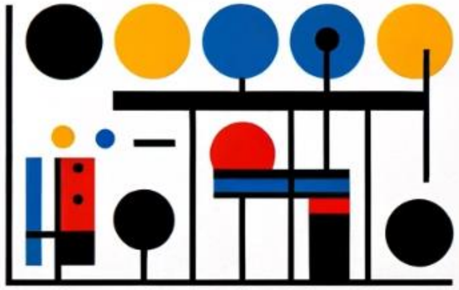
Object-Oriented Programming

Lecture 1: Introduction to C++

Rust Programming



Instructor

Telegram Group

Phairoj Jatanachai

phairoj@jatanachai.space

- Thirty years experience in system integration, software architecture, design and development, mainly in manufacturing
- Bachelor degree of computer engineer, master of business administration
- Certified on Data Science, IoT, Game Development
- Bizinfo Thai Company – Chief Solutions Advisory
- Digital Focus–Digital Technology Consultant
- Superb Consultant– IT Consultant
- Dental Corp Group – Board of Audit Committee

What You’ll Learn

- Object-oriented programming concept
- How to think in term of objects.
- Analyze program specifications and identify appropriate classes and objects.
- Additional programming topics include basic UML modeling such as class diagram and object diagram, principles of object-oriented design, and design patterns.
- Using programing tools, IDE to compile and debug program
- Create C++ application.

Course Learning Outcomes

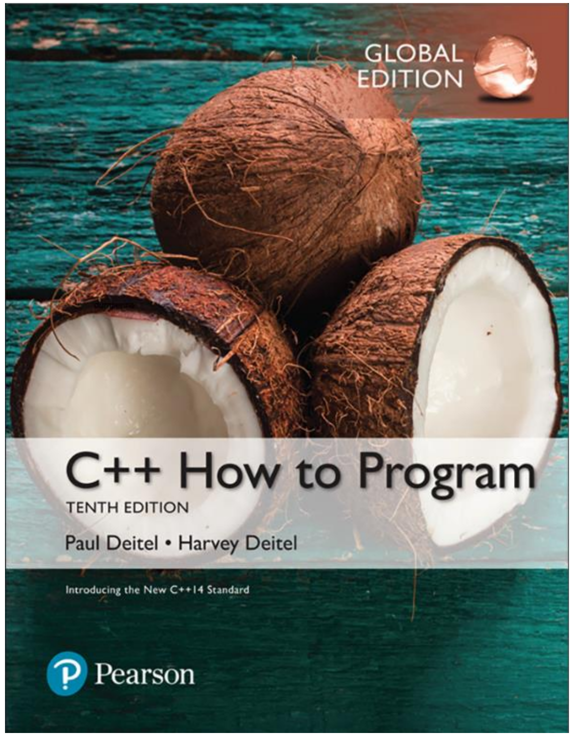
CLO-1	Write simple C++ programs, catch and fix syntax errors, compile the program to an executable and verify that the program run correctly
CLO-2	Understand structured programming concepts and how to split parts of program into logical modules, for example, split common operations into appropriate functions
CLO-3	Understand the rules for defining objects in different scopes in the program, for example, how objects are initialized, assigned, modified, passed into functions and return from functions
CLO-4	Use C++ library to read input, transform data, write output, and manage program data and functions
CLO-5	Test and debug programs with the use of IDE support
CLO-6	Write generic functions using C++ template
CLO-7	Write C++ class, identify class interface, choose appropriate data structure and its operations for class implementation
CLO-8	Understand how to manage low-level data structures for implementation of classes and functions
CLO-9	Understand inheritance and polymorphism in object-oriented programming and their applications
CLO-10	Create a document for a program describing its design and implementation

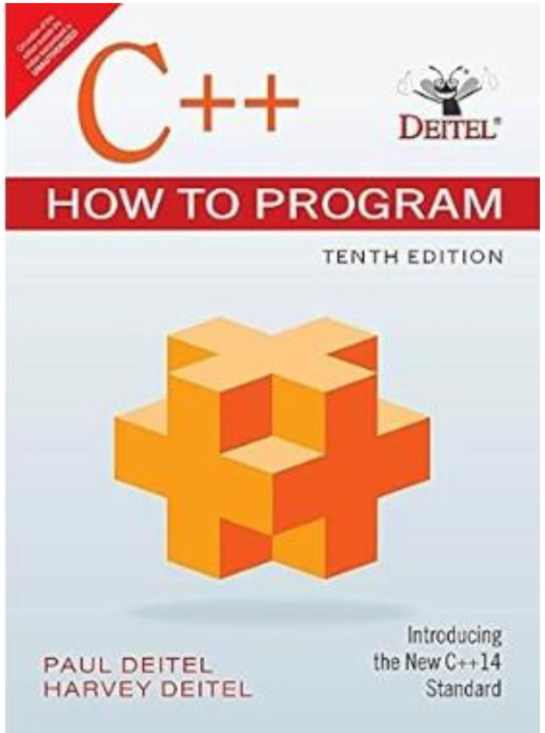
Course Plan

Week	Topic	Activity	CLO
1	Introduction to C++ programming	Lecture / Quiz	1
2	Structured Programming	Lecture	2-3
3	Functions	Lecture / Home work	2-4
4	Functions and Program Structure	Lecture / Quiz	2-4
5	Sequential Containers (Vectors and Strings)	Lecture	2-6
6	User-Defined Types	Lecture / Quiz	2-4, 7
7	Memory Management	Lecture	2-4, 8
8	Inheritance	Lecture	9
9	Memory Management (Part II)	Lecture / Quiz	2-4, 8, 9
10	Algorithm Design	Lecture / Home work	2-4, 6-8
11	Data Abstraction	Lecture	7, 9-10
12	Object-Oriented Modeling	Lecture / Quiz	7, 9-10
13	Advanced C++	Lecture	6-9
14	Selected Problems and Applications	Project Presentation	6-9

Books

- C++ How to Program (10th Edition)
  HARVEY M. DEITEL PAUL DEITEL (Author)


GLOBAL EDITION
C++ How to Program
TENTH EDITION
Paul Deitel • Harvey Deitel
Introducing the New C++14 Standard
Pearson



Assessment

Mid-term	25%
Final Exam	35%
Quiz	5%
Lab	10%
Lab Exam	5%
Homework	5%
Software Project	15%

- Course documents:

https://drive.google.com/drive/folders/18g1c21bNKiLPCf70Djo11U-lYs4VN3p2?usp=sharing

Introduction to C++

Object Oriented Concepts

Object Orientation

- An Object oriented approach views systems and programs as a collection of interacting objects.
- An object is a thing in a computer system that is capable of respond to messages
- The idea of OOP is to try to approach programming in a more natural way by grouping all the code that belongs to a particular object - such as a checking account or a customer - together

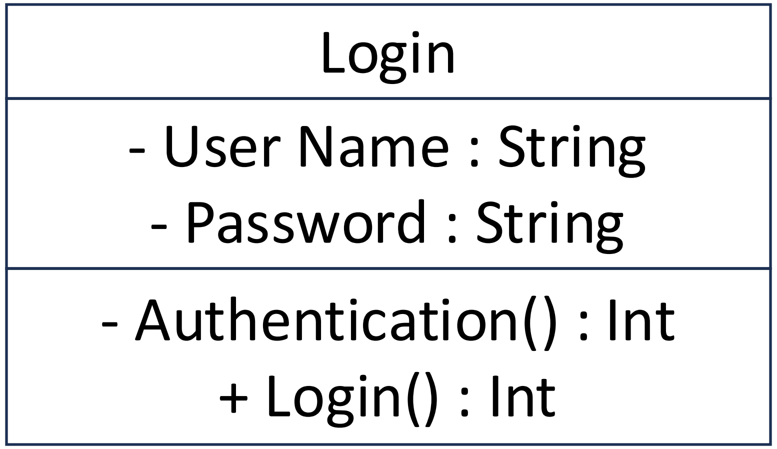
Objects

- Core to the idea of OOPs is the concept of an object.
- An object is anything that is relevant to your program
- A customer, an employee, Inventory, a database, a
- button, a form, a sale are all potential objects
- Benefit of objects
  - More natural way to look at thing
  - Re-usability

Objects/Classes

- A class is a description of an object.
- This description can include:
  - attributes: describe the class
  - methods: describe things the object can do.
- In programming an object is an actual instance of a class

A Class Diagram



Login

- User Name : String
- Password : String

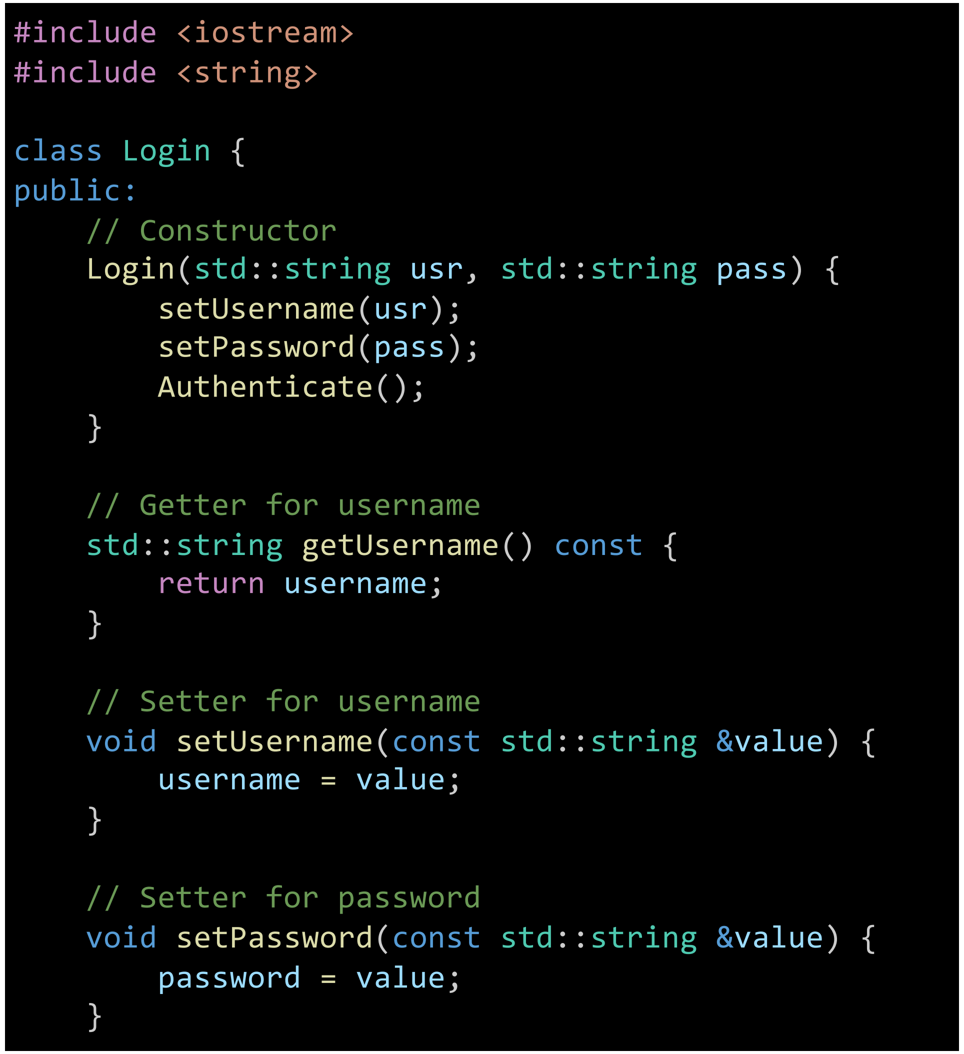
- Authentication() : Int
+ Login() : Int

Class name

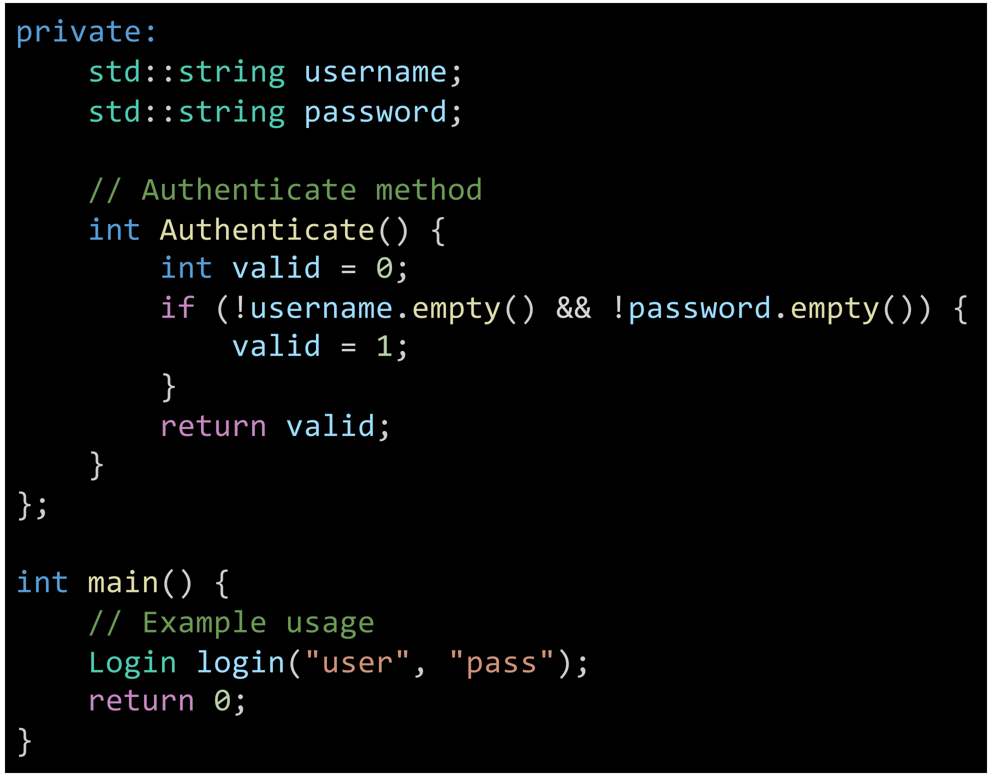
Field name

Method name

- Private
+ Public
# Protect



```cpp
#include <iostream>
#include <string>

class Login {
public:
    // Constructor
    Login(std::string usr, std::string pass) {
        setUsername(usr);
        setPassword(pass);
        Authenticate();
    }

    // Getter for username
    std::string getUsername() const {
        return username;
    }

    // Setter for username
    void setUsername(const std::string &value) {
        username = value;
    }

    // Setter for password
    void setPassword(const std::string &value) {
        password = value;
    }
```



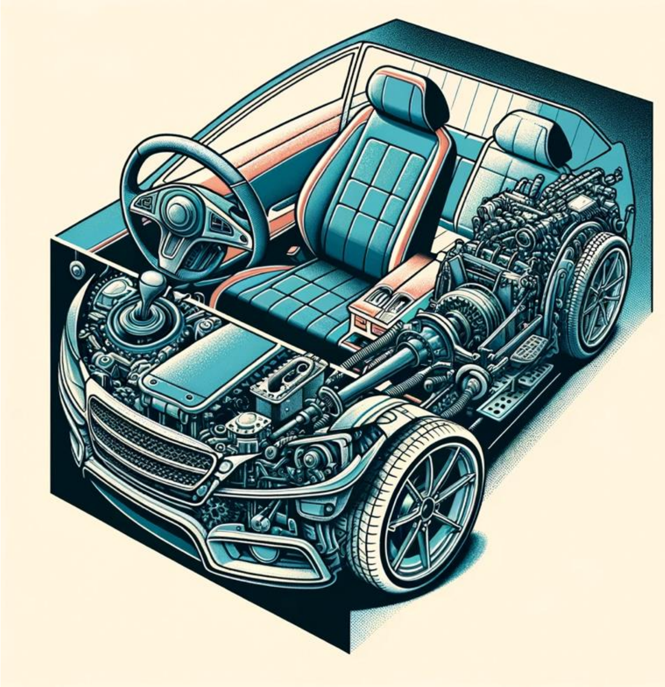
```cpp
private:
    std::string username;
    std::string password;

    // Authenticate method
    int Authenticate() {
        int valid = 0;
        if (!username.empty() && !password.empty()) {
            valid = 1;
        }
        return valid;
    }
};

int main() {
    // Example usage
    Login login("user", "pass");
    return 0;
}
```

Principles of OOP

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

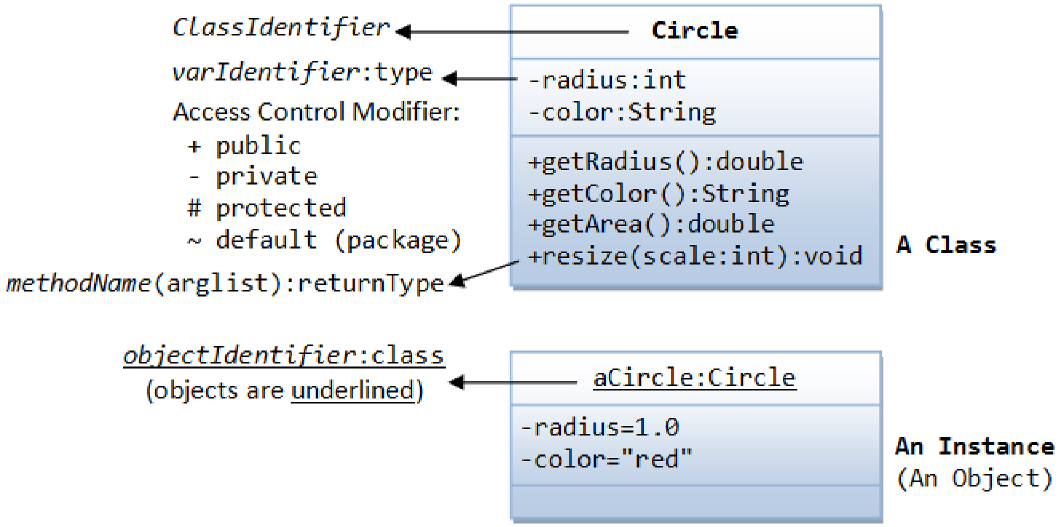
Abstraction

- Simplifying complex reality by modeling classes appropriate to the problem.
- Hiding complex implementation details and exposing only the necessary parts or information to the user.
- In OOP, abstraction is achieved through the using of abstract classes and interfaces.
- A car as an object, where its operations like start(), stop() are abstracted.



Encapsulation

- Hiding the internal details of an object from the outside world.
- Bundling data with methods that operate on the data.
- To keep both the data and the methods that manipulate the data safe from outside interference and misuse.
- A class with private data fields and public methods.



ClassIdentifier

varIdentifier:type

Access Control Modifier:
+ public
- private
protected
~ default (package)

methodName(arglist):returnType

Circle

-radius:int
-color:String

+getRadius():double
+getColor():String
+getArea():double
+resize(scale:int):void

A Class

objectIdentifier:class
(objects are underlined)

aCircle:Circle

-radius=1.0
-color="red"

An Instance
(An Object)

Inheritance

- Mechanism where one class acquires the properties (methods and fields) of another.
- To promote code reuse and establish a subtype from an existing object.
- A base class Vehicle, and derived classes Car and Bike.

```
#include <iostream>

class Animal {
public:
    void eat() {
        std::cout << "I can eat!" << std::endl;
    }

    void sleep() {
        std::cout << "I can sleep!" << std::endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        std::cout << "I can bark!" << std::endl;
    }
};

int main() {
    Dog dog;
    dog.eat(); // Output: I can eat!
    dog.sleep(); // Output: I can sleep!
    dog.bark(); // Output: I can bark!
    return 0;
}
```

Polymorphism

- The ability of different classes to be treated as instances of the same class through inheritance.
- To allow a single interface to represent different underlying forms (data types).
- A function draw() that behaves differently for Circle, Square, Triangle classes.

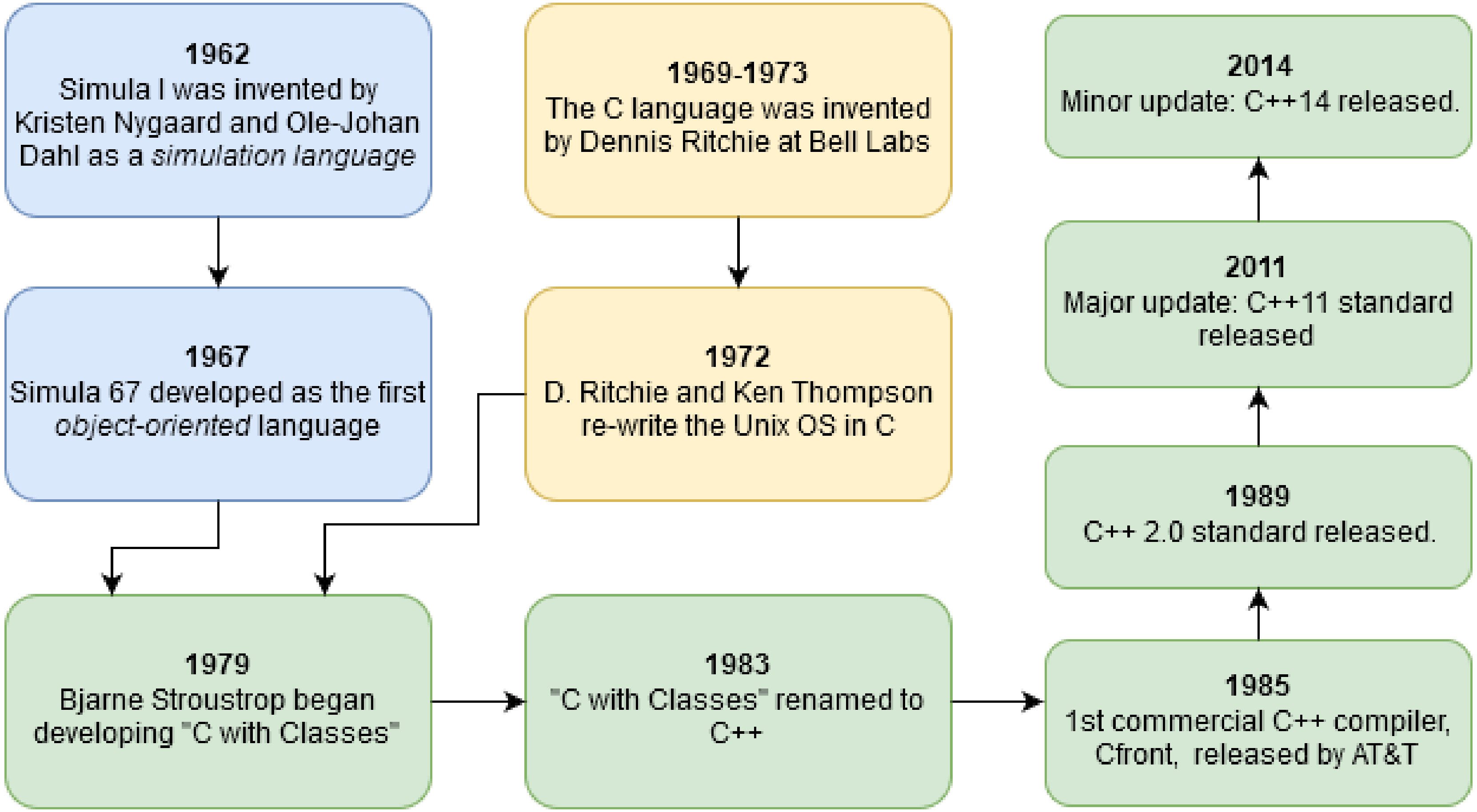
```
#include <iostream>

class Shape {
public:
    virtual void draw() {
        std::cout << "Drawing a shape" << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle" << std::endl;
    }
};

class Square : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a square" << std::endl;
    }
};

int main() {
    const int size = 2;
    Shape* shapes[2] = {new Circle(), new Square()};

    for (auto shape : shapes) {
        shape->draw();
    }
    return 0;
}
```

Quiz 1

https://docs.google.com/forms/d/e/1FAIpQLSfHtchnr9rjX6lD4pXoOgV5ZxfdlmKoNWSs0EK_Jfp81ll2VA/viewform?usp=sf_link

Getting Started

The Evolution of C++



1962
Simula I was invented by Kristen Nygaard and Ole-Johan Dahl as a *simulation language*

1967
Simula 67 developed as the first *object-oriented* language

1969-1973
The C language was invented by Dennis Ritchie at Bell Labs

1972
D. Ritchie and Ken Thompson re-write the Unix OS in C

1979
Bjarne Stroustrop began developing "C with Classes"

1983
"C with Classes" renamed to C++

1985
1st commercial C++ compiler, Cfront, released by AT&T

1989
C++ 2.0 standard released.

2011
Major update: C++11 standard released

2014
Minor update: C++14 released.

The Evolution of C++

- Early 1802s - Inception Creation by Bjarne Stroustrup at Bell Labs. C with Classes.
- 1985 - C++ Debut Officially named C++. Introduction of OOP features.
- 1998 - Standardization (C++98) First standardized version. Stability and portability.
- 2003 - C++03 Bug fixes and minor improvements.
- 2014/2017 - C++14 and C++17 Incremental updates with modern features and enhancements.
- 2020 - C++20 Modules, coroutines, concepts, ranges.

Why C++?

- Performance
- Control over System Resources
- Object-Oriented Programming (OOP)
- Compatibility with Low-Level Operations
- Extensive Libraries and Tools
- Cross-Platform Development
- Community and Support

Overview of C++

- Object oriented programming is a programming paradigm.
- Tool which is used to build more reliable and reusable system.
- Two types of programming paradigms
  - Procedure oriented programming
  - Object oriented programming

Structured Programming

- Gives importance to the logic and algorithm rather than data.
- Programs are divided in to modules.
- Independent functions (procedure) to discrete tasks.
- Do not support inheritance and polymorphism.
- In procedural languages like FORTRON, PASCAL, COBOL, C etc. a program is a list of instructions.
- Each statement in the program tells the computer to do something.
- Procedural approach has its own limitations.
  - Division into functions
  - Complex
  - Data undervalued

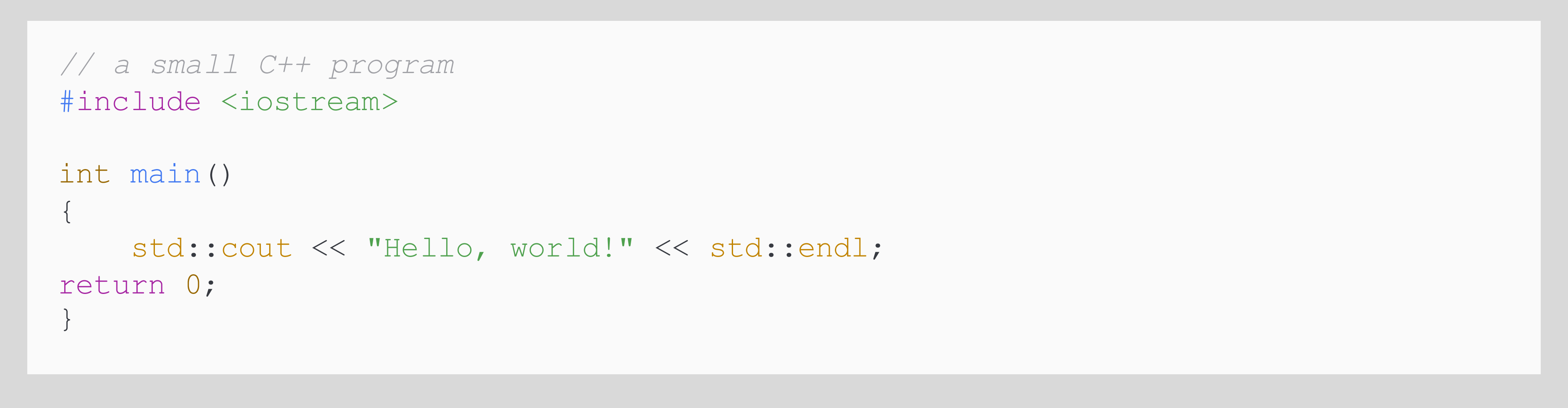
Structured Programming

- Division into functions:
  - loosely defined discipline
  - Program divided into number of functions.
  - Grouping number of functions into larger
  - entity called module / library
- Complex:
  - Large programs become complex to debug and maintain.
- Data undervalued:
  - Importance to actions and not for the data
  - Data is not secure.
  - Any function can access the global variable and changes its values .

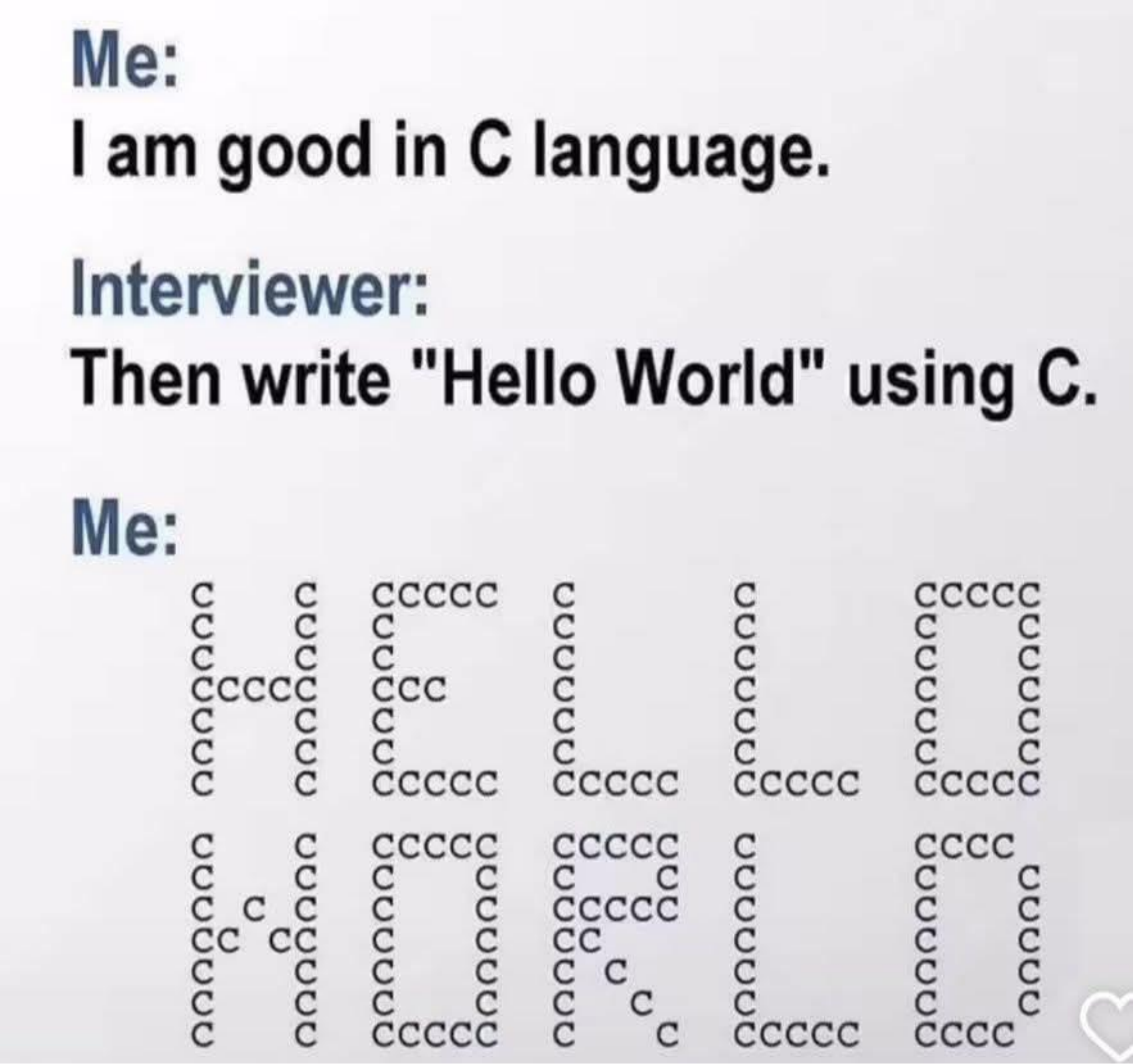
Object Oriented Programming

- To overcome the limitation of the procedural language.
- OOP languages provide the programmer, the ability to create class hierarchies.
- Programmer can create modular and reusable code.
- The fundamental idea behind object oriented languages is to combine into a single unit, both data and functions. - object .
- Functions inside the objects are called method as member functions and these method provide the
- If you want to read a data item in an object then you call the method in the object.
- It will read that data item in an object and returns the value. You can not access the data directly. Data is hidden (encapsulated).

A Small C++ Program



```
// a small C++ program
#include <iostream>

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



Me:

I am good in C language.

Interviewer:

Then write "Hello World" using C.

Me:

```
C   C  CCCCC  C      C      CCCCC
C   C  C      C      C      C   C
C   C  C      C      C      C   C
CCCCC  CCC    C      C      C   C
C   C  C      C      C      C   C
C   C  C      C      C      C   C
C   C  CCCCC  CCCCC  CCCCC  CCCCC

C   C  CCCCC  CCCCC  C      CCCC
C   C  C   C  C   C  C      C   C
C C C  C   C  CCCCC  C      C   C
CC CC  C   C  CC     C      C   C
C   C  C   C  C C    C      C   C
C   C  C   C  C  C   C      C   C
C   C  CCCCC  C   C  CCCCC  CCCC
```

Expressions

- An **expression** expresses what to compute
- The computation yields a **result** and may have **side effects**
- Examples:
  3 + 4 yields 7and has no side effects
- **`std::cout << "Hello"`** yields the **reference** to **`std::cout`**
- as a result of an expression (more on this later in the course)

Syntax (or Compile-Time) Errors

Typos

```
 cot << "Hello, world!\n";
cout << "Hello, world!\";
```

Violating the language rules

Compiler often catches errors, and reports them:

```
hello.cpp:7: error: 'cot' was not declared in this scope
hello.cpp:7: error: missing terminating " character
```

Common strategy for fixing syntax errors is to start fixing them from the very first error at the top down to the bottom of the source code

Misspelling Words

Misspelled words are often not obvious

e.g. `cot` vs `cout`, `std:cout` vs `std::cout`

C++ is case-sensitive:

All keywords and most library functions and definitions are lower-case

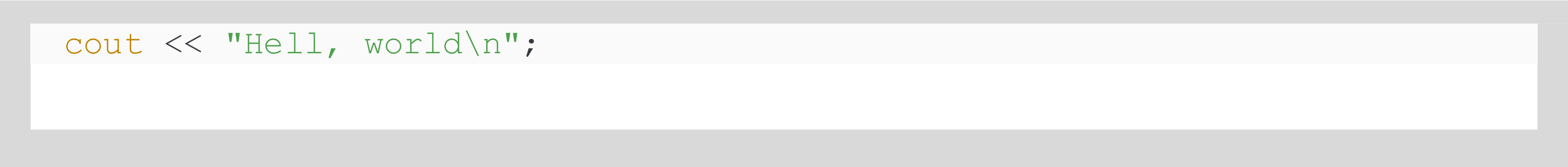
Logic or Run-Time Errors

Program compiles fine (the code is legal)

Program does not do what it is supposed to do

Much harder to find

Need a test run to find the error



```
cout << "Hell, world\n";
```

Overflow and Round-Off Errors

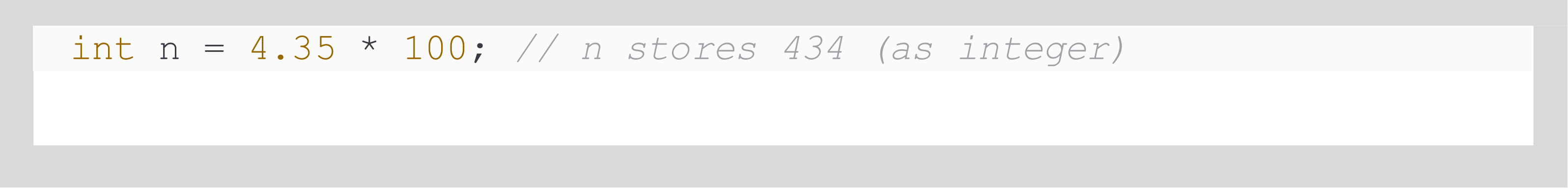
Computation result is outside of the numeric

range either the value is too big or too small

Example:

(1.0 / 3) are **truncated** when assigned to an **`int`**

Rounding errors



```
int n = 4.35 * 100; // n stores 434 (as integer)
```

Fixing Errors

Testing

- Validating program correctness
- Is very important for ensuring software quality

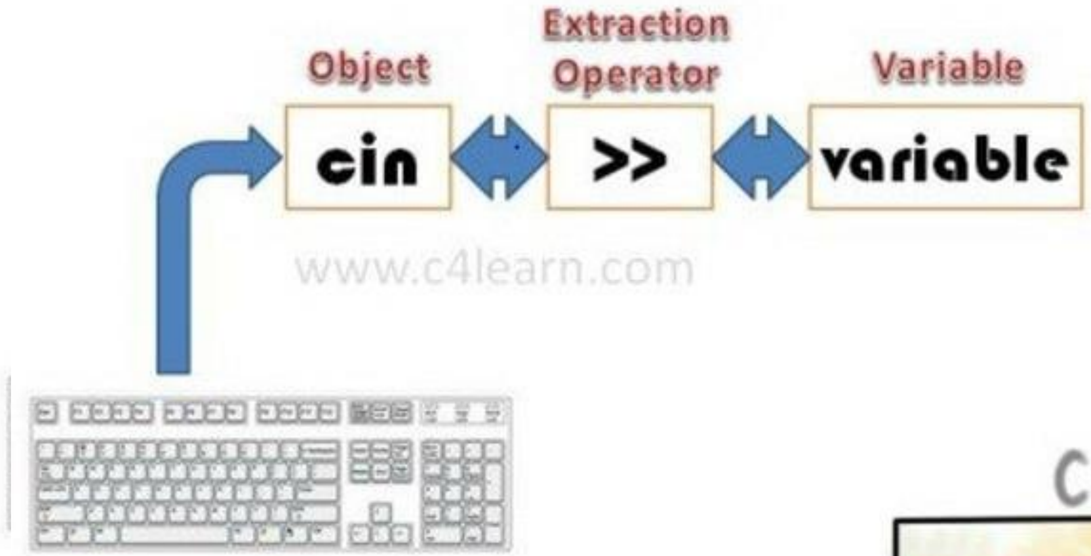
Debugging

- Finding the source of an error
- A **debugger** is a handy tool for the task

Defensive Programming

- When possible, minimize errors even before compiling the program
- Strategies include crafting programs to limit, minimize, localize errors if they do occur

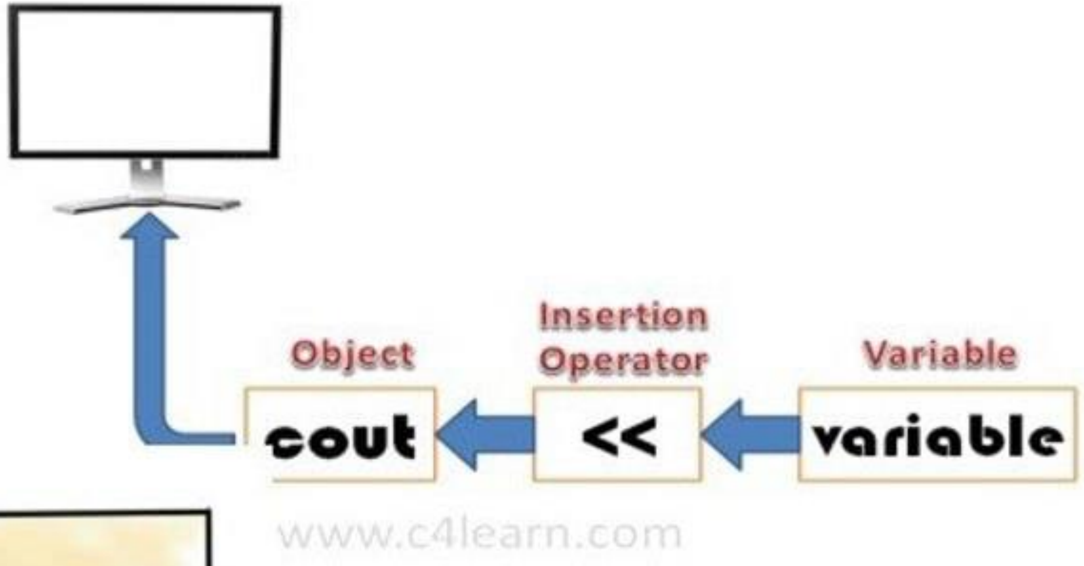
CLI
Input



Object | Extraction Operator | Variable

cin >> variable

www.c4learn.com

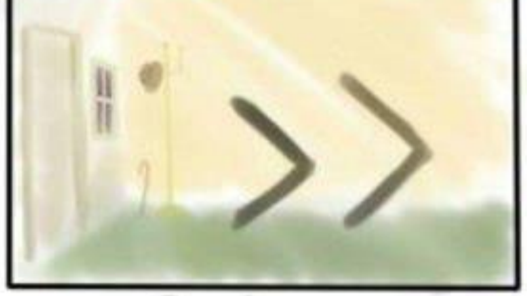


Object | Insertion Operator | Variable

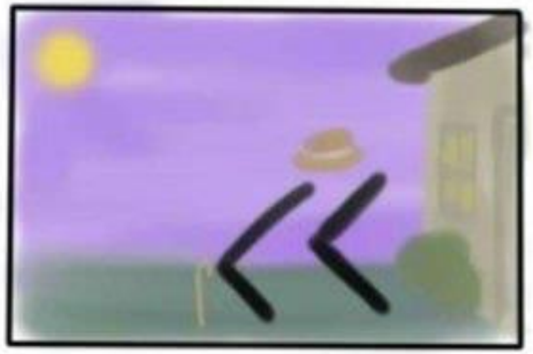
cout << variable

www.c4learn.com

Cin

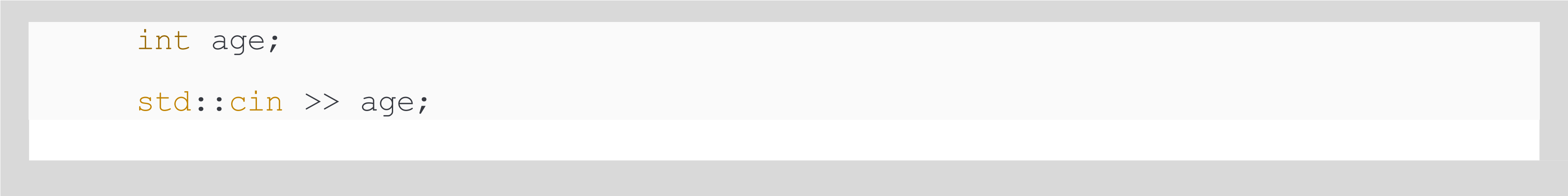


Cout



Standard Input (CLI)

- The input stream defines **>>** (**stream-extraction** operator) for expressing the data extraction operation
- Use **`std::cin`** to read an input from the console (standard input)



```
int age;
std::cin >> age;
```

- Each extraction will try to convert the input into the appropriate value of the target variable (or object)
- The data conversion process varies depending on the type of the target object

Standard Input (CLI)

- Similar to output stream the **>>** operator can be chained to read multiple values sequentially

```
      cin >> pennies >> nickels >> dimes >> quarters;
```

- Inputs are separated by one or more white space characters (including new-lines)

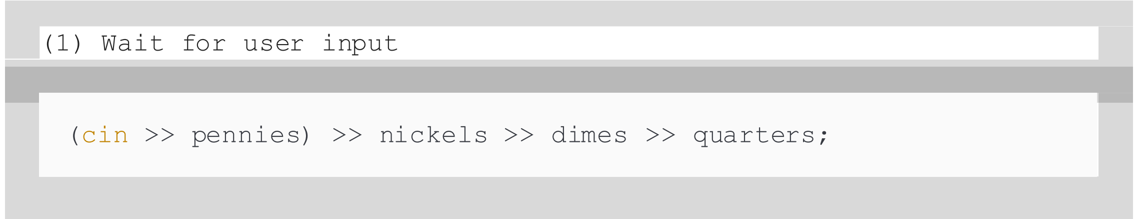
```
81  0 4 35
```

```
81
 0
  4
35
```

Input Extraction (1)

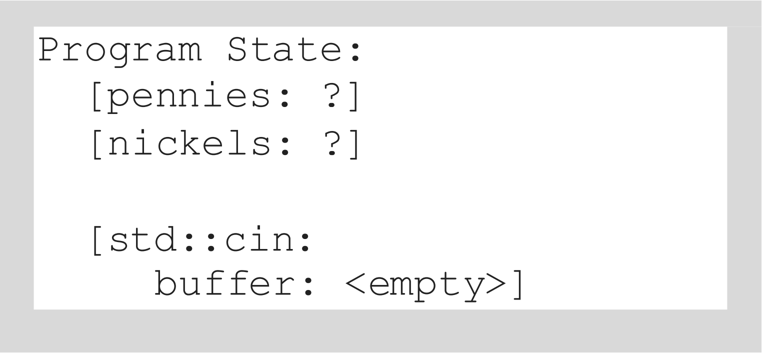
The processing of the extraction begins with the first **>>** operator

- The program will wait for user input at the first operator as the interal buffer inside **`cin`** will be initially "empty"
- Demonstration:



```
(1) Wait for user input
```

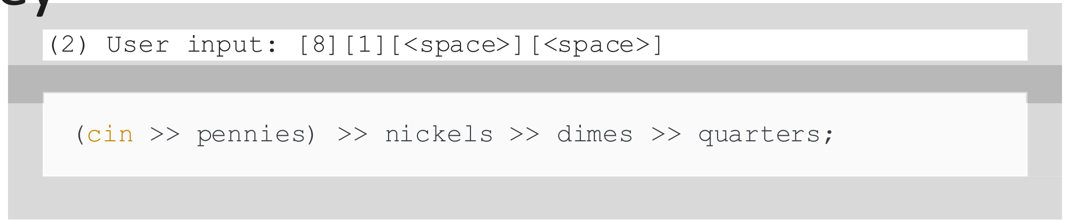
```
(cin >> pennies) >> nickels >> dimes >> quarters;
```



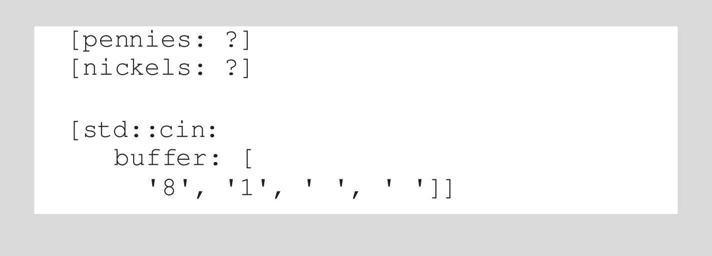
```
Program State:
  [pennies: ?]
  [nickels: ?]

  [std::cin:
     buffer: <empty>]
```

Input Extraction (2)

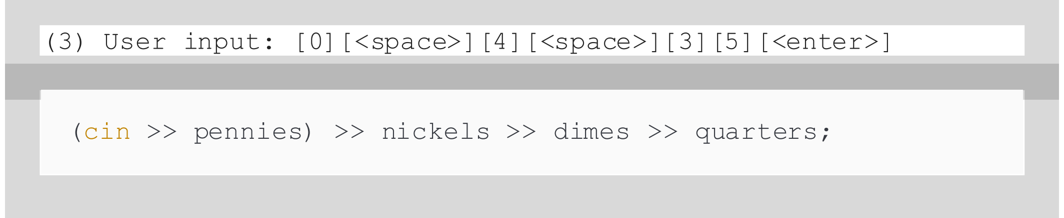
- The first **>>** will finished only when user hit the enter key



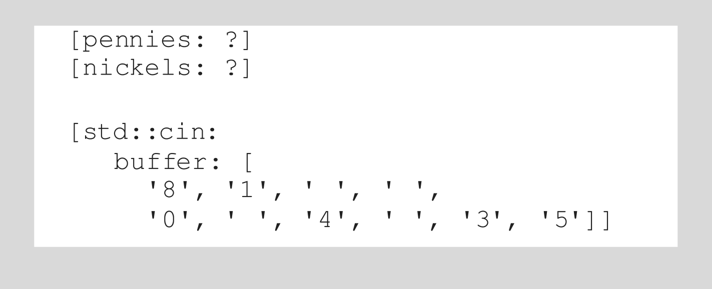
```
(2) User input: [8][1][<space>][<space>]

(cin >> pennies) >> nickels >> dimes >> quarters;
```



```
[pennies: ?]
[nickels: ?]

[std::cin:
   buffer: [
      '8', '1', ' ', ' ']]
```



```
(3) User input: [0][<space>][4][<space>][3][5][<enter>]

(cin >> pennies) >> nickels >> dimes >> quarters;
```



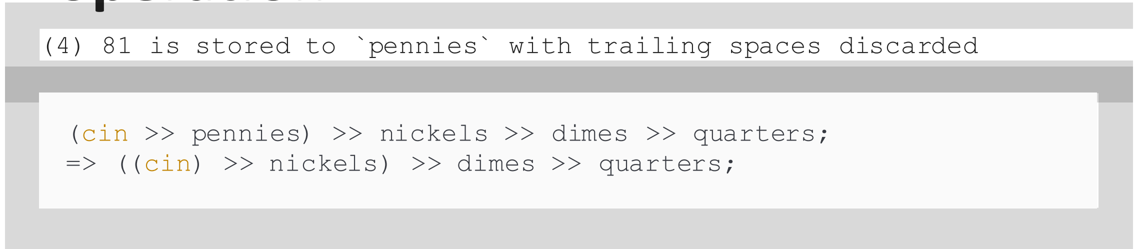
```
[pennies: ?]
[nickels: ?]

[std::cin:
   buffer: [
      '8', '1', ' ', ' ',
      '0', ' ', '4', ' ', '3', '5']]
```

- After the enter key is hit, the internal input buffer will be filled with characters

Input Extraction (3)

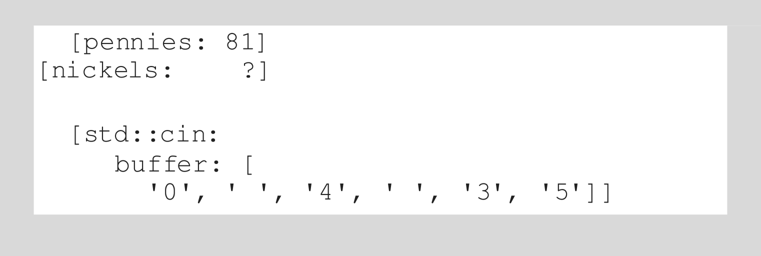
- Agter completing the conversion, value will be stored to the variable as a side-effect
- The operation will yield **`cin`** which is used at the left side of the next

  << operation



```
(4) 81 is stored to `pennies` with trailing spaces discarded
```

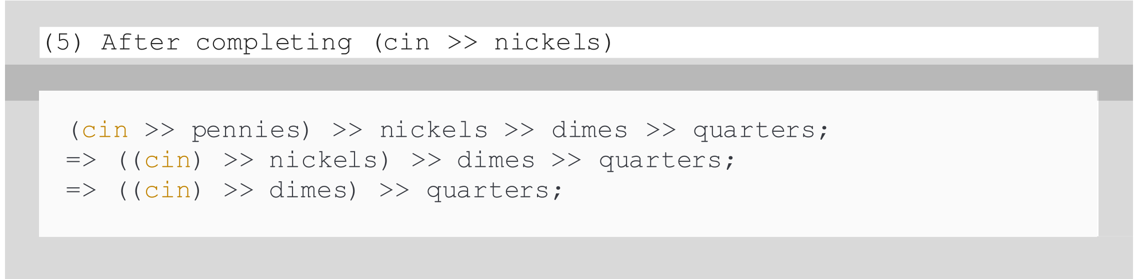
```
 (cin >> pennies) >> nickels >> dimes >> quarters;
=> ((cin) >> nickels) >> dimes >> quarters;
```



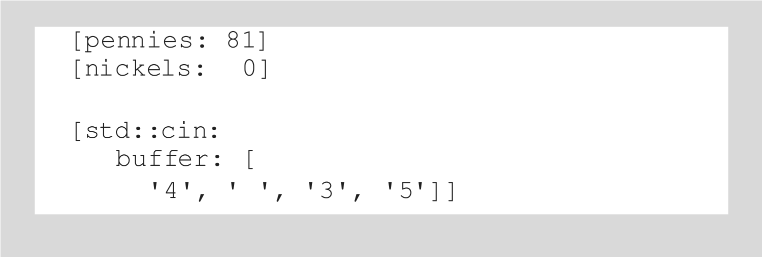
```
  [pennies: 81]
[nickels:    ?]

  [std::cin:
     buffer: [
       '0', ' ', '4', ' ', '3', '5']]
```

Input Extraction (4)

- The next **<<** will not wait for more input as the internal buffer is still having some data to extract

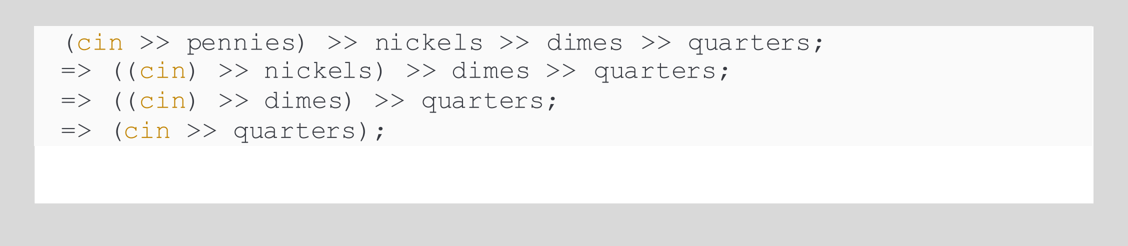


```
(5) After completing (cin >> nickels)

 (cin >> pennies) >> nickels >> dimes >> quarters;
=> ((cin) >> nickels) >> dimes >> quarters;
=> ((cin) >> dimes) >> quarters;
```

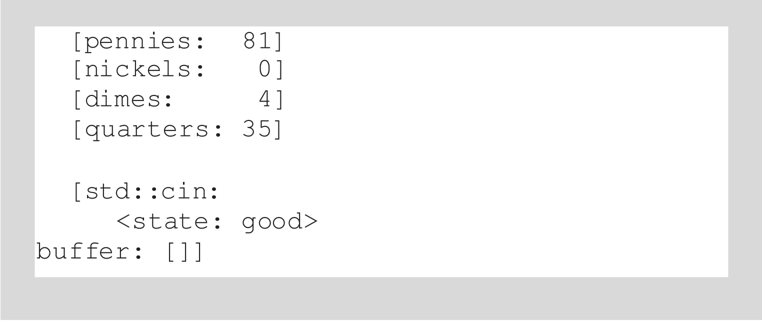


```
   [pennies: 81]
   [nickels:  0]

   [std::cin:
      buffer: [
         '4', ' ', '3', '5']]
```

- After completing the whole statement



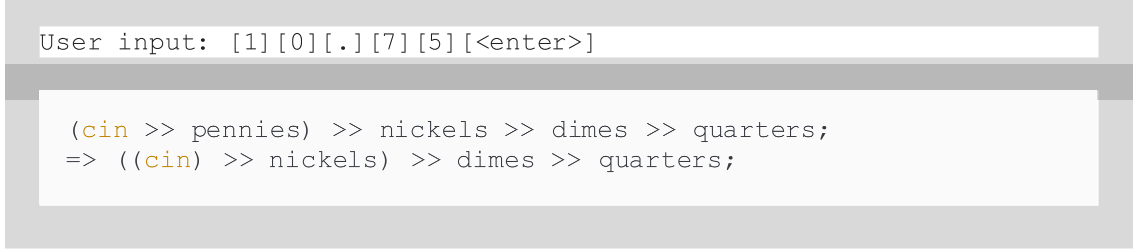
```
 (cin >> pennies) >> nickels >> dimes >> quarters;
=> ((cin) >> nickels) >> dimes >> quarters;
=> ((cin) >> dimes) >> quarters;
=> (cin >> quarters);
```



```
   [pennies:   81]
   [nickels:    0]
   [dimes:      4]
   [quarters: 35]

   [std::cin:
      <state: good>
buffer: []]
```

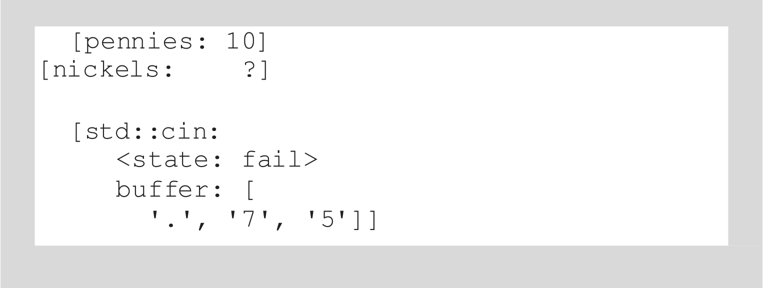
Failed Input

- If user input “10.75” instead, the extraction will fail to extract the second value and stops at reading the `'.'` character



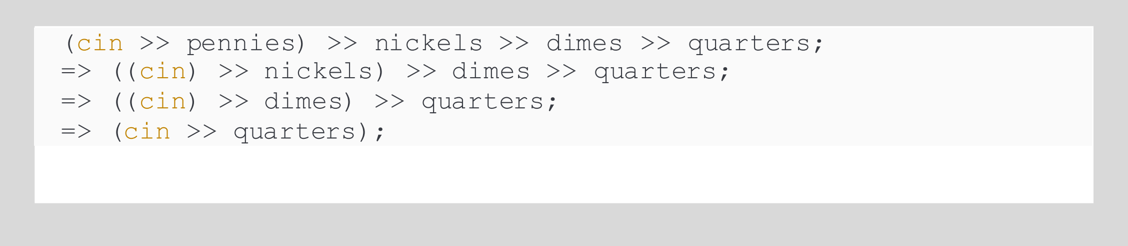
```
User input: [1][0][.][7][5][<enter>]
```

```
 (cin >> pennies) >> nickels >> dimes >> quarters;
=> ((cin) >> nickels) >> dimes >> quarters;
```

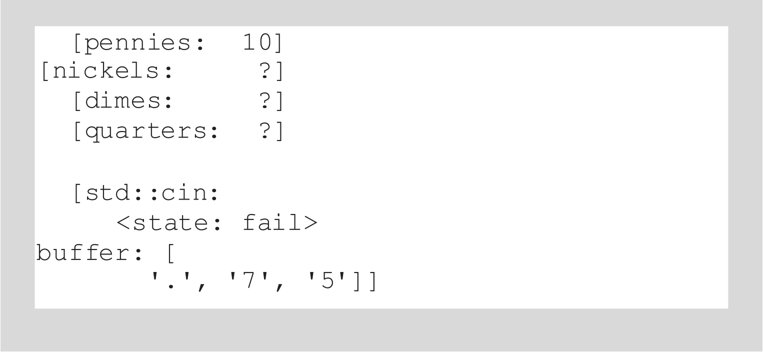


```
  [pennies: 10]
[nickels:    ?]

  [std::cin:
     <state: fail>
     buffer: [
       '.', '7', '5']]
```

- The execution continues up to the end of statement without storing other values and the stream is now in a "fail" state



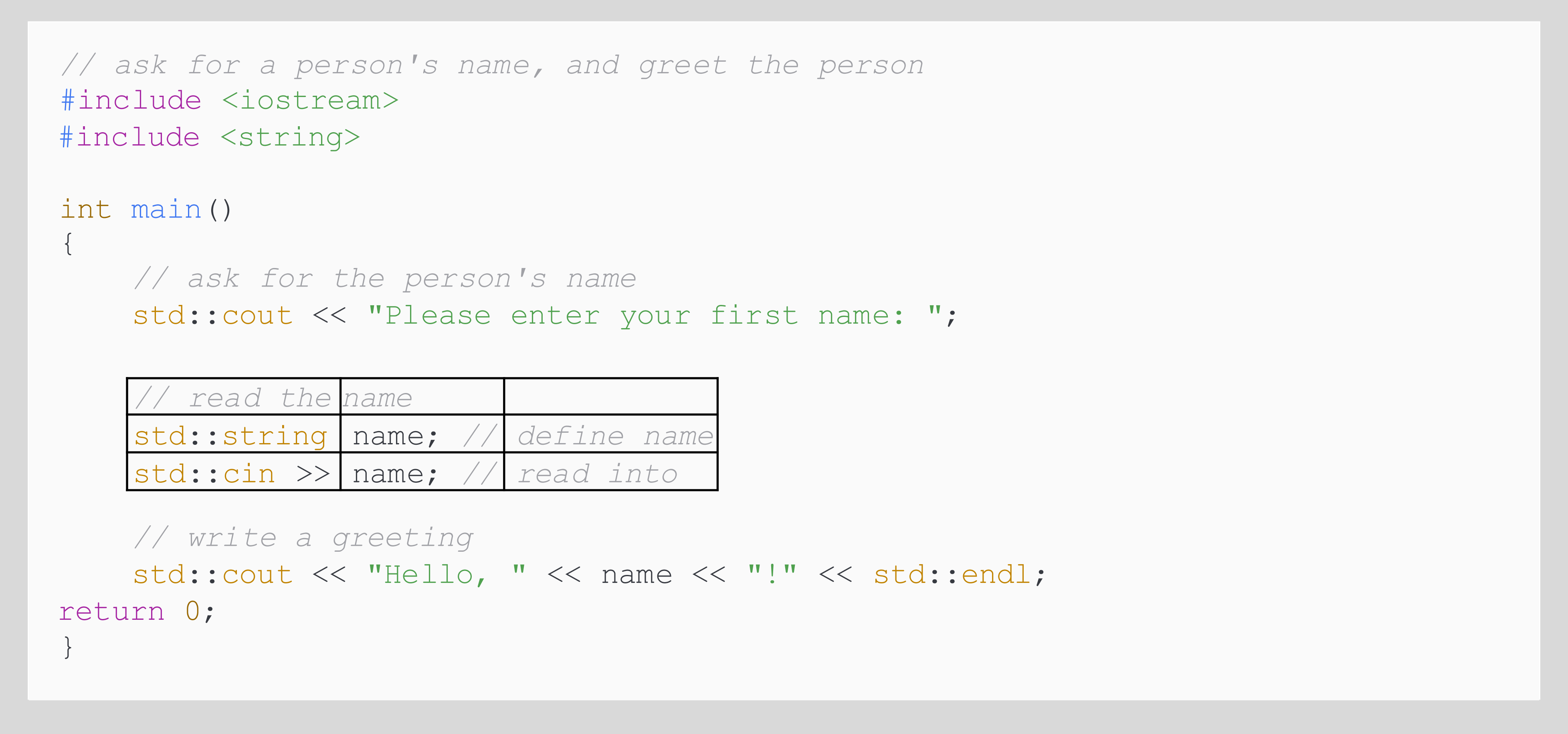
```
 (cin >> pennies) >> nickels >> dimes >> quarters;
=> ((cin) >> nickels) >> dimes >> quarters;
=> ((cin) >> dimes) >> quarters;
=> (cin >> quarters);
```



```
  [pennies:   10]
[nickels:      ?]
  [dimes:      ?]
  [quarters:   ?]

  [std::cin:
     <state: fail>
buffer: [
       '.', '7', '5']]
```

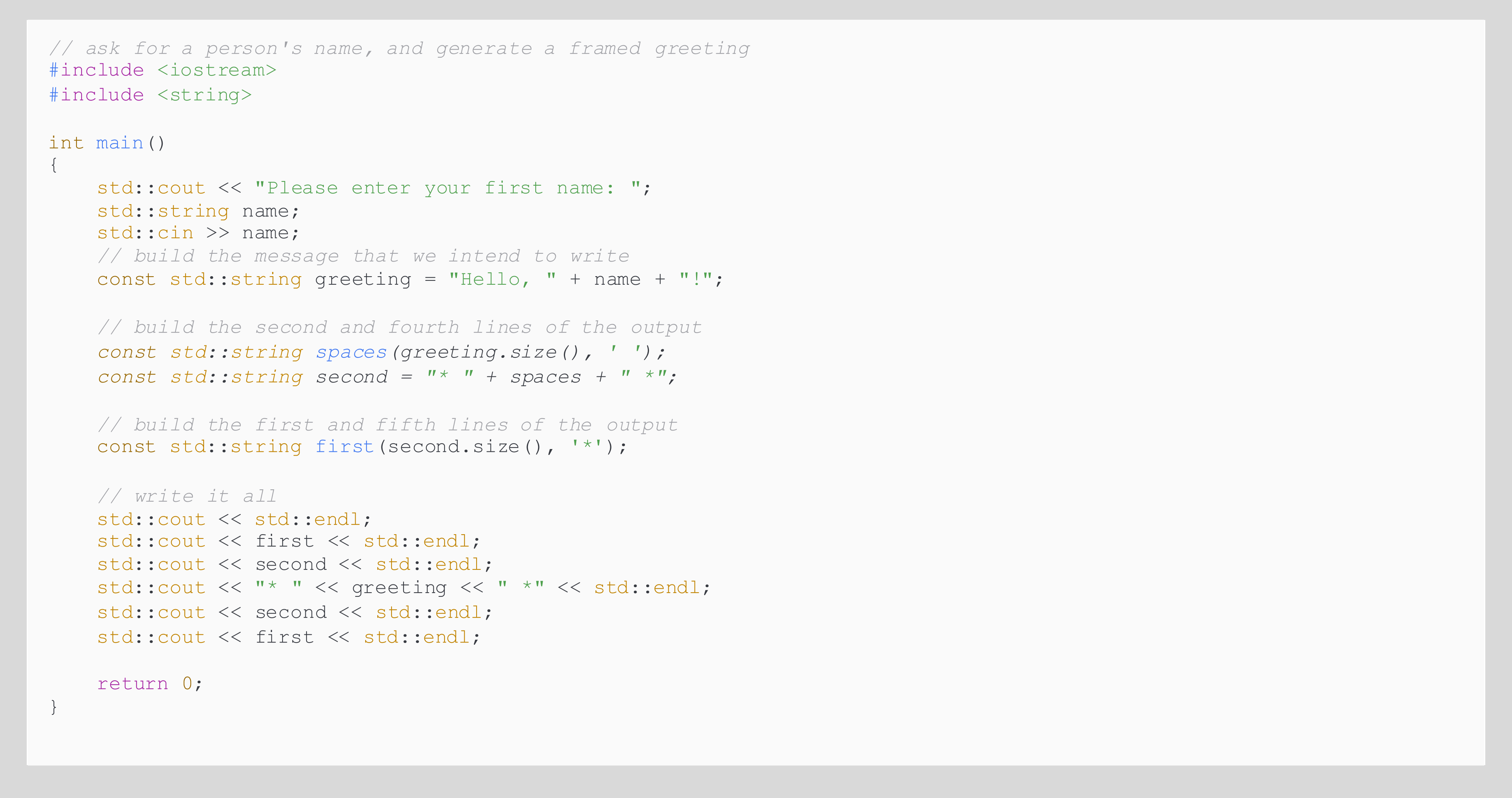
Working with Strings

Simple Interaction with CLI Input/Output



```
// ask for a person's name, and greet the person
#include <iostream>
#include <string>

int main()
{
    // ask for the person's name
    std::cout << "Please enter your first name: ";

    // read the name
    std::string name; // define name
    std::cin >> name; // read into

    // write a greeting
    std::cout << "Hello, " << name << "!" << std::endl;
return 0;
}
```

Framing a Name

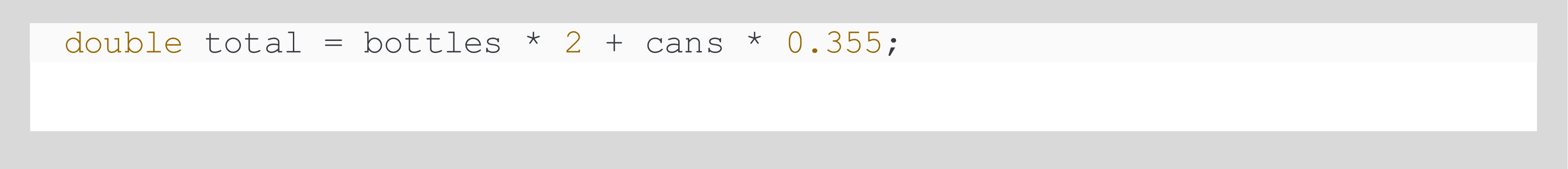


```cpp
// ask for a person's name, and generate a framed greeting
#include <iostream>
#include <string>

int main()
{
    std::cout << "Please enter your first name: ";
    std::string name;
    std::cin >> name;
    // build the message that we intend to write
    const std::string greeting = "Hello, " + name + "!";

    // build the second and fourth lines of the output
    const std::string spaces(greeting.size(), ' ');
    const std::string second = "* " + spaces + " *";

    // build the first and fifth lines of the output
    const std::string first(second.size(), '*');

    // write it all
    std::cout << std::endl;
    std::cout << first << std::endl;
    std::cout << second << std::endl;
    std::cout << "* " << greeting << " *" << std::endl;
    std::cout << second << std::endl;
    std::cout << first << std::endl;

    return 0;
}
```

Constants

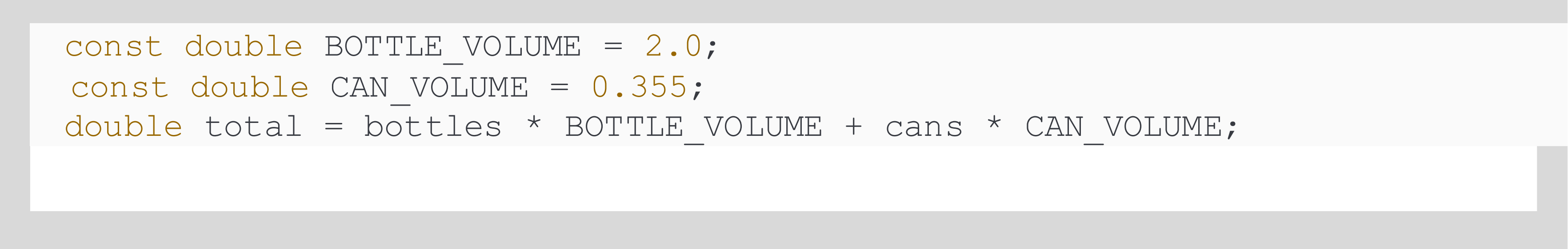
Constants

- Descriptive identifiers make a program easier to read
- The same is true for constants
- What is **0.355** in the following statement?



```
double total = bottles * 2 + cans * 0.355;
```

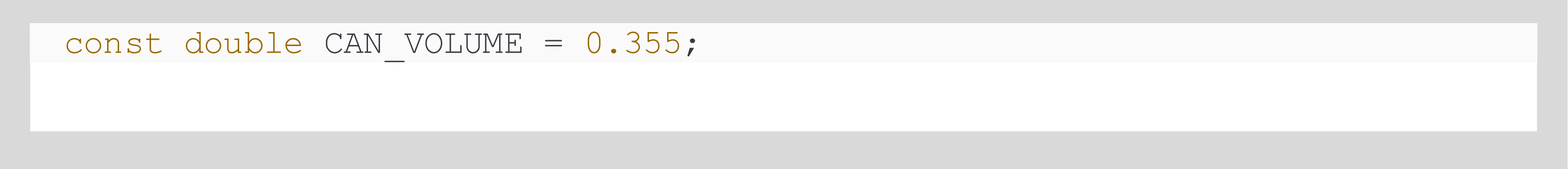
- **0.355 is the number of liters in a 12 oz. can**
- Defining constants can make the code more readable



```
const double BOTTLE_VOLUME = 2.0;
const double CAN_VOLUME = 0.355;
double total = bottles * BOTTLE_VOLUME + cans * CAN_VOLUME;
```

Constants

- Declared with the keyword `const`
- A constant can never be changed It
- must be initialized at definition:



```
const double CAN_VOLUME = 0.355;
```

- Often written all in capital letters

Constants — volume.cpp

```
#include <iostream>

using namespace std; int main()
{
    double bottles;
    cout << "How many bottles do you have? ";
    cin >> bottles;

    double cans;
    cout << "How many cans do you have? ";
    cin >> cans;

    const double BOTTLE_VOLUME = 2.0;
    const double CAN_VOLUME = 0.355;

    double total = bottles * BOTTLE_VOLUME + cans * CAN_VOLUME; cout << "The
total volume is " << total << " liter.\n"; return 0;
}
```

Constants

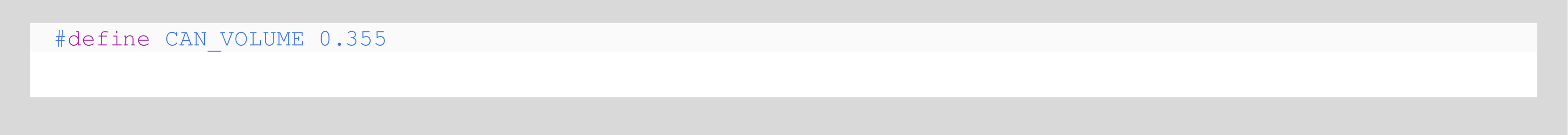
- Code is easier to read and less prone to errors Easier to
- modify/maintain code
- Consider changing bottles from 2 liters to 1/2 gallons
  - Without constants:

    Search for every 2, replace them with 1.893? With
  - With constants:

    Just update your constant once (and its associated comment)

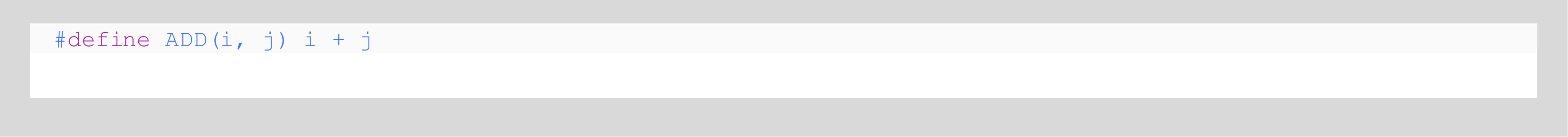
const vs #define

- In C, **#define** is often used to define a constant



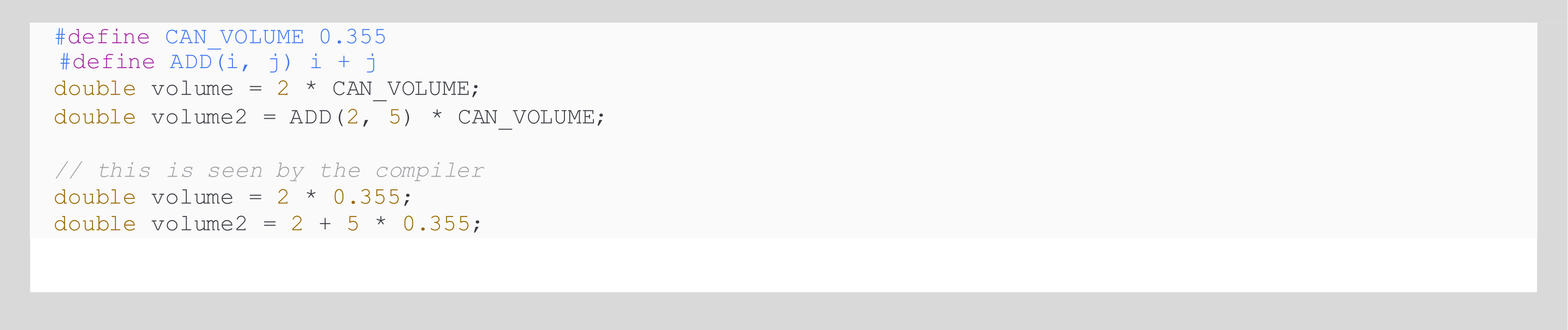
```
#define CAN_VOLUME 0.355
```

- Or used to define a preprocessor macro



```
#define ADD(i, j) i + j
```

- **#define is processed by the preprocessor before the compilation process so there is no compile-time checking and should be avoided in C++**



```
#define CAN_VOLUME 0.355
 #define ADD(i, j) i + j
double volume = 2 * CAN_VOLUME;
double volume2 = ADD(2, 5) * CAN_VOLUME;

// this is seen by the compiler
double volume = 2 * 0.355;
double volume2 = 2 + 5 * 0.355;
```

`const` vs `#define`

- In C++, we prefer **`const`** to `#define`
  - Smaller code
  - Feel natural with the rest of the language
  - Well supported by the C++ type system
- As an alternative to the preprocessor macro,
  `inline` function is almost always a better substitute in C++
  (We will explore `inline` function later)

Input/Output Manipulators

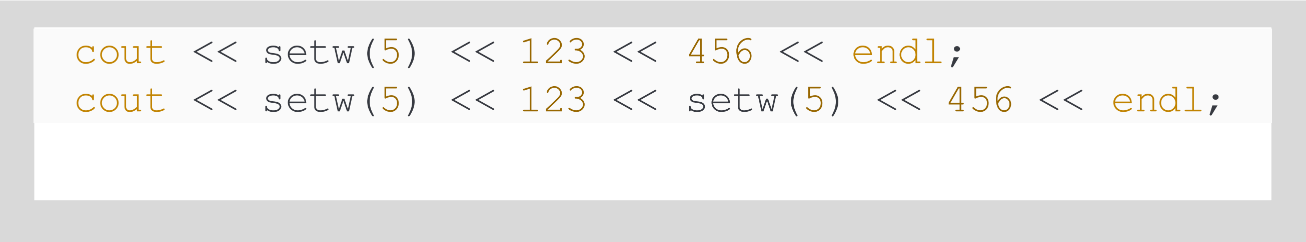
Input/Output Manipulators

- A helper functions that make it possible to control input/output streams (i.e. `std::cin` and `std::cout`)
- Example output produced using output stream + manipulators
- See https://en.cppreference.com/w/cpp/io/manip for a reference

setw() — <iomanip>

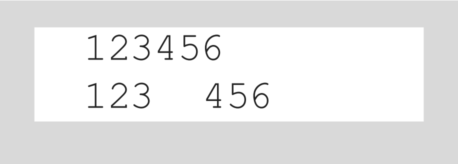
- **setw(w)** **allows you to set the width w of a field for the next output (and ONLY the next output)**

Example:



```
cout << setw(5) << 123 << 456 << endl;
cout << setw(5) << 123 << setw(5) << 456 << endl;
```

Output:



```
  123456
  123  456
```

left and right — <ios>

- **left and right are used to arrange the position for the output**

Example:

```
  cout << setfill('*');
  cout << left << setw(5) << 123 << 456 << endl;
cout << right << setw(5) << 123 << 456 << endl;
  cout << left << setw(5) << 123 << setw(8) << 456 << endl;
  cout << right << setw(5) << 123 << setw(8) << 456 << endl;
```

Output:

```
123**456
**123456
123**456*****
**123*****456
```

setprecision() — <iomanip>

- **setprecision(n) sets the precision of an output stream to n (the number of digits after decimal) for the floating-point number output**

Example:

```
for (int x = 1; x < 11; ++x) {
    cout << "precision " << x << ":\t"
         << setprecision(x) << 12.3456
        << endl;
}
```

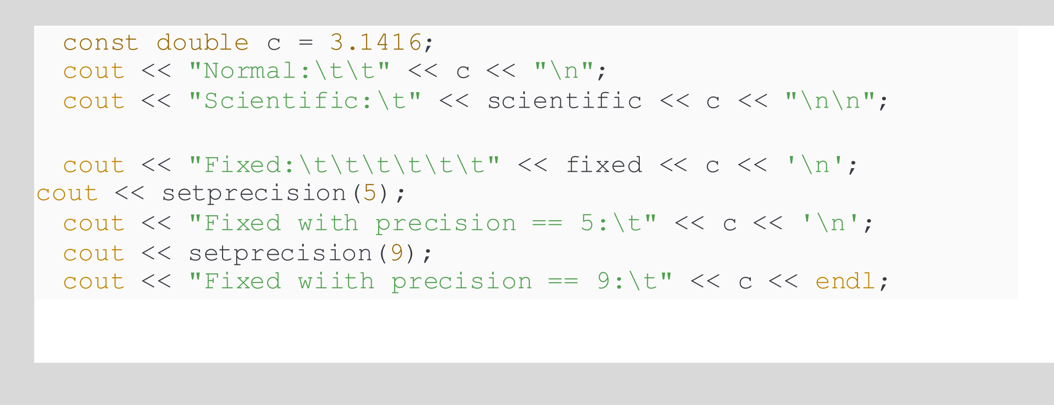
Output:

```
precision 1:    1e+01
precision 2:    12
precision 3:    12.3
precision 4:    12.35
precision 5:    12.346
precision 6:    12.3456
precision 7:    12.3456
precision 8:    12.3456
precision 9:    12.3456
precision 10:   12.3456
```

fixed and scientific — <ios>

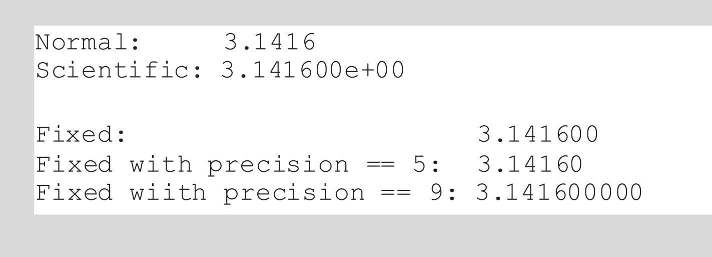
- **fixed displays trailing zeroes up to the current precision**
- **scientific changes the number format to scientific format**

Example:



```
  const double c = 3.1416;
  cout << "Normal:\t\t" << c << "\n";
  cout << "Scientific:\t" << scientific << c << "\n\n";

  cout << "Fixed:\t\t\t\t\t\t" << fixed << c << '\n';
cout << setprecision(5);
  cout << "Fixed with precision == 5:\t" << c << '\n';
  cout << setprecision(9);
  cout << "Fixed wiith precision == 9:\t" << c << endl;
```

Output:

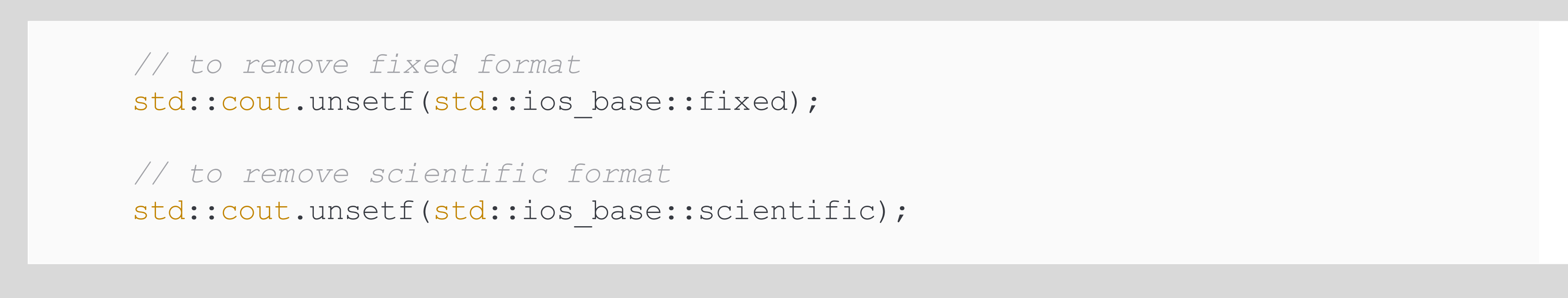


```
Normal:     3.1416
Scientific: 3.141600e+00

Fixed:                      3.141600
Fixed with precision == 5:  3.14160
Fixed wiith precision == 9: 3.141600000
```

Output Stream Number Format

	Normal mode	Fixed mode	Scientific mode
Effect	Normally display value (0 are not added)	Control the number of digits after decimal point (0 might be added)	Display value in the scientific format
Precision	All digits	Digits after decimal point	Digits after decimal point
`123.45` with precision of 4	`123.5`	`123.4500`	`1.2345e+002`
`123.45` with precision of 6	`123.5`	`123.4500`	`1.2345e+002`

Note: use `cout.unsetf(flags)` to clear the number format



```
// to remove fixed format
std::cout.unsetf(std::ios_base::fixed);

// to remove scientific format
std::cout.unsetf(std::ios_base::scientific);
```

dec, hex, oct and showbase — <ios>

```
cout << hex << 31 << endl;
cout << showbase << hex << 31 << endl;
```

Output:

```
1f
0x1f
```

```
cout << oct << 31 << endl;
cout << showbase << oct << 31 << endl;
```

Output:

```
37
037
```

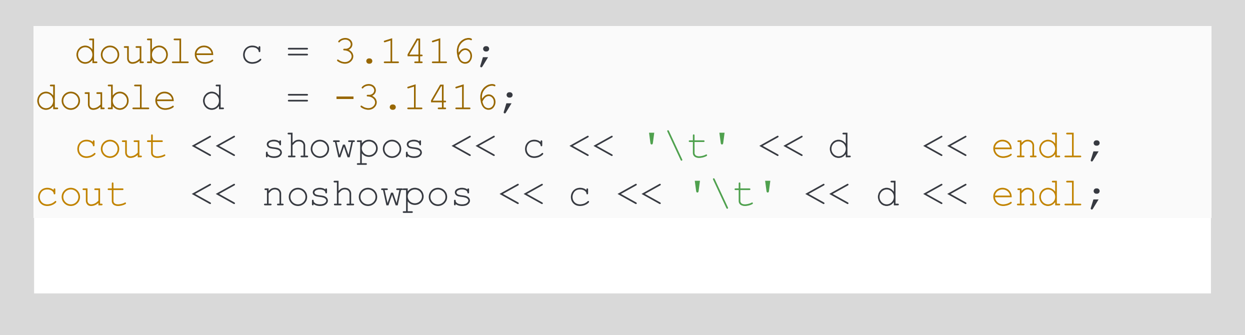
cout	<<	dec << 0x1F << endl;
cout	<<	dec << 037 << endl;

Output:

```
31
31
```

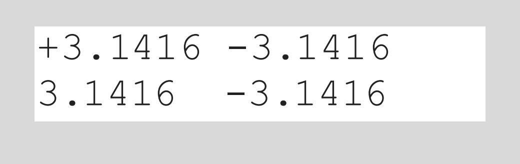
showpos and noshowpos — <ios>

- **showpos displays the + sign for the non-negative number**



```
  double c = 3.1416;
double d   = -3.1416;
  cout << showpos << c << '\t' << d   << endl;
cout   << noshowpos << c << '\t' << d << endl;
```

Output:



```
+3.1416 -3.1416
3.1416  -3.1416
```

showpoint and noshowpoint — <ios>

- **showpoint displays the decimal point in a floating-point value**
- **noshowpoint displays the decimal point when it is necessary**

```
  double f   = 10.0;
cout   << f << endl;
  cout << showpoint << f << endl;
cout   << noshowpoint << f << endl;
```

Output:

```
10
10.0000
10
```

boolalpha and noboolalpha — <ios>

```
bool a = true;
cout <<   a
     << endl;
cout << boolalpha << a << endl;
a = false;
cout << a << endl;
cout << noboolalpha << a << endl;
```

Output:

```
1
true
false
0
```

List of Manipulators

Manipulator	Purpose
setw(w)	Set the width **w** of a field for the next output
left	Set left position arrangement for the output
right	Set right position arrangement for the output
setprecision(n)	sets the precision of an output stream to **n** for the floating-point number output
fixed	Insert float-point values in fixed format
scientific	Insert float-point values in scientific format
uppercase	Use uppercase letters on insertion
nouppercase	Don’t use uppercase letters on insertion

List of Manipulators (Cont')

Manipulator	Purpose
dec	Insert values in decimal (base 10) format
hex	Insert values in hexadecimal (base 16) format
oct	Insert values in octal (base 8) format
showbase	Show the base of value
noshowbase	Do not prefix a value with its base
showpos	Insert the **+** sign before non-negative number
noshowpos	Do not insert the **+** sign before non-negative number
showpoint	Show the decimal point in a floating-point value
noshowpoint	The decimal point is only shown when necessary
boolalpha	Insert boolean value as text (`true` or `false`)
noboolalpha	Insert boolean value as number (`1` or `0`)

&A