type for floating-point numbers.

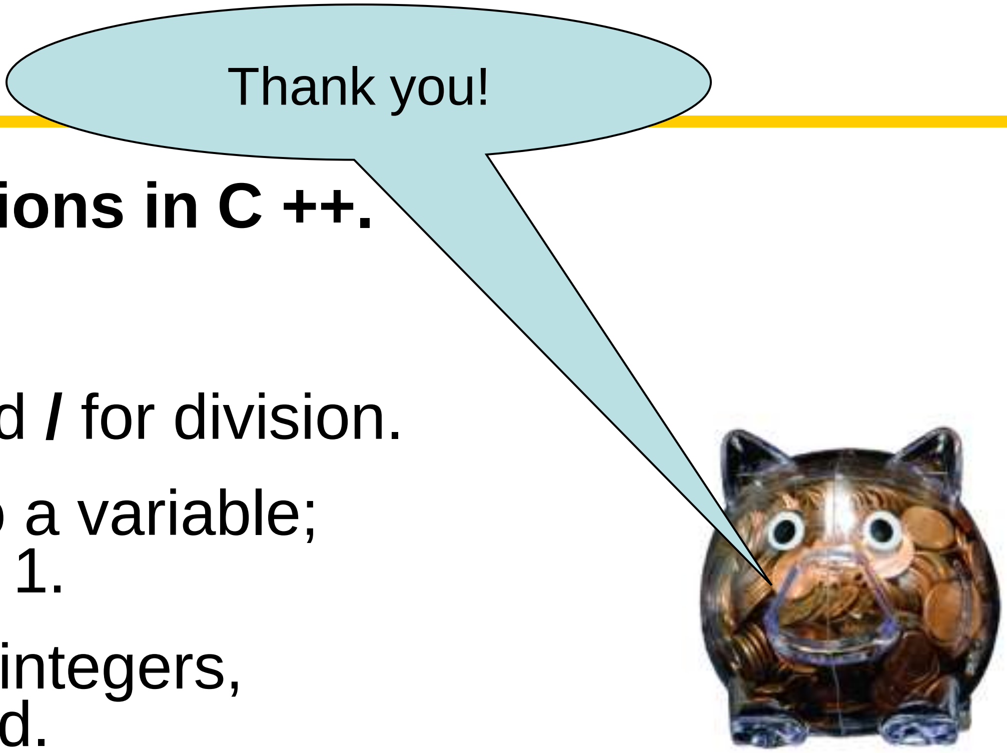
Chapter Summary

- An assignment statement stores a new value in a variable, replacing the previously stored value.
- The assignment operator **=** does *not* denote mathematical equality.
- You cannot change the value of a variable that is defined as **`const`**.
- Use comments to add explanations for humans who read your code.
  The compiler ignores comments.

Chapter Summary

Use the arithmetic operations in C ++.

- Use ***** for multiplication and **/** for division.
- The **++** operator adds 1 to a variable; the **--** operator subtracts 1.
- If both arguments of **/** are integers, the remainder is discarded.
- The **%** operator computes the remainder of an integer division.
- Assigning a floating-point variable to an integer drops the fractional part.
- The C++ library defines many mathematical functions such as **`sqrt`** (square root) and **`pow`** (raising to a power).



Chapter Summary

Write programs that read user input and write formatted output.

- Use the **>>** operator to read a value and place it in a variable.
- You use manipulators to specify how values should be formatted.

Carry out hand calculations when developing an algorithm.

- Pick concrete values for a typical situation to use in a hand calculation.

Chapter Summary

Write programs that process strings.

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

End Fundamental Data Types II